Proofs and Programs[1]

Giuseppe Longo
Département d'Informatique
CNRS - Ecole Normale Supérieure
http://www.di.ens.fr/users/longo

1 . Introduction.

In order for machines to do Mathematics what is required first of all is a language that describes Mathematics in adequate terms for machines. This language has to be completely *formalised* and without any semantic ambiguity. Computers cannot make operational choices as a function of the meaning of a phrase, especially if it is uncertain or depends on the context, but only on analysing its syntactic structure. Furthermore, the language has to be sufficiently simple, even if artificial, and comprehensible by the human programmer.

There are two main approaches to making computers do Mathematics. On the one hand, the studies of 'computation' systems, that is the study of languages and techniques that aid algebrists or analysts in 'calculating' what would otherwise be arduous or out of reach for a human being. On the other hand, the development of languages that replace or complement the logical and deductive activity of a mathematician (automatic proof and symbolic calculus). In this exposition, we will present, in a very informal manner, the second approach, with the particular point of view, which is important or at least paradigmatic, of logical and functional computations that are grouped under the name 'λ–calculus'. We will especially see the linguistic aspect of the problem of elaborating a language to perform deductions and manipulate symbols, and more generally to program in a rigorous, modular and translatable manner. Indeed, the ability to conceive advanced languages and reliable programs is the principal objective of the 'relation' we will describe, between machines and mathematical proofs, and it is also the reason for this paper.

Let us observe to begin with, that if today the existence of machines encourages the development of automatic proofs and of symbolic calculi, it can also be stated that the basic ideas for the conception of modern computers is found in the study of proofs, as abstract human activity. The notion of effective computation and languages of which we will speak dates from the twenties and thirties, that is well in advance of numerical computers and formed an integral part of the Hilbertian Proof Theory, that is of the project to formalise mathematical deduction, and answer Hilbert's

[1] **Synthese**, Kluwer, Boston, ~~2002, to appear~~.

fondational problem. The goal was to give rigor or logico-formal foundations to all mathematical activities.

In spite of the failures of the Hilbert's initial project, the ideas of logicians and mathematicians like Turing, Kleene, Church, Gödel, Herbrand... set the bases for Computer Science. Turing Machines, later developed also by Von Neumann, have formed the paradigm of the first computers and of languages known as *imperative programming* (based on orders like 'do', 'go to'...). The formal systems for computing we will talk about had above all an influence on *functional programming* and *logical programming*, which are recent styles of programming and differ from the imperative style. Moreover, they are linked even more directly to the developments of mathematical logic since the '30s.

Mathematical logic, in its 'metamathematical' analysis aspect, has Mathematics themselves as its *object of study*, its languages and its deductive methods, like geometry, to take a purely mathematical discipline as an example, has as its object of study figures and structures of space. One can therefore imagine rather artificial but convenient a three-level stratification, which has organised, with the passing of this century, the 'mathematical discourse': the geometric-algebraic structures, the mathematical theories which study them (algebra, geometry...) and, finally, metatheories that deal with mathematical theories and where one may develop a 'theory of proofs' (one will also try to see the limits of this 'organisation of mathematical discourse'). In other words, from the point of view of computations and languages, linear algebra and analytic geometry, for instance, study the expressions that represent lines on a plane or surfaces in space; λ–calculus, as a language of Proof Theory, manipulates words or expressions that represent formal proofs. In fact, expressions of this language codify abstract mathematical proofs and, therefore computations carried out on them correspond to formal operations on proofs, rather than on the lines or on the surfaces. The fact that λ-calculus is programmable, and that in fact it is a paradigmatic programming language, allows to describe the passage from Proof Theory, as an abstract theory in mathematical logic, to automated proofs and symbolic calculus, as mathematical methods in computing.

Although we will speak further on of Proof Theory, we will underline here the 'constructivist' approach in order to present 'proofs as lambda-terms' and study the computer version of provability. While using λ–calculus, we will mention the role of Category Theory in the mathematical semantics of deduction and of formal programming languages (see below). Indeed, the results that link the different sectors of Mathematics and Computer Science place λ–calculus and Combinatory Logic, an equivalent system, at the meeting point of vast sectors of Logic and their applications, providing these theories with an importance that goes beyond their origins as a system for calculability or effective provability. As we have said, the two theories, which have as a common base the 'algebraic calculus without variables' of Shoenfinkel dating from the early Twenties and that are owed to Church and Curry (1928-1936), essentially proposed to 'formalise'

the notion of computable function or process and effective proof, to give them a mathematical definition and to found Mathematics on the 'unshakeable certainties' of minimal symbolic systems. The equivalence, due to Church, Turing and Kleene, with the other computation systems (recursive functions and Turing machines in particular) provided a complete generalisation for these systems as instruments for computation as early as the Thirties. During this time, it was λ–calculus that played a central role in the proofs of these equivalence theorems: in fact it was proved that calculable functions, in Turing's sense, are exactly those that are definable within λ–calculus and that these, in turn, coincide with partial recursive functions (see Barendregt [1984]). The coincidence of the expressive capacities of these diverse formal systems suggested to Turing and Church a working hypothesis known as *Church's Thesis*: all intuitively calculable functions in a finite manner (a finite number of instructions, a finite number of computation steps...) can be represented in one of the mentioned systems and therefore, thanks to the equivalence, in all of them.

In the same way and till today, λ–calculus played a central role while becoming an important medium, as we will try to establish, in the applications of Proof Theory and Category Theory to Computer Science, and also a language for automatic proof, especially thanks to some of its recent extensions, such as the Calculus of Constructions (see section 8). In general, automatic methods of proof allow deduction of theorems within a logic system, and to synthesise both proofs and theorems. In this paper, we will omit a fundamental aspect: the methods known as methods of *resolution and unification*. The difficult technique of these methods renders their synthetic presentation arduous; and furthermore, a glance over the other aspects of automatic elaboration (symbolic calculus, functional programming and its mathematical semantics), more than the deepening of specific techniques, allows for a better outline of the transfer of certain mechanical tasks from the man to the machine, by treating them from the point of view of different forms of mathematical knowledge.

