

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/228107078>

Lambda-Calculus

Chapter · September 2008

DOI: 10.1002/9780470050118.ecse212

CITATIONS

0

READS

226

2 authors:



[Mariangiola Dezani-ciancaglini](#)

Università degli Studi di Torino

278 PUBLICATIONS 5,469 CITATIONS

SEE PROFILE



[J. Roger Hindley](#)

Swansea University

58 PUBLICATIONS 3,384 CITATIONS

SEE PROFILE

Lambda-Calculus

Mariangiola Dezani-Ciancaglini,
Dipartimento di Informatica, University of Turin, Italy,

J. Roger Hindley,
Department of Mathematics, Swansea University, Wales, U.K.

November 8, 2007

The λ -calculus is a primitive programming language which is *higher-order*, in the sense that in it one can write programs which modify other programs.

It was invented in 1928 by an American logician, Alonzo Church, as part of a logical system in which he hoped to describe the foundations of mathematics. This larger system turned out to be inconsistent and was abandoned, but the λ -calculus at its core survived and Church's group found that, using it, they could give a precise definition of what computability meant, and from this they discovered the first rigorous proof that certain important problems could never be solved by computer.

But until the 1970s very little higher-order computing was possible, and the computing world was dominated by first-order languages whose structure was much simpler than λ -calculus.¹

Since then, however, several higher-order programming languages have been developed. They incorporate either a form of λ -calculus or something equivalent to λ -calculus, and earlier studies of that system have helped to show what these languages can do. Roughly speaking, techniques can be tried out and developed on λ -calculus, and then applied to the more complex practical languages.

To give the reader the flavour of λ -calculus as quickly as possible, we shall describe here its simplest, 'pure', form, but with the warning that most applications use more complicated variants. We shall later sketch a few of these variants.

Further information on λ -calculus is available in many websites and books on computing, as well as in the introductory book [18] and the comprehensive work [4].

1 Grammar of λ -calculus

2

¹M. SAYS:What do you mean here? Programming language syntax is much longer than λ -calculus syntax, and the operational semantics is usually ambiguous...

²M. SAYS:I suggest instead: Syntax and Operational Semantics of λ -calculus

An arithmetical expression such as “ $x^2 + 3$ ” defines a function of x ; Church denoted this function by

$$\lambda x . x^2 + 3.$$

Associated with this notation is a rule: for all numbers a ,

$$(\lambda x . x^2 + 3)(a) = a^2 + 3.$$

Church’s notation is useful in dealing with expressions containing more than one variable: for example the expression “ $x + 2y$ ” can be viewed as defining either a function of x , with y held constant, or a function of y , with x held constant. In the λ -notation these two functions are easily distinguished; they are called, respectively,

$$\lambda x . x + 2y, \quad \lambda y . x + 2y.$$

We have

$$\begin{aligned} (\lambda x . x + 2y)(a) &= a + 2y, \\ (\lambda y . x + 2y)(a) &= x + 2a. \end{aligned}$$

Church’s λ -notation led to a formal language, whose expressions are called *λ -terms* and are intended to denote operators or programs or mathematical functions.

Definition 1.1 (λ -terms) (Cf. [18] §1.1.) Assume given an infinite sequence of variables $x, y, z, x_1, y_1, z_1, x_2, y_2, z_2, \dots$ (to denote arbitrary programs or operators). Then *λ -terms* are constructed as follows:

- (a) each variable is a λ -term;
- (b) from any λ -terms M and N , construct the new λ -term (MN) (to denote the application of operator M to input N);
- (c) from any variable x and λ -term M , construct the new λ -term $(\lambda x . M)$ (to denote the function of x that M defines).

Notation 1.2 A term (MN) is called an *application* and $(\lambda x . M)$ an *abstraction*. (In mathematics the application of M to N is usually called “ $M(N)$ ”; the reason it is called “ (MN) ” in λ -calculus is merely a historical accident.) To denote arbitrary λ -terms, we shall use capital letters. We shall write

$$M \equiv N$$

to mean that M is the same term as N . Parentheses and repeated λ ’s will often be omitted in such a way that, for example,

$$MNPQ \equiv (((MN)P)Q), \quad \lambda xyz . MN \equiv (\lambda x . (\lambda y . (\lambda z . (MN)))).$$

Examples of λ -terms (letting x, y, z be any three distinct variables):

- (a) $(\lambda x . (xy))$, (c) $(x (\lambda x . (\lambda x . x)))$,
- (b) $((\lambda y . y)(\lambda x . (xy)))$, (d) $(\lambda x . (yz))$.

In (c) there are two occurrences of λx in one term; this is allowed by the definition of “ λ -term”, though discouraged in practice. In (d) there is a term of form $(\lambda x.M)$ such that x does not occur in M ; this is allowed, and such terms denote constant-functions.

To show how λ -terms are used as programs, some more apparatus is needed. We shall just give a brief sketch.