In sections 2 and 3, we will present 'types as propositions', that is to say that we will study a very simple logical calculus, whose system of proofs is a calculus of terms, the terms of λ–calculus. The goal of this presentation isn't only to give a certain unity to these related topics, but to produce a simple 'semantic', for both logical formulas and types. In fact, on one hand, the logical meaning of types is certainly rich in information, particularly for the reader who is familiar with propositional calculus. On the other hand, an interpretation of propositions and proofs as types and terms (§ 4) can be of great use for the programmer who is accustomed to functional languages but ignores logic. In other words, we will underline that the translation between diverse formalisms is a 'semantic' itself. However, in chapter 5 we will study the most complex point of a true 'mathematical semantic' of a programming language, even if it is in a very limited frame, such as that of λ–calculus. By mathematical semantic, we understand something more than a translation of a language or formal system into another. In short, a formal sign calculus acquires a mathematical meaning when basically different mathematical instruments, geometric or algebraic structures for

example, having an aim, and with independent techniques from the given calculus, provide an interpretation or translation, within these structures, of these terms and formal operations. In general, more of the mathematical structures proposed for interpretation are 'culturally remote' from the formalism in question, more is the provided sense rich in information, because it establishes unexpected bridges, and requires theories that unify. It needs to be said that the general study of the semantics of programming languages rose greatly, specifically thanks to λ–calculus, from the work of D. S. Scott in the Seventies.

Furthermore, this is a concrete experience in computing: an innovative mathematical meaning can suggest extensions or variations in the language in question, inspired by the present constructions in the models and not obtainable within the given formalism.

In chapters 6 and 7 we will study the polymorphism and its semantics, that is, the possibility for a lambda-term or for a functional program to have numerous *types* or to prove proposition schemes. Polymorphism is a paradigmatic form of modularity in programming, directly derived from higher order logic.

Chapter 8 will be devoted to general methods and the limits of automatic proof. Conclusions, in chapter 9, will provide the opportunity for a methodological reflection. The reader that solely wishes to reflect on the 'philosophical' thesis of this article can go directly to chapters 8 and 9. The real motivation of considerations that one finds, resides however in the notions and in the technical results presented in the preceding sections.

2. Natural Deduction and Terms.

The basic idea of natural deduction systems, to those that which we will refer, is the formalisation of the notion of *logical derivation*, understood as the abstraction of mathematical deduction. The minimum deductive step is given by the application of a *rule of inference* that describes the deduction of a consequence, say C, from given premises, for example A_1, A_2, …, A_n:

$$\frac{A_1 \quad A_2 \qquad A_n}{C}.$$

The rules can be composed vertically, that is, given the rules

$$\frac{A \quad B}{D} \qquad \frac{C}{E} \qquad \frac{D \quad E}{C}$$

it is possible to compose them in a deduction (or deductive tree) of E over the hypotheses A, B, C, in the following manner:

$$\dfrac{\dfrac{A \quad B}{D} \qquad \dfrac{C}{E}}{\dfrac{C}{E}}.$$

A ‘tree’ represents a deduction formed from the vertical composition of several rules:

$$\dfrac{\begin{array}{c} A_1 \quad A_2 \qquad A_n \\ \vdots \\ \vdots \\ B \end{array}}{C}.$$

In this case one can use a fundamental notion, that of *erasing*. One erases hypothesis A from which hypothesis B will be derived, if such a derivation is a premise in the deduction of the formula $A \to B$. In fact, the truth of $A \to B$ does not depend on that of A: it may be observed that $A \to B$ is true, even when A is false (‘ex falso quodlibet’). Let us suppose for example to have deduced that if it rains, then the weather is wet. In whatever formal language with an implication ‘$\to$‘ this metalinguistic deduction will have as a formal consequence ‘it rains $\to$ the weather is wet’. However, such an implication is true even if it doesn't rain. One can hence omit or erase the hypothesis ‘it rains’ in the deduction ‘it rains $\to$ the weather is wet’, since the formal implication subsists in any case, independently from the hypothesis, and can be asserted with all truth, even on a sunny day.

The minimum intuitionist system has as formulas, the atomic formulas, A, B, C, ... and the implications between formulas, $(A \to B)$, and no others. This one is based, on terms of natural deduction, only on two inference rules: the *introduction rule*, $(\to I)$, where $[A]$ indicates that A is erased, and the *elimination rule*, $(\to E)$:

Introduction rule **Elimination Rule**

$$(\rightarrow I)\quad \frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A\rightarrow B} \qquad (\rightarrow E)\quad \frac{A \qquad A\rightarrow B}{B}$$

The reader will recognise in $(\rightarrow E)$ a classic 'modus ponens': if A and A *imply* B, then B. In $(\rightarrow I)$, A is erased, in the sense that we mention below, that is to say that it isn't a necessary hypothesis to validate $(A\rightarrow B)$. The rule $(\rightarrow I)$ transfers into the language of formulas the metalinguistic deduction $\frac{A}{B}$. That is, it asserts that from the deduction of B from A, the formula $(A\rightarrow B)$ can be deduced.

A *proof* is a tree made of successive applications of rules of inference. The roots, that are at its base, are the proved theorem. What will interest us more particularly are the metatheorems, that is, the properties of deductive calculus or, more precisely, of calculus of the terms associated to the theorems.

The constructive meaning of this minimal system, based solely on the implication, is given by what is called Heyting-Kleene's interpretation: a proof of $(A\rightarrow B)$ is a procedure of calculus that transforms every proof of A into a proof of B. We will see that the terms found in λ-calculus (λ-terms) formalise this interpretation, they explicitly provide a calculus of proofs. In fact, $c : C$ will mean that the λ-term c is (the code of) an effective proof of the formula C.

Let us build a 'language for proofs'aand its words (or terms). In other words, let us define the λ-terms. In first place, the variables x, y, ... are the terms, and $x : A$ means that x is an arbitrary proof of A and that this one can be used in a hypothesis that can eventually be erased. Suppose then that from an arbitrary proof x of A, that is, $x : A$, a proof b of B can be deduced, $b : B$ (read b proves B). Then, the rule $(\rightarrow I)$ gives $A\rightarrow B$: in our calculus, one will denote $\lambda x : A.b$ the term that proves $A\rightarrow B$, that is $(\lambda x : A.b) : A\rightarrow B$. If on the other hand $c : A\rightarrow B$ and $a : A$, we will write $(c\ a)$ as the term that denotes the application of the proof of c to $A\rightarrow B$ to the proof a of A; this, as we had said, is a proof of B, and thus $ca : B$.