Definition 1.3 (Free and bound variables) (Cf. [18] §1.11.) Any occurrence of a variable x in a term $\lambda x.M$ is said to be *bound* by the λx . The x in λx is said to be *binding and bound*. Any non-bound occurrence in a term is said to be *free*. The set of all variables that occur free in a term P is called

$$FV(P).$$

A *combinator* or *closed term* is a term in which no variable occurs free.

Warning. When free or bound variables are mentioned, it is really *occurrences* of variables that are meant. A variable can have bound occurrences and free occurrences in the same term. For example, consider the term

$$P \equiv (\lambda v.x)(\lambda y.yx(\lambda x.yvx)).$$

In this term, the leftmost v is bound and binding, the other v is free, the leftmost two x ’s are free, the other two x ’s are bound, and all three y ’s are bound. Also

$$FV(P) = \{v, x\}.$$

Definition 1.4 (Substitution) (Cf. [18] §1.12.) For any terms M , N and any variable x , define $[N/x]M$ to be the result of substituting N for each free occurrence of x in M , and changing any λy ’s in M to prevent variables free in N from becoming bound in $[N/x]M$. In detail:

- (a) $[N/x]x \equiv N$;
 - (b) $[N/x]y \equiv y$ if $y \neq x$;
 - (c) $[N/x](PQ) \equiv ([N/x]P[N/x]Q)$;
 - (d) $[N/x](\lambda x.P) \equiv \lambda x.P$;
 - (e) $[N/x](\lambda y.P) \equiv \lambda y.P$ if $x \notin FV(P)$;
 - (f) $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$ if $x \in FV(P)$ and $y \notin FV(N)$;
 - (g) $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$ if $x \in FV(P)$ and $y \in FV(N)$;
- (Here z is a variable chosen to be $\notin FV(NP)$.)

Example. Let $M \equiv \lambda y.yx$.

$$\begin{aligned} \text{If } N \equiv xv : \quad & [(xv)/x](\lambda y.yx) \equiv \lambda y.[(xv)/x](yx) && \text{by (f)} \\ & \equiv \lambda y.y(xv) && \text{by (a)–(c).} \\ \text{If } N \equiv xy : \quad & [(xy)/x](\lambda y.yx) \equiv \lambda z.[(xy)/x](zx) && \text{by (g)} \\ & \equiv \lambda z.z(xy) && \text{by (a)–(c).} \end{aligned}$$

Remark. If (g) were omitted from the definition of substitution, then we would have the undesirable fact that, although $\lambda v.x$ and $\lambda y.x$ both denote the same operator (the constant-operator whose output is always x), they would come to denote different operators when v was substituted for x :

$$[v/x](\lambda y.x) \equiv \lambda y.v, \quad [v/x](\lambda v.x) \equiv \lambda v.v.$$

Informally speaking, terms that differ only by changing bound variables have the same meaning. For example, $\lambda x.x$ and $\lambda y.y$ both denote the identity-operator. The process of changing bound variables is defined formally as follows.

Definition 1.5 (Changing bound variables, α -conversion) (Cf. [18] §1.16.)
If $y \in FV(M)$, we say

$$(\alpha) \quad \lambda x.M \equiv_{\alpha} \lambda y.[y/x]M.$$

If P changes to Q by a finite (perhaps empty) series of replacements of form (α) , we say

$$P \equiv_{\alpha} Q.$$

The relation \equiv_{α} can be proved symmetric; i.e. if $P \equiv_{\alpha} Q$ then $Q \equiv_{\alpha} P$, cf. [18] §1.17.

The λ -calculus analogue of computation is defined as follows.

Definition 1.6 (β -contraction, β -reduction) (Cf. [18] §1.24.) A term of form

$$(\lambda x.M)N$$

is called a β -redex. (It represents an operator applied to an input.) If it occurs in a term P , and we replace one occurrence of it by

$$[N/x]M,$$

then we say that we have *contracted* that occurrence of it. A finite (perhaps empty) or infinite series of contractions and changes of bound variables is called a β -reduction. If it is finite and changes P to Q , we say that P β -reduces to Q or

$$P \rightarrow_{\beta} Q.$$

*Example.*³ Here are some reductions; the redex contracted at each step is underlined. In (c) the reduction is infinite, even though the term never changes.

$$\begin{aligned} \text{(a)} \quad & \underline{(\lambda x.((\lambda y.xy)u))(\lambda v.v)} \rightarrow_{\beta} \underline{(\lambda y.(\lambda v.v)y)u} \\ & \rightarrow_{\beta} \underline{(\lambda v.v)u} \\ & \rightarrow_{\beta} u. \end{aligned}$$

³M. SAYS: It would be better to introduce one-step β -reduction? It is used in example (c) and I think it would be clearer to use it in the other examples and in discussing evaluation strategies

$$\begin{array}{ll}
\text{(b)} & (\lambda x. ((\lambda y. xy)u))(\lambda v. v) \rightarrow_{\beta} (\lambda x. xu)(\lambda v. v) \\
& \rightarrow_{\beta} (\lambda v. v)u \\
& \rightarrow_{\beta} u. \\
\text{(c)} & (\lambda x. xx)(\lambda x. xx) \rightarrow_{\beta} (\lambda x. xx)(\lambda x. xx) \\
& \rightarrow_{\beta} \dots
\end{array}$$

Definition 1.7 (β -normal form) A term N containing no β -redexes is called a β -normal form or β -nf. If $M \rightarrow_{\beta} N$, then N is called the β -normal form of M .