The rules of inference thus define the λ-terms as being *variables*, x, y..., *λ-abstraction* $(\lambda x : A.b)$ of a term b w.r. to an arbitrary variable x, and *applications* (ca) of a term c to a term a. We will

omit the parenthesis when there is no ambiguity. We can now rewrite the rules of introduction and elimination as follows

$$(\to I)\ \frac{\begin{array}{c}[x:A]\\ \vdots \\ b:B\end{array}}{\lambda x:A.b:A\to B} \qquad\qquad (\to E)\ \frac{a:A \qquad c:A\to B}{c\ a:B}.$$

The rules clarify or give name to the transformations that will pass, for example, from a proof $c \equiv \lambda x:A.b$ of $A \to B$ to the proof $(\lambda x:A.b)a$ of B, for $a:A$, thanks to $(\to E)$. We will observe that '$\lambda x:A$' is an *abstraction* operation that *bounds* the variable x in $\lambda x:A.b$, which may occur free in b, that is to say that it can appear without already being bound to b. In fact, $(\lambda x:...)$ corresponds to $\{x \mid ...\}$ in set theory or to the integral $\int ...dx$ in analysis: the meaning or the value of the term, the set or the integral, does not depend on the name of the variable, thus $\{x \mid \mathrm{P}(x)\}$ is equivalent to $\{y \mid \mathrm{P}(y)\}$, $\int f(x)\,dx$ to $\int f(y)\,dy$, as $\lambda x:A.b$ is identical to $\lambda y:A.b'$, provided that b' is obtained from b when *substituting* y for x in the correct manner (we write $b' \equiv [y/x]b$, and we will equally say that x is *renamed* y in b).

We will use $\vdash a:A$ to indicate the provability of $a:A$ in this minimal system; the possible undeleted hypotheses will be placed to the left of '$\vdash$' : for example, $x:A \vdash b:B$. To simplify this, we can omit the type A in the term $(\lambda x:A.b):(A\to B)$, and write $\lambda x.b$. A result, mentioned below, on the possibility to decide the affectation of a proof to a proposition will justify this convention. We observe that the free variables in one term always depend on a hypothesis which is not erased: $\lambda y.yz:(C\to D)\to D$, for example, will be written instead of $(\lambda y:(C\to D).yz):(C\to D)\to D$, under the hypothesis $z:C$ which is not erased. So as to not abuse the λ's, we will abbreviate $\lambda x.\lambda y.\lambda z.(...)$ as $\lambda xyz.(...)$. The interested reader can study and complete the two examples that follow, by observing that those develop the *proofs* of two *axioms* of propositional calculus and, at the same time construct the λ-terms that code the proofs. (Those intended axioms, thus, need not be assumed : the introduction and elimination rules here are strong enough to derive them).

2.1 Examples:

$\vdash \lambda xyz.xz(yz):\ (A\to(B\to C))\ \to\ ((A\to B)\ \to\ (A\to C))$;
$\vdash \lambda xy.x:A\to\ (B\to A)$.

Proof: the rules that are used are indicated to the side of the line of inference.

$$\dfrac{\dfrac{\dfrac{[z:A] \qquad [x:A\to(B\to C)]}{xz:B\to C}(\to E) \qquad \dfrac{[z:A] \qquad [y:A\to B]}{yz:B}(\to E)}{xz(yz):C}(\to E)}{\lambda z.xz(yz):A\to C}(\to I)$$

$$\dfrac{\lambda z.xz(yz):A\to C}{\lambda yz.xz(yz):(A\to B)\to(A\to C)}(\to I)$$

$$\dfrac{\lambda yz.xz(yz):(A\to B)\to(A\to C)}{\lambda xyz.xz(yz):(A\to(B\to C))\to((A\to B)\to(A\to C))}(\to I)$$

We leave the second example, which is simpler, to the reader. ⊗

In the worked out example, the hypotheses are all erased in the last three deductive steps; in particular, the third to last erases two occurrences of the hypothesis $z : A$. Let us note moreover that the structure of the term $\lambda xyz.xz(yz)$ bi-univocally codes the tree of the proof of

$$(A\to(B\to C))\to((A\to B)\to(A\to C)).$$

In fact, in a general manner, the order of the applications and the λ-abstractions corresponds exactly to the order in which the rules $(\to E)$ and $(\to I)$ have just been used.

2.2 Note: 1 (Combinatory Logic). The reader who is experienced in elementary logic will have perceived that the two proven propositions in the example are exactly the two axioms of (positive) propositional calculus, whose formulas do not contain but the implication and which only uses the inference rule 'Modus Ponens', that we called $(\to E)$. Thus, by 2.1, with only the two rules of inference $(\to E)$ and $(\to I)$ and without any axioms, we have the possibility to deduce the axioms of propositional calculus. Now let $S \equiv \lambda xyz.xz(yz)$ and $K \equiv \lambda xy.x$ be the two associated terms as proofs of the two axioms in our inference system. And well, S and K are the two base combinators, or constants that, with the sole application (ab), that is the rule $(\to E)$, constitute Combinatory Logic. Reciprocally, a theorem of propositional calculus, the deduction theorem, restores the rule $(\to I)$, thus proving the logical equivalence between λ-calculus and Combinatory Logic.

2. (Products). The minimal system can easily be widened with conjunctions or product logic. The following rules introduce and eliminate the product, by associating them the terms which in this case are equally formed by couples <... , ...> and the first and second projections, p_1 and p_2, in addition to the applications and λ-abstractions.

$$(\times I)\ \dfrac{a:A \qquad b:B}{<a,b>:A\times B}$$

$$(\times E_1)\ \dfrac{c:A\times B}{p_1(c):A} \qquad\qquad (\times E_2)\ \dfrac{c:A\times B}{p_2(c):B}$$

Observe the constructive meaning of the introduction of the conjunction: the proof of the conjunction of two formulas is constructed from proofs of each component. It needs to be observed that, since there is no negation, the conjunction is not derivable from the implication.