Not every term has a β -normal form; in fact Example (c) above shows that

$$(\lambda x. xx)(\lambda x. xx)$$

has none. Examples (a) and (b) above raise the question of whether the normal form is always unique. In fact it is, except for changes of bound variables. This is a consequence of the following general theorem.

Theorem 1.8 (Church-Rosser theorem for β -reduction) (Cf. [18] §1.32.) If $P \rightarrow_{\beta} M$ and $P \rightarrow_{\beta} N$ (see Figure 1), then there exists a λ -term T such that

$$M \rightarrow_{\beta} T, \quad N \rightarrow_{\beta} T.$$

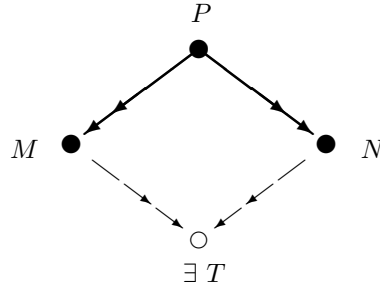


Figure 1:

Definition 1.9 ⁴ A relation \rightarrow is called *confluent* when it satisfies the Church-Rosser theorem, i.e. when (for all P, Q, M) $P \rightarrow M$ and $P \rightarrow N$ implies that there exists T such that

$$M \rightarrow T \text{ and } N \rightarrow T.$$

Note 1.10 (Some combinators) Here are some of the most commonly used combinators, with their standard names and reduction-properties. {In the table, “ X ”, “ Y ”, “ F ”, “ G ” denote arbitrary λ -terms.}

⁴M. SAYS: Confluence is never used so I suggest to erase this definition

I $\equiv \lambda x.x$	(identity combinator):	I $X \rightarrow_{\beta} X$;
B $\equiv \lambda xyz.x(yz)$	(composition):	B $FGX \rightarrow_{\beta} F(GX)$;
C $\equiv \lambda xyz.xzy$	(commutator):	C $FXY \rightarrow_{\beta} FYX$;
K $\equiv \lambda xy.x$	(constant-forming):	K $XY \rightarrow_{\beta} X$;
S $\equiv \lambda xyz.xz(yz)$	(substitution & composition):	S $FGX \rightarrow_{\beta} FX(GX)$;
W $\equiv \lambda xy.xyy$	(doubling):	W $FX \rightarrow_{\beta} FXX$;
Y $\equiv (\lambda ux.x(uxx))(\lambda ux.x(uxx))$	(fixed-point combinator):	Y $F \rightarrow_{\beta} F(\mathbf{Y}F)$;
$\bar{0}$ $\equiv \lambda xy.y$	(to represent zero):	$\bar{0}$ $FX \rightarrow_{\beta} X$;
\bar{n} $\equiv \lambda xy.x^n y$	(to represent number n):	\bar{n} $FX \rightarrow_{\beta} F^n X$,
where $F^n X \equiv \underbrace{F(F(\dots(FX)\dots))}_{n \text{ "F"s}}$		

Note 1.11 (Functions of many variables) A λ -term F represents an operator which accepts an input X and produces an output which we call (FX) . But in mathematics there are functions such as addition that need two inputs before they can produce an output. There are no such functions in λ -calculus.

However, mathematical many-place functions can be represented indirectly by one-input operators if we choose these operators to be higher-order (i.e. operators whose outputs are other operators). For example, let f be a 2-place function which accepts numbers x, y as inputs and produces an output-number $f(x, y)$. Define a corresponding one-input operator f^* as follows. First, to each input-value x there corresponds a one-input operator f_x such that, in standard mathematical notation,

$$f_x(y) = f(x, y) \quad \text{for all } y.$$

Then define f^* by setting

$$f^*(x) = f_x \quad \text{for all } x.$$

This gives, for all x, y ,

$$(f^*(x))(y) = f_x(y) = f(x, y).$$

The act of representing f by f^* is often called *Currying*, after Haskell Curry who first brought it to the notice of computing community. (But Curry did not claim it was his own idea.)

By the way, Curry was a contemporary of Church. He contributed to the study of λ -calculus, but his main work was on developing a rival system, *combinatory logic*, which has no bound variables and whose grammar is thus much simpler; see [18] Chapter 2. It has proved useful because of this, but is not as widely used in computer science as λ -calculus.