3. Calculus of Proofs and Terms.

A variable in a term can be *instantiated* by another term. For example, with the same notation as the one used to rename variables, we write $[a/z](xzx) \equiv aza$ to instantiate x by a in xzx. In an inductive manner: $[a/x]x \equiv a$; if x is not free in b (i.e. it is not bound by λ) or if it does not appear in b, then $[a/x]b \equiv b$; finally, $[a/x]\lambda y.c \equiv \lambda y.[a/x]c$ and $[a/x](cd) \equiv ([a/x]c)([a/x]d)$. It also needs to be supposed that, in $\lambda y.[a/x]c$ the variable y, does not appear to be free in a, otherwise it would be bound in an improper manner to λ (we will say that a is independent for x in $\lambda y.c$). This condition is not restrictive: it only forces us to rename y, that appears bound in $\lambda y.c$ with a variable that is not independent in a if necessary. Computation will essentially be based on the operation of substitution of a term in the places of occurrence (which might be several) of an independent variable. Lets us see from the beginning the logical meaning of the operation that needs to be formally introduced.

Consider the following proof:

$$(\rightarrow E)\ \frac{\lambda y.b : A \rightarrow B \qquad\qquad a : A}{(\lambda y.b)a : B}$$

Since $x : A$ is an arbitrary or hypothetical proof of A and x can have an occurrence in b, the proof (P.1) is simplified (is reduced) in the following manner:

(P.2)
$$\begin{array}{c} a : A \\ \hline [a/x]b : B. \end{array}$$

The passage from proof P.1 to proof P.2 is known as the *elimination of a cut* and corresponds to the following mathematical reasoning: if I know that from an arbitrary proof x of A I can always deduce a proof b of B, then, in particular, I can deduce from a specific proof a of A, a specific proof, $[a/x]b$ of B. To be precise, the inverse reasoning is particularly pertinent and common in Mathematics: to prove B from a specific proof of A, I may prove at the beginning a general lemma, that assures me that, from each proof x of A, I can deduce a proof b of B, that is to say, I prove that $\lambda x.b : A \rightarrow B$; hence, I obtain the particular case, $[a/x]b : B$, as an instance of a general case, $(\lambda x.b)a : B$. In conclusion, the general proof $(\lambda x.b)a$, given by the lemma $\lambda x.b : A \rightarrow B$, is reduced to the particular proof of $[a/x]b$. We write $c > d$ to say that the term c is reduced to the term d.

3.1 Definition. 1 - *λ-calculus (λβ>) is the calculus of the terms introduced above based on the following axiom of reduction:* $(\lambda x.b)a > [a / x]b$ *where a is independent of x in b (read substitute a in place of all the free occurrences of x in b).*
2 – *λβη> calculus is obtained adding the following axiom:* $\lambda x.cx > c$ *provided x is not independent in c.*

The logical meaning of (η) is not as relevant as that of (β). It can on the other hand be used to point out the functionality of the calculus we are defining. $\lambda x.cx$ is understood as a function that depends explicitly on *x*, whose *body of the definition* is *cx*. Now if *c* does not contain *x*, apply the function $\lambda x.cx$ or *c* directly to a term *a*, of the same type as *x*, it is the same thing: indeed, due to (β), we have $(\lambda x.cx)a > ca$. Then (η) reduces $\lambda x.cx$ to *c*.

The operational meaning of the symbol calculus that we present should be clear. The fundamental axiom (β) is but a mechanical rule for rewriting: the symbol *a* is taken and rewritten or put in place of all the free occurrences of *x* in *b*. It is copied n times if *x* appears n times in *b* (in fact deleted, if $n = 0$). In accordance with observation 2.2.2, it becomes easy to widen the computation of terms by means of logical conjunction: it simply needs to be described formally that the first and second projections, p_1 and p_2 choose the first or second element of each couple and that the couples are always obtained by coupling the first and second projections. The calculus is completed by a rule that expresses the transitive character of the reduction and by the rules that thus guarantee the possibility of the application of the axioms in the sub-terms (substitutability). Summarising what is wanted is the following: if $c > d$, then $\lambda x.a(cb) > \lambda x.a(db)$, in any context $\lambda x.a(-b)$.

3.2 Note (Theories of equality). An immediate extension of the theory of reduction that we just presented is given by considering the equality between formulas as the minimum congruence indicated by '>'. That is to say, $a > b$ implies $a = b$ and '=' is the minimal relation, thus created, that is reflexive, symmetric, transitive and substitutive. λβη= is the theory of equality.

The fundamental notion is then the notion of *normal form* for the λ-terms. A term *is in normal form* if it does not contain sub-terms of the form $(\lambda x.b)a$, i.e. sub-terms to which the axiom (β) can be applied. It *has a normal form* if it reduces to a term in normal form. We then have the typed calculus for:

3.3 Theorem (Normalisation). *Each term of λβ> has a normal form.*

From the standpoint of natural deduction, a proof is a normal form when it does not contain the application of the rule $(\rightarrow I)$ followed by the application of the rule $(\rightarrow E)$, two rules that, when followed by each other, introduce $(\lambda x.b)$ and $(\lambda x.b)a$ respectively. Therefore thanks to the bi-

univocal correspondence between terms and proofs, that was established by construction, the proof of 3.3, carried out within the calculus of terms, provides a logical result for the intuitionist Proof Theory: every proof can be reduced to another, without cuts, of the same assertion.

It needs to be observed that a term can contain several sub-terms to which (β) can be applied, thus different chains of reduction can arise. However, the following theorem guarantees that they are always confluent.

Theorem (Church-Rosser). *If a term a of λβ> is reduced to b and c, then there exists d to which b as well as c are reduced.*

The two proofs require a few specifications (see Hindley & Seldin [1986]). They can easily be extended, however, to computations with (η) and the projections for the product (see 2.2.2), since these reductions do not interfere with (β).

Therefore, because of the perfect correspondence between terms and proofs, each proof, in the corresponding deductive systems, has a canonical form, without cuts, by 3.3, and is itself unique in view of 3.4. If in fact a term *a* is reduced to two normal forms *b* and *c*, those cannot be reduced again, in particular don't have terms in common, they can't but be identical. This fact also guarantees the non contradiction or *consistency* of the equation systems presented, where, in absence of negation, by consistency we understand that not all equations are provable. In fact, 3.4 is equivalent to the following statement: $b = c$ implies that there exists *d* such that $b > d$ and $c > d$. It is therefore not possible to deduce the equality of two distinct normal forms, say *b* and *c*, since, once again, they cannot be reduced to a common *d*.

In conclusion, this constructive approach to Proof Theory suggests a language where the terms code the proofs. The language can be written by a machine, implemented and manipulated automatically, it suffices that the automaton applies axioms (β) and (η). We have thus presented at the same time a language for mathematical proofs and a programming language where the programs are λ-terms. On one side, in fact we are able to manipulate proofs as terms of λ-calculus; of synthesising them from propositions and, inversely, to write the proven proposition from a term; on the other, to carry out purely symbolic computations of terms without logical meaning. In the next section we will examine these facts from the viewpoint of programming.

4. Formulas as Types; Calculus without Types.

λ-calculus has acquired an important role in programming, especially thanks to the programming language LISP (List Processing), which is very common in artificial intelligence and the language ML (Meta Language) and its derivatives, see Mitchell [1993]. In fact, λ-calculus became a paradigm

for all languages that are referred to as functional or application languages, often obtained from this calculus, solely thanks to extensions with constructions that make them more efficient from the standpoint of programming. Functional languages are based on the writing of programs as functions to be applied to arguments ($\lambda x.b$ that is functionally applied to a and reduced to $[a/x]b$) instead of sets of orders (the 'do's', 'go to's' of imperative language programming). The manipulation is purely symbolical, it does not deal with meaning, *a priori*: as we have explained, the base axioms formalise the very simple operations of deletion and copy of symbols.

From the programming point of view, the λ-terms are programmed and the propositions (A, B, $(A \to B)$...), that we have considered as the proven formulas by the λ-terms are called *types* of programs, taking in this a concept developed by Russell. Intuitively, a type is a set of terms; in the terminology of Physics, this can be understood as the 'dimension' of an expression: in $f = ma$, in Mechanics, the expressions that have two members have the 'type' of a force. Obviously not all formulas are propositions: only the formulas that have a proof are. Consider, say, $A \to B$ and $A \to (B \to A)$: only the second formula is provable and the proof is coded as $K \equiv \lambda xy.x$. We will thus call *inhabited* a type that is a proposition, that is to say, that, as a formula, it has a proof coded by terms without free variables. According to what we saw, the type of a term, without a free variable or that has an explicitly typed variable is unique, while a type can contain several terms even if they are all in normal form (a theorem can have many proofs). This corresponds to practical intuition, that a type of program can contain numerous programs: if, for instance, Int is the type (the set) of integers, $\text{Int} \to \text{Int}$ contains all the programs from the integers with an integer value. The types of computations presented until now are called *simple* and, as we have seen, correspond to the formulas (and propositions) of a positive propositional calculus.

However it still needs to be mentioned that λ-calculus as well as LISP, as programming languages, were conceived without type. In fact, if we only consider the terms that admit types, the definable mathematical functions are far from numerous: it is due to this that we talk of the extensions of λ-calculus with variable types in § 5. For us who started with the correspondence between λ-calculus and logic, it is possible to come back to a calculus without types, simply rereading the rules of good formation of terms without any information or restriction of type. Then, the application (ab) is authorised for each a and b, $a = b$ inclusive, without the restrictions imposed by the hypotheses in the rule ($\to E$). Evidently, we cannot attribute a type to all the terms: precisely, xx is a term of the *type-free (or untyped) calculus*, while it isn't in the calculus with types, since in the rule ($\to E$)'s hypothesis it cannot be that x has a type A and, at the same time, a type $A \to B$, as should be necessary to apply x to itself. The reduction axioms (β) and (η) are identical. However, for the calculus without types, the normalisation theorem (3.3) is not valid. It can be seen immediately that for example the term $o = (\lambda x.xx)(\lambda x.xx)$ does not have normal form: it reduces to itself indefinitely. It is worthwhile to note the analogy between $\lambda x.xx$ and the non founded set $\{x \mid x \in x\}$, which is complementary to the paradoxical set, that suggested to Russell the paradox for

Frege's system and the introduction of a Theory Types for set theory: it suffices to substitute the set abstraction $\{x \mid ..\}$ with the abstraction $\lambda x...$ and the self-belonging $x \in x$ with the self-application xx. The non convergence of $o = (\lambda x.xx)(\lambda x.xx)$, a 'negative' fact, if you wish, is in reality linked to the whole expressiveness of computations in a calculus without types. In fact, a variant of o is of great interest. Consider $\theta \equiv \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$ (like in LISP, the parentheses are very important!). Then, in terms of equality and of applying three times axiom (β), one obtains

$\theta a = [a/y](\lambda x.y(xx))(\lambda x.y(xx)) \equiv (\lambda x.a(xx))(\lambda x.a(xx))$

$= a(\lambda x.a(xx)(\lambda x.a(xx))) = a(\theta a)$

This result is very important, as it ensures that for each term a we can find a *fixed point* θa, i.e. an term such that $\theta a = a(\theta a)$. Moreover, the fixed point is provided in a *uniform* and *effective* manner, in other words, inside the language, thanks to the term θ. From this we can deduce the representability, in λ-calculus without types, of all the partial recursive functions, that we mentioned in the introduction. These precisely are defined by recursive equations, where the equations with a fixed point are a generalised version: type-free λ-calculus computes all of them, by solving the defining equations in a uniform and effective manner.

However, having lost all relation to logic (the terms do not necessarily code the proofs, given that they may be type-free) the problem that arises is that of consistency of reduction or equation theory, they have been defined exactly in the same way as experienced in the calculus with types, thus leaving aside all restrictions of type in the formulation of the axioms (β) and (η). Again, in absence of negation, consistency is expressed *in terms of non provability*... of all the equations between the terms. The Church-Rosser theorem (see 3.4), which subsists as well for the λβη calculus without types, guarantees the fact that the equations are not all deducible: like in the case with types, thus it is not possible to deduce the equality of distinct normal forms.