Note 1.12 (Computable functions) If the combinators $\bar{0}, \bar{1}, \bar{2}, \dots, \bar{n}, \dots$ are chosen to represent the numbers $0, 1, 2, \dots, n, \dots$, then certain other combinators represent mathematical functions or operators. For example, let

$$\overline{Add} \equiv \lambda uvxy. ux(vxy), \quad \overline{Mult} \equiv \lambda uvx. u(vx);$$

then it can be shown that, for all natural numbers m and n ,

$$\overline{Add} \bar{m} \bar{n} \rightarrow_{\beta} \overline{m+n}, \quad \overline{Mult} \bar{m} \bar{n} \rightarrow_{\beta} \overline{m \times n}.$$

In general, a k -argument function f of natural numbers is said to be *represented* by a λ -term M when

$$M\bar{n}_1 \dots \bar{n}_k \rightarrow_{\beta} f(n_1, \dots, n_k)$$

for all natural numbers n_1, \dots, n_k . It can be proved that every function that is computable (by any of the standard idealised computers in the literature, such as a Turing machine) can be represented by a combinator.

In this sense all computable functions can be programmed in λ -calculus.

2 Types in λ -calculus

Types are expressions that are intended to denote sets, and when a program or function or operator f changes members of a set denoted by σ to members of a set denoted by τ , we may assign to f the type $(\sigma \rightarrow \tau)$. Types were introduced into λ -calculus by Church in [9]; he assigned to every variable a unique type, and constructed composite typed terms thus:

$$\left\{ \begin{array}{l} \text{(i) from typed terms } M^{(\sigma \rightarrow \tau)} \text{ and } N^{\sigma}, \text{ construct } (M^{(\sigma \rightarrow \tau)} N^{\sigma})^{\tau}; \\ \text{(ii) from a variable } x^{\sigma} \text{ and term } M^{\tau}, \text{ construct } (\lambda x^{\sigma}. M^{\tau})^{\sigma \rightarrow \tau}. \end{array} \right\} \quad (1)$$

Systems of λ -calculus in which types are part of a term's construction in this way are called *Church-style*. In a Church-style system, for example, the expression $(\lambda x.x)$ is not a term; but for every type τ , there is a variable v^{τ} from which we can build a term

$$(\lambda v^{\tau}. v^{\tau})^{\tau \rightarrow \tau}.$$

Roughly speaking, for every definable set a Church-style system has an identity-operator on that set, but it does not have a “universal” identity operator.

In contrast, in a *Curry-style* system, the λ -terms are constructed without types, and types are assigned to terms by formal rules. For example, in such a system $(\lambda x.x)$ is a term and it receives an infinite number of types:

$$a \rightarrow a, \quad b \rightarrow b, \quad (a \rightarrow b) \rightarrow (a \rightarrow b), \quad \text{etc.} \quad (2)$$

Also in the Curry style, types may contain parameter-symbols called *type-variables*; if a is a type-variable, then all the types in (2) can be obtained

from the single type $a \rightarrow a$ by substitution; $a \rightarrow a$ is called a *principal type* of $(\lambda x.x)$.

There are many different type-systems based on varieties of λ -calculus. intermediate between these two styles.

To show one type-system in more detail, we shall here describe the simplest Curry-style system. (Cf. system $\text{TA}_{\lambda}^{\rightarrow}$ in [18] Ch. 12 or $\lambda \rightarrow$ -Curry in [5] §3.)

Definition 2.1 (Simple types) Let \mathbf{N} be a symbol to denote the set of natural numbers $\{0, 1, 2, 3, \dots\}$; we shall call \mathbf{N} a *type-constant*.⁵ Let a, b, c, \dots be a sequence of other symbols; we shall call them *type-variables*. Then *types* are expressions built from type-constants and type-variables by this rule:

$$\text{From any types } \sigma, \tau, \text{ build } (\sigma \rightarrow \tau). \quad (3)$$

Examples. The following are types:

$$a, \quad \mathbf{N}, \quad (a \rightarrow b), \quad (a \rightarrow \mathbf{N}), \quad ((\mathbf{N} \rightarrow a) \rightarrow (b \rightarrow \mathbf{N})).$$

Notation. Greek letters ρ, σ, τ will denote arbitrary types. Parentheses may be omitted from types in such a way that, for example,

$$\rho \rightarrow \sigma \rightarrow \tau \equiv (\rho \rightarrow (\sigma \rightarrow \tau))$$

Definition 2.2 (System $\text{TA}_{\lambda}^{\rightarrow}$) A *type-assignment formula* or *TA-formula* is any expression $M : \tau$, where M is a λ -term (as in Definition 1.1) and τ is a type.