As we have already said, terms in normal form are quite important. The normalisation theorem is the fundamental application of calculus with types in logic, especially important in the case, that we will mention in chapter 6, of higher order logic. In the calculus without types, the terms with a normal form represent the computations that end; certain authors, and from the start Church and Böhm considered endowed with meaning only these terms (we shall return to the notion of 'meaning', not only in computations but when speaking of models). In that case, because of a result due to Böhm [1968] (see also Barendregt [1984]) it is not possible to make equal two different computations that end. More precisely, if: a and b possess different normal forms, $\lambda\beta + (a = b)$ is not consistent. Böhm's theorem also ensures that no calculus of symbols, that is an extension of λ-calculus, can be ambiguous on the computations that end: if they can be expressed in the λ-calculus, we cannot confuse them with each other. From the semantic point of view, Böhm's theorem is a result of "relative compliteness", relative to normal forms: once that an arbitrary model of calculus without types is fixed (see § 5), an equality between normal forms is true if and only if it can be proven.

Another result, that is extremely interesting for programming, that links non typed calculus with typed calculus, is the following: one can decide if a term with type is well typed, and also if a non typed term can be assigned a type (Hindley-Milner algorithm, see Hindley & Seldin [1986]). Said in other words, given a functional program, or a freely written term, without paying attention to types, an automatic type-checker can determine if the program is well typed or if it can admit types. Remember the analogy we have mentioned, between the notion of type in programming and dimension in Physics: the type checking algorithm for the functional programs can be compared, due to its nature and practical side, to the dimensional control of equations in Physics. It is known that general algorithms for the control of the exactitude of programs are not possible (Rice's theorem), that is, it is proven in general that it is not possible to effectively control whether a program calculates the function that it is going to implement. Types then provide an effective tool to partially control the correction of programs, completely analogous to 'dimensional control' in Physics: given an equation in Physics, one calculates, one develops, and at the end one verifies that to the left one finds a force (an energy ...), then to the right one also has to find a force (an energy ...), if the computation is correct. In this case it is also about partial control: in no case does the dimension control ensure the exactitude of the computations made. The same is true for the control of types in a program. However, almost all computation errors in an equation in Physics, or in the implementation of a program, are revealed by a dimensional errors or a type errors. The type checking algorithm is in fact the heart of programming languages of the kind ML: in fact it is divided into a control for the 'typability' and a type assignement algorithm, based on the logical inference rule in § 1.

Let us summarize finally the relation between terms and types from the view point of logic. In short, the affectation of a type to a program is the proof of a proposition, that is, its type. The association of a term to a type is the synthesis of a proof, that coded by the term.

5 . Semantics.

Formalisation of the types and terms presented up to now already has an interpretation: types as propositions, terms as proofs (or vice versa). Let us reflect now on the possibility of a mathematical meaning, non formal or by a purely sign calculus, for the introduced rules and terms. It is desirable to make room for this aspect, apparently not important for the mechanic elaboration, for at least two reasons. Abstract logic formalisms can be adapted to machines that elaborate without 'giving meaning', but that are often hostile to human intelligence. Comprehension of a logic system, whether it be essential or minimal in the formal parts, improves if it is immersed in mathematical structures, not necessarily constructive nor elementary, but based on known experiences of conceptual synthesis or non formal intuition of space-time. Lastly, the role played by λ-calculus in

Computer Science, as a symbolic manipulator and language that describes mathematical functions, is also due to the study of semantics of programming languages that it itself inspired.

We will remember that the two inference rules of λ-calculus, $(\rightarrow I)$ and $(\rightarrow E)$, have very precise roles. The first one "introduced" the metalinguistic deduction from A to B, into the language as a formal implication '$A \rightarrow B$, as well as the terms that code it. This passage is essential for a formal/linguistic treatment of logic as metamathematics: its object of study is mathematical proof and it may give a rigorous linguistic form to deduction in Mathematics, which is often informal, always metalinguistic, of an assertion in a specific language or mathematical theory (the language or theory of groups, of topological spaces...). The other rule, $(\rightarrow E)$, codes with the terms of λ-calculus the classic 'modus ponens' underlining its functional character, as already described by Heyting-Kleene's interpretation. That is to say that the intuitive meaning of $A \rightarrow B$ is that of being a set of effective functions or procedures that transform the elements (proofs) of A into elements (proofs) of B.

In order to give a rigorous mathematical meaning to this intuitive meaning of syntax, we recall the mathematical definition of *category* as a collection of objects, A, B,... and morphisms between objects, f, g,... Morphisms include the identity id_A for each object A and are closed by composition, $f \circ g$; associativeness $(f \circ g) \circ h = f \circ (g \circ h)$ and the identity properties $f \circ \mathrm{id} = f$ and $\mathrm{id} \circ g = g$, complete the definition (see Asperti & Longo [1991]). The category of sets (without structure) with classic functions between sets like morphisms, the category of groups with homomorphisms between groups like morphisms and that of topological spaces, with continuous functions as morphisms, are the common examples of categories. In fact, a category is often a collection of 'structured sets' where the structural properties are described by sets, which are not necessarily structured, of morphisms between each pair of objects. The reader, even if inexperienced can understand in an intuitive manner, that the notion, be it explicit or implicit, of category is fundamental in Mathematics.

We understand thus our formal symbols and logical computations by interpreting types as objects and terms as appropriate category morphisms. However, in general the space of morphisms between two objects of a category is a collection or a set 'outside' the category, in other words, as we have said, it is not necessarily structured as the objects of the category in question, exactly like the deduction of mathematical and metalinguistic practice is outside the theory or the mathematical language object of this study. The necessity to correlate the two notions is clearly suggested by the Heyting-Kleene interpretation of the type $A \rightarrow B$ as a collection of morphisms of A in B, see § 2. Then, to give a mathematical meaning to the rule $(\rightarrow I)$, that brings metalinguistic deduction inside the language, it should be necessary to find categories in which the notion of collection of morphisms between two objects can be internalised, that is, that it can be seen as an object of the given category. In other words, if A and B are objects of the category C, it will also be needed that

$C[A,B]$, the set of morphisms of A in B, be it (represented by) an object of the same category, the *exponent* of A in B, that we define by B^A or $A \to B$. In the case of the category of sets (without structure), it is clear that the notion of morphisms space is immediately internalised: the set of functions between two sets is a set, that is, an object of the category. This is not the case of the other two examples in which the objects are sets with structure: in general, homomorphisms between two groups do not form a group. When dealing with topological spaces, even if the set of continuous functions between two topological spaces can be given a topological structure, it is not always the case that it itself has the necessary property to define exponents in a sufficiently expressive manner so as to interpret the types as objects and the terms of the λ-calculus as morphisms. One observes in the first place that a λ-abstraction allows the formation of a function of more arguments, by 'an argument at a time': given a term $a : A$, that may contain two free variables $x : B$ and $y : C$, the term $\lambda x : C.(\lambda y : B.a) : (C \to (B \to A))$ has the meaning of a function that on taking an argument in C gives as a result a function $\lambda y : B.a$ in $(B \to A)$. But the two free variables in a, equally give $a : A$ the meaning of a function of two arguments: $(\lambda < x, y >: C \times B.a) : (C \times B \to A)$, provided that we have some notion of product in the category of meanings. But this is easy: the Cartesian product of two sets is a set and the same holds as for groups and topological spaces. The categorical generalisation of the idea of Cartesian product as a (structured) set of couples of two (structured) sets is simple and we send the reader to the existing literature or cited text for details. The difficulty lays precisely in the following fundamental operation of λ-calculus called 'currying' (due to H. B. Curry, see Hindley & Seldin [1980]): a function of several arguments can be defined in an equivalent manner by the abstraction of an argument each time. For example, it would be necessary, so that topological spaces provide an interpretation, that a continuous function be considered such, knowing only that it is so w.r. to each argument; it is known on the contrary that, in topology, there exist functions of several variables, continuous in relation to each argument, but not globally continuous, that is not continuous in the topology of the product space. The reader that is familiar with elementary continuous functions on product spaces and that knows that continuity cannot be proved variable by variable, has understood the real mathematical meaning of λ-abstraction and its expressive power: if endowed with great inventiveness or mathematical experience, he/she can construct the class of categories that can provide a rigourous semantic to this peculiar phenomenon. Let us now present them explicitly, for convenience of the readers.

The property required by a category to interpret λ-calculus is to be Cartesian Closed, that is to have all products $C \times B$ and an isomorphism (uniformly internal to the category or *natural*, see Asperti & Longo [1991]) between $(C \to (B \to A))$ and $(C \times B \to A)$, for all C, B and A, objects of the category. For different reasons, as we have said, groups and topological spaces do not have this property. Again, the category of sets comes to our rescue: this isomorphism is trivial between sets.

However, in the semantics of λ-calculus it is necessary to go beyond the simple category of unstructured sets. From the start, as λ-calculus is also a paradigm for functional programming and, if we want to write sufficiently expressive programs with a calculus with simple types, it is necessary to understand it to have the possibility to give recursive definitions of functions. Those, as we have said in § 4, are the ones that are definable in the calculus without types (and thus in the programming language LISP and its dialects). They must be added in the case of languages with types like the language ML. In fact we remember that a recursive definition of a term or a function *f* is given when this one is the solution of an equation $x = a(x)$, or well when *f* is a fixed point of *a*, § 4. For example, the factorial function fact can be defined as follows, by an equation:

$$\underline{\text{fact}}\ (n) = \text{if } n = 0 \text{ then } 1, \text{ otherwise } n \cdot \underline{\text{fact}}(n - 1).$$

Now, while supposing to have coded our metalanguage in λ-calculus, that is our (if ... then …) and in allowing some abuse of the language, the term $a = \lambda xy.$ (if y = 0 then 1, otherwise $y \cdot x(y-1)$) has as a fixed point the factorial since $\underline{\text{fact}} = a\underline{\text{fact}}$. According to what we observed in § 4 on the calculus without types, by taking $\underline{\text{fact}} = \theta a$ one mechanically obtains the solution. Moreover, if one finds a mathematical semantic of *typed* calculus in which there exists an operator, a functional, with the properties of θ, this would justify and also guarantee the logical consistency of the extension of this calculus with a term having θ's property. We are interested, in second place, in a more structured category than that of sets because we want to find in it a model of the calculus without types. In the end this isn't 'but a particular case' of calculus with types: it is the same calculus without the restrictions of types or, if one wants, it is a calculus with a single 'universal' type. It should be necessary, to give meaning, to find a structure with a 'universal' type in the sense that it must contain all the functions on the proper elements; then, each term could be applied to every other term, and in particular to itself. However, no set can 'contain' the set of functions defined on it, except a set composed of a single element or the *trivial* set. In fact, a classic Cantor result ensures that the set of functions on a non trivial set is strictly larger, in terms of cardinality, than the given set. We will find, and it is not easy, a non trivial topological space in which one can isomorphically immerse the space of endomorphisms (morphisms of an object on itself).

With this double goal in mind, the recursion for terms with types and a model for the calculus without types, we will construct a subcategory of the category of topological spaces that are Cartesian Closed and that have fixed points for every endomorphism. Furthermore, it will contain a universal object, in which, in particular, its own endomorphisms can be immersed. The construction demands certain mathematical attention.

Let us take a partially ordered set $(A, \leq)$, a subset *D* of *A* is called *directed* if every pair of elements of *D* admits an upper bound in *D* (that is: $\forall x, y \in D$ (for all *x* and *y* in *D*) $\exists z \in D$ (there

exists z in D) $x \leq z$ & $y \leq z$). Now let $\underline{A} = (A, A0, \leq)$ be a partially ordered set and $A0$ a subset of A; A is a Scott space (*S-space*) if the following conditions are satisfied.

1. every directed set D admits a least upper bound, sup D, in A;
2. A has a least element, lets say ∇;
3. for every x and y in A, if $x/\leq y$ (x is not inferior or equal to y), there exists $z0$ in $A0$ such that $z0 \leq x$ and $z0/\leq y$ ($A0$ separates elements in A);
4. for every $x0$ and $y0$ in $A0$, if $x0$ and $y0$ have an upper bound in A, then they have a least upper bound $z0$ = sup $\{x0, y0\}$ in $A0$.