$\text{TA}_{\lambda}^{\rightarrow}$ has the following three *rules*. Roughly speaking, each rule means that from the expressions above the line, one may deduce the expression below. Deductions are built as trees, with one formula at the bottom and assumptions at the tops of branches. (See [18] §§12.1–12.6 for details. $\text{TA}_{\lambda}^{\rightarrow}$ is very like Gentzen’s “Natural Deduction” systems in logic, cf. [36].)

$$(\rightarrow \text{e}), \text{ the } \rightarrow\text{-elimination rule: } \frac{M : \sigma \rightarrow \tau \quad N : \sigma}{(MN) : \tau},$$

$$(\rightarrow \text{i}), \text{ the } \rightarrow\text{-introduction rule: } \frac{\begin{array}{c} [x : \sigma] \\ \vdots \\ M : \tau \end{array}}{(\lambda x.M) : (\sigma \rightarrow \tau)},$$

$$(\equiv_{\alpha}), \text{ rule of bound variables: } \frac{M : \tau \quad M \equiv_{\alpha} N}{N : \tau}.$$

⁵M. SAYS: why do you introduce type constants?

Explanation. Rule $(\rightarrow i)$ means that if we already have made a deduction of $M:\tau$ from $x:\sigma$ and perhaps a set Γ of other assumptions not involving x , then we can deduce, from Γ alone, the statement $(\lambda x.M):(\sigma \rightarrow \tau)$. (The rule is not allowed to be used if Γ contains x .) Also, after we use rule $(\rightarrow i)$, we enclose the assumption $x:\sigma$ in brackets wherever it occurs in the deduction-tree above $M:\tau$ to show that it is now no longer regarded as an assumption. It is now called a *cancelled* or *discharged* assumption. (Cf. [18] §12.1.) So a deduction grows in two ways, by adding new conclusions at the bottom, and by adding brackets to assumptions. A deduction with all assumptions discharged is called a *proof*.

Definition 2.3 If x_1, \dots, x_n are distinct variables and there is a deduction of $M:\tau$ with all assumptions cancelled except those in the set $\{x_1:\rho_1, \dots, x_n:\rho_n\}$, we say the formula $M:\tau$ has been *deduced from* $\{x_1:\rho_1, \dots, x_n:\rho_n\}$, or

$$x_1:\rho_1, \dots, x_n:\rho_n \vdash M:\tau.$$

In the special case that $n = 0$, i.e. that there exists a proof of $M:\tau$, we say

$$\vdash M:\tau.$$

Example. Let $\mathbf{S} \equiv \lambda xyz.xz(yz)$ and let ρ, σ, τ be any types. Here is a proof of

$$\mathbf{S} : (\rho \rightarrow (\sigma \rightarrow \tau)) \rightarrow ((\rho \rightarrow \sigma) \rightarrow (\rho \rightarrow \tau)).$$

Note that each of the assumptions is undischarged at the start of the deduction and then becomes discharged later.

$$\frac{\frac{\frac{[x:\rho \rightarrow (\sigma \rightarrow \tau)] \quad [z:\rho]}{xz:\sigma \rightarrow \tau} (\rightarrow e) \quad \frac{[y:\rho \rightarrow \sigma] \quad [z:\rho]}{yz:\sigma} (\rightarrow e)}{xz(yz):\tau} (\rightarrow e)}{\lambda z.xz(yz):\rho \rightarrow \tau} (\rightarrow i) \text{ (discharge both occurrences of “} z:\rho \text{”)} \\ \frac{\lambda yz.xz(yz):(\rho \rightarrow \sigma) \rightarrow (\rho \rightarrow \tau)}{\lambda xyz.xz(yz):(\rho \rightarrow (\sigma \rightarrow \tau)) \rightarrow ((\rho \rightarrow \sigma) \rightarrow (\rho \rightarrow \tau))} (\rightarrow i) \text{ (dis. “} y:\rho \rightarrow \sigma \text{”)} \\ \frac{}{\lambda xyz.xz(yz):(\rho \rightarrow (\sigma \rightarrow \tau)) \rightarrow ((\rho \rightarrow \sigma) \rightarrow (\rho \rightarrow \tau))} (\rightarrow i) \text{ (dis. “} x:\rho \rightarrow (\sigma \rightarrow \tau) \text{”)}$$

TO BE CONTINUED

3 λ -calculus in Computer Science

The λ -calculus can be considered the smallest universal programming language. Quoting Peter Landin [25]: “whatever the next 700 programming languages turn out to be, they will surely be variants and extensions of λ -calculus”.

In this section we will overview some of the more interesting developments of computer science in which the λ -calculus has played a crucial role. The main source of the historical information is [7].

3.1 Computability Theory

As we have seen in Note 1.12 we can represent numbers and computable functions on numbers by λ -terms. We gave also the λ -terms encoding some mathematical operators, but we did not show the encoding of the predecessor, which is quite tricky. When Kleene showed to his teacher Church the λ -term defining the predecessor function, Church said: “But then all intuitively computable functions must be λ -definable” [6]. This intuition of Church can then be considered at the basis of “Church Thesis”:

λ -definability exactly captures the informal notion of effective calculability.

In fact Kleene analyzed the notion of λ -definability showing that every recursive function can be coded (by means of “normal forms”) into the λ -calculus [22, 23]. Church related the notion of effective calculability to that of recursive function, and then of λ -definability, proving at the same time that the equivalence of two λ -terms (not in normal-form) is undecidable [8]. Immediately after the work of Church, Turing introduced his machine approach to computation and proved the undecidability of the halting problem [40]; then, he also proved the equivalence between his notion of computability and that of λ -definable function [41].

3.2 Functional Programming

John McCarthy (Artificial Intelligence Laboratory, Stanford University) proposed at the end of the 50-th the first functional language, *LISP* (LISt Processing) [31]. He wrote: “To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ -notation of Church”. LISP allows to reduce a β -redex (see Definition 1.6)

$$(\lambda x.M)N$$

only if N is a *value*, i.e. either a λ -abstraction or a constant. Usually such a β -redex is called a β_v -redex [35]. Moreover LISP does not allow to reduce under a λ -abstraction (as we did in Example (b) after Definition 1.6) and the substitution captures free variables, i.e. clauses (e)-(f) of Definition 1.4 are replaced by:

$$(e') \quad [N/x](\lambda y.P) \equiv \lambda y.P$$

where the free occurrences of y in N (l.h.s.) are captured by the λ -abstraction in the r.h.s. In other words the λ -calculus uses *static binding*, while LISP uses *dynamic binding*. New LISP dialects (Common Lisp [38], Scheme [2], etc.) use static binding. Lisp today is a family of computer programming languages used in artificial intelligence, Web development, finance, computer science education, and various other applications.

While LISP is an untyped language, *ML* (Meta-Language) is a functional programming language based on type assignment for λ -calculus (Definition 2.2): the programmer can write untyped programs, but the compiler will either infer

types or return an error message [17, 32]. There are several languages in the ML family today: SML (Standard ML) [34], Caml (Categorical Abstract Machine Language) [13], etc. ML's applications include language design and manipulation (compilers, analyzers, theorem provers), bioinformatics, financial systems, a genealogical database, and a peer-to-peer client/server program.

3.3 Evaluation Strategies

In Example (a) and (b) after Definition 1.6 we have seen that the same λ -term can be reduced by choosing different β -redexes. We know from the Church-Rosser Theorem for β -reduction (Theorem 1.8) that this choice does not influence the final result, if any. But it can influence the number of reductions, since for example:

$$\begin{aligned} \underline{(\lambda x.xx)((\lambda y.y)(\lambda z.z))} &\rightarrow_{\beta} \underline{(\lambda y.y)(\lambda z.z)((\lambda y.y)(\lambda z.z))} \\ &\rightarrow_{\beta} \underline{(\lambda z.z)((\lambda y.y)(\lambda z.z))} \\ &\rightarrow_{\beta} \underline{(\lambda y.y)(\lambda z.z)} \\ &\rightarrow_{\beta} (\lambda z.z) \end{aligned}$$

while:

$$\begin{aligned} (\lambda x.xx)(\underline{(\lambda y.y)(\lambda z.z)}) &\rightarrow_{\beta} \underline{(\lambda x.xx)(\lambda z.z)} \\ &\rightarrow_{\beta} \underline{(\lambda z.z)(\lambda z.z)} \\ &\rightarrow_{\beta} (\lambda z.z) \end{aligned}$$

More interesting, the choice of β -redexes can lead either to a terminating or to a diverging computation, as in:

$$\begin{aligned} \underline{(\lambda xy.y)((\lambda z.zz)(\lambda z.zz))} &\rightarrow_{\beta} (\lambda y.y) \\ (\lambda xy.y)(\underline{(\lambda z.zz)(\lambda z.zz)}) &\rightarrow_{\beta} (\lambda xy.y)(\underline{(\lambda z.zz)(\lambda z.zz)}) \\ &\rightarrow_{\beta} (\lambda xy.y)(\underline{(\lambda z.zz)(\lambda z.zz)}) \\ &\rightarrow_{\beta} \dots \end{aligned}$$

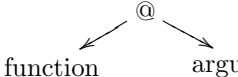
In the first reductions of the above examples we choose always the left-most outer-most β -redex, while in the second reductions we choose always the left-most outer-most β_v -redex.

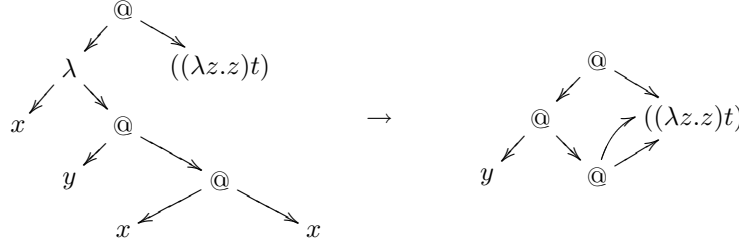
An *evaluation strategy* is a systematic way of indicating which β -redex or β_v -redex to reduce in an arbitrary λ -term. The above exemplified strategies are called respectively *call-by-name* and *call-by-value* [35]. They correspond respectively to the call-by-name and call-by-value parameter passing of programming languages procedures. For example considering the procedure $square(x) = x \times x$ the evaluation of $square(2 + 1)$ when x is a call-by-name parameter gives $(2 + 1) \times (2 + 1)$, while when x is a call-by-value parameter it gives $square(3)$.

An interesting strategy is the *lazy evaluation*, which waits until the last possible moment to evaluate a term, and never reduce under abstractions, increasing in this way the performance. It allows also to deal with infinite data structure: one could create a function that creates the infinite list of Fibonacci numbers. The calculation of the n -th Fibonacci number would be merely the extraction of that element from the infinite list. The entire infinite list is never calculated, but only the values that influence a calculation. For example *Miranda* [43] and *Haskell* [21] are lazy functional programming languages.

A natural question is: “which is the best evaluation strategy?” Unfortunately J. J. Lévy showed that there is no optimal evaluation strategy [27].

The study of evaluation strategies for λ -calculus strongly influenced the implementation of functional programming languages. Various techniques were developed in order to get more efficient implementations, we only mention here *graph reduction*. In graph reduction a λ -term is represented as a directed graph

(inverted tree):  in order to avoid duplicated computation of shared sub-terms [45]. For example:



The efficiency of graph reduction is enhanced when a functional program is translated to a fixed set of functions without free variables (combinators) [42] or when a functional program is translated to a set of functions without free variables and the members of the set are selected to be optimal for that program (super combinators) [20].

Thanks to these techniques and to the computational power of modern computers imperative languages are no longer more efficient than functional languages.

3.4 Semantics

Algol 60 (ALGOrithmic Language) [3] is an imperative language designed by the Working Group 2.1 of IFIP (International Federation for Information Processing). Peter Landin translated the core of Algol 60 into λ -calculus extended with assignment [24]. This translation was the first formal semantics of a real programming language.

Two key problems of programming language semantics are:

- the semantics of looping constructs (**while**, **until**, ... statements) and of recursive procedures requires the solution of recursive equations. For example Euclid's algorithm is a recursive procedure:
 $\text{gcd}(n, m) = \text{if } m = 0 \text{ then } n \text{ else } \text{gcd}(m, n \text{ modulus } m);$
- (higher order) procedures can receive as arguments other procedures, so the meaning of a procedure must be both a function and an argument.

These problems need to be solved already for giving the semantics of λ -calculus since:

- recursive equations can be solved thanks to the fixed-point combinator (see Note 1.10);
- each λ -term can appear either in function or in argument position, so its meaning must be both a function and an argument.

Dana Scott gave the first model of λ -calculus solving the domain equation

$$\mathcal{D} = [\mathcal{D} \rightarrow \mathcal{D}]$$

in the category of complete lattices and continuous functions [37]. Scott's model is the basis of the denotational semantics of programming languages [39].

More recently the λ -calculus of objects was used to give the semantics and to study the types of object-oriented languages [1]. Lastly we mention that a linear and reversible λ -calculus with constants can represent atomic quantum logic gates [44].

3.5 Verification and Extraction of Proofs and Programs

It is easy to check that by erasing λ -terms in the system $\text{TA}_{\lambda}^{\rightarrow}$ of Definition 2.2 we get the intuitionistic implicational logic. This observation is at the basis of the *Curry-Howard isomorphism* [14, 19] which explicit the following correspondences:

types	\Leftrightarrow	logical formulas
terms	\Leftrightarrow	proofs
β -rule	\Leftrightarrow	cut elimination

Certified Programming is centred on Curry-Howard isomorphism. The key idea in fact is that the development of a program satisfying a specification *is* finding a proof of a logical formula. In this way the obtained program comes with its correctness proof. The typed λ -calculus is used as a programming language, a specification language and a programming logic. For example the constructive proof of the sentence: “for all pairs of integers n_1, n_2 there is an integer m such that $m = n_1 + n_2$ ” is the program that computes the sum of two integers. Usually in this way one obtains fairly too large programs, from which it is crucial for efficiency to cancel non computational parts.

In the late sixties N.G. de Bruijn started the *AUTOMATH* (AUTOMated MATHematics) project [15], based on a λ -calculus with *dependent types* (types

which can contain terms). He designed a language for expressing complete mathematical theories in such a way that a computer can verify the proof correctness. The full text book “Grundlagen der Analysis” of E. Landau was verified [33].

In *Martin-Löf’s Constructive Type Theory* [28, 29, 30] the following identifications can be made:

- a is an element of the set τ
- a is a proof of the proposition τ
- a is an object in the type τ
- a is a program with specification τ
- a is a solution to the problem τ .

Martin-Löf developed his type theory (based on a λ -calculus with dependent types) during 1970-1980 as a foundational language for mathematics. He designed a functional programming language including its own logic.

The *PRL* (Proof/Program Refined Logic) Project [10] focuses on implementing computational mathematics and on providing logic-based tools that help automate programming. A proof of a logical formula is compiled into an executable and certified code (essentially a λ -term). *Nuprl* (pronounced “new pearl”, Nuprl means instances - instance Greek “nu” - of PRL) [11] is a family of proof development systems for the incremental verification of software systems properties. Martin-Löf’s type theory strongly influenced the development of Nuprl.

CoC (Calculus of Constructions) [12] integrates dependent types and universal quantification on type variables. It is a higher-order typed λ -calculus where types are first-class values: it allows to define functions from, say, integers to types, types to types as well as functions from integers to integers. *Coc* is the basis of *Coq* [16], a proof assistant which

- handles mathematical assertions,
- checks mechanically proofs of these assertions,
- helps to find formal proofs,
- extracts a certified program from the constructive proof of its formal specification.

Coq is written in the Ocaml (Objective Caml) [26] system, which is the main implementation of the Caml language.

References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
- [2] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1996. 2nd edn.
- [3] J. Backus, F. Bauer, J. Green, C. Katz, J. M. P. Naur, A. Perlis, H. Rutishauser, K. Samuelson, B. V. J. Wegstein, A. van Wijngaarden, and M. Woodger. *Revised Report on the Algorithmic Language Algol 60*. IFIP, 1963.
- [4] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland Co., Amsterdam, 1984. 2nd (revised) edn., reprinted 1997 (1st edn. was 1981).
- [5] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures*, pages 117–309. Clarendon Press, Oxford, England, 1992.
- [6] H. P. Barendregt. The impact of the lambda calculus. *Bulletin of Symbolic Logic*, 3(2):181–215, 1997.
- [7] F. Cardone and J. R. Hindley. History of lambda-calculus and combinatory logic. To appear.
- [8] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58:345–363, 1936.
- [9] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [10] R. L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of IFIP Congress*, pages 229–233, 1971.
- [11] R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, New Jersey, 1986.
- [12] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
- [13] G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
- [14] H. Curry. Some properties of equality and implication in combinatory logic. *Annals of Mathematics*, 35:849–850, 1934.

- [15] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In M. Laudet, D. Lacombe, and M. Schuetzenberger, editors, *Symposium on Automatic Demonstration*, pages 29–61, IRIA, Versailles, 1970. Springer Verlag, Berlin, 1970. Lecture Notes in Mathematics **125**; also in [33].
- [16] G. K. Gérard Huet and C. Paulin-Mohring. *The Coq Proof Assistant A Tutorial*. INRIA, 2006.
- [17] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
- [18] J. R. Hindley and J. P. Seldin. *Lambda-calculus and Combinators, an Introduction*. Cambridge Univ. Press, England, 2008.
- [19] W. Howard. The formulas-as-types notion of construction. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, New York, 1980. Academic Press.
- [20] R. J. M. Hughes. *The design and implementation of programming languages*. PhD thesis, University of Oxford, 1984.
- [21] S. P. Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.
- [22] S. C. Kleene. A Theory of Positive Integers in Formal Logic. *American Journal of Mathematics*, 57:153–173 and 219–244, 1935.
- [23] S. C. Kleene. Lambda-Definability and Recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.
- [24] P. Landin. A correspondence between ALGOL 60 and Church’s lambda notation. *Communications of the ACM*, 8:89–101, 158–165, 1965.
- [25] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.
- [26] X. Leroy. *The Objective Caml system release 3.10*. INRIA, 2007.
- [27] J. J. Lévy. *Réductions correctes et optimales dans le λ -calcul*. PhD thesis, University of Paris 7, 1978.
- [28] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium ’73*, pages 73–118, Amsterdam, 1975. North-Holland. Studies in Logic and the Foundations of Mathematics **80**.
- [29] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.

- [30] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Napoli, 1984.
- [31] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.
- [32] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, aug 1978.
- [33] R. Nederpelt, J. Geuvers, and R. d. Vrijer, editors. *Selected Papers on Automath*. Studies in Logic and the Foundations of Mathematics **133**. North-Holland, Amsterdam, 1994.
- [34] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. 2nd edn.
- [35] G. D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoret. Comput. Sci.*, 1(2):125–159, 1975.
- [36] D. Prawitz. *Natural Deduction*. Almqvist and Wiksell, Stockholm, 1965.
- [37] D. Scott. Continuous lattices. In *Toposes, algebraic geometry and logic (Conf., Dalhousie Univ., Halifax, N. S., 1971)*, pages 97–136. Lecture Notes in Math., Vol. 274. Springer, Berlin, 1972.
- [38] P. Seibel. *Practical Common Lisp*. Apress, 2005.
- [39] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, 1977.
- [40] A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings London Mathematical Society*, 42(2):230–265, 1936. correction *ibid.* 43, pp 544–546 (1937).
- [41] A. Turing. Computability and λ -definability. *Journal of Symbolic Logic*, 2:153–163, 1937.
- [42] D. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.
- [43] D. A. Turner. Miranda—a non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Functional programming languages and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*. Springer Verlag, 1985.
- [44] A. van Tonder. A lambda calculus for quantum computation. *Siam J. on Computing*, 3:1109–1135, 2004.
- [45] C. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. D.Phil thesis, University of Oxford, Programming Research Group, Oxford, U.K., 1971.