The reader that knows some Geometry can observe that each S-space $\underline{A}$ can be endowed a topological structure, given by the order, that has as base elements $\{z \in A: x0 \leq z\}$ for $x0$ in $A0$, and the empty set. In such a topology, the continuous functions between two S-spaces are all monotone (non decreasing), that is, they preserve order and, in particular, when applied to a directed set the result is a directed set. Moreover, if f is continuous and D is directed the f (sup D) = sup f (D).

An interesting example of an S-space is constituted by the set of subsets, PB, of an arbitrary infinite set B. It suffices to take the inclusion between sets as a partial order, the collection of finite sets as subsets $PB0$ of PB, so that $PB = (PB, PB0, \subseteq)$ satisfies (1-4), with the empty set as least element.

One can now verify that the category of S-spaces is Cartesian Closed. Thus, a function with more than one argument is continuous if it is continuous in each argument. The Cartesian closure guarantees the possibility to interpret types as S-spaces. The interpretation of λ-terms as morphisms is an easy induction over the structure of the terms themselves: each variable of type A is a morphism of the trivial space $\{o\}$, with a single element, in the interpretation of A; the abstraction $\lambda x : A.b : A \rightarrow B$ defines a morphism of the interpretation of A into that of B; the formal application cd, for $c : A \rightarrow B$ and $d : A$, is the functional application of c to d.

The other property of the construction that interests us is that the objects of the category are topological spaces that also are complete partial orders which is exactly what is required in the hypothesis (1). It is now possible to use a construction due to Knaster and Tarski to construct minimum fixed points of monotone functions. If f is any continuous function, and thus monotone, of an S-space $\underline{A}$ into $\underline{A}$, then the chain $\nabla \leq f(\nabla) \leq f^2(\nabla) = f(f(\nabla)) \leq ...$, as a directed set, admits a least upper bound, sup $f^n(\nabla)$. Indeed, as f is continuous, f (sup$n$ $f^n(\nabla)$) = sup$n$ f ($f^n(\nabla)$) = sup$n$ $f^n(\nabla)$; furthermore, sup$n$ $f^n(\nabla)$ is the minimum fixed point of f. Thus, the functional Θ (f) = sup$n$ $f^n(\nabla)$, that associates to each endomorphism f of the category its minimum fixed point, provides an interpretation for a recursion operator θ of λ-calculus with types.

In conclusion, we have constructed a mathematical model of the calculus with types and operators of fixed points. The instruments used are taken from elementary Geometry and are all independent of λ-calculus.

It is now necessary for us to find, in the category of S-spaces, a model for the calculus without types, in which the terms can be at the same time functions and arguments of functions. To be able to do this it is necessary to construct an object that contains or into which the spaces of its endomorphisms (functions are elements) can be immersed and such that each element defines an endomorphism (elements are functions). Recall that the power set of an infinite set is an S-space and consider the familiar power set P**N**, of the set **N** of natural numbers. The proof that the order and topological structure of P**N**, as an S-space, has the desired qualities is rather technical and uses the property of integer numbers; in particular, the possibility to code pairs and finite sets with numbers (see Scott [1976]). Thus, every function is an element of P**N** and, conversely, every $a \in$ P**N** can be applied, as a function, to each element $b \in$ P**N**: the application *ab* between arbitrary elements of P**N** gives a meaning to the formal application between arbitrary terms, without type restriction, as it is defined in the type-free calculus.

The above mentioned construction completes the semantics of λ-calculus with and without types, while giving a mathematical meaning to abstract symbol manipulations, such that the internalisation of the metalinguistic application, λ-abstraction, the auto-applicability of a term to itself. The autonomy of the used topological structure from the syntax has been underlined not only for epistemological reasons, linked to the notion of meaning as translation, all the more filled with information than the possibility to correlate the different universes, but also for practical reasons. As we said in the introduction, the issue that we study is not only a particular case of mathematical research in Computer Science; in fact, the semantics of λ-calculus has had a paradigmatic role in the manner in which research activity is carried out in the semantics of programming languages. In certain cases, the meaning of geometric or algebraic structures has suggested variants or extensions to programming languages (the ML dialect CAML, the extensions of the prototype language Quest ... and much more). Sometimes, the obscure programming constructions, which are barely clear for the author himself, become intelligible, and are improved if needed. The effort to immerse languages and programs in solid mathematical models has certainly (and at least) contributed to an important improvement of the presentation style of many of them. It is certain that in recent years some programming manuals have become readable, or almost readable, thanks to the increasing influence of a mathematical style that encourages, at the same time, rigor generality and search for meaning.

6 . Polymorphism

At the end of § 4 we said that given a term without types, is 'decidable' if it can be assigned a type. This type however, is not necessarily unique: the identity $\lambda x.x$, for example has type $A \to A$ *for all types A*. That is, it has a *scheme* of type, denoted usually by metavariables of type: $X \to X$. The Hindely-Milner type assignement algorithm (§ 4), implemented in ML, gives to each term, if it has it, the most general scheme for which all other scheme or type of the term in question is a particular instance. For example, $\lambda x.x$ also has the scheme $(Y \to Y) \to (Y \to Y)$, a particular of $X \to X$. The notion of type scheme is completely analogous to that of axiom schemes in logic. To return to the examples given in 2.1, $\lambda xy.x$ has a type scheme (the most general) $X \to (Y \to X)$ and this is one of the two axiom schemes of positive propositional calculus of which we talked about in 2.2.1

Thus, briefly, terms without types, when they can be given a type, are polymorphic, because they have type schemes and thus usually more types, contrary to what we saw in § 2 and 3 when dealing with terms with types. The languages of the class ML are polymorphic exactly in this sense. From the point of view of logic, the programs are the proofs of schemes of propositions.

In the implicit polymorphism, ML style, quantification by relation of variables of type is metalinguistic and only external to type schemes. Recall now the elementary or intuitive meaning of types as sets or, more formally, as objects of a category. *Explicit* quantification, within the language, on variables of sets or objects of a category is on the contrary at the base of second order systems, where sets or objects of categories are quantified. In particular, it is at the base of Analysis, interpreted as a second order arithmetic, since real numbers are sets of integer numbers. Second order λ-calculus, $\lambda\beta\eta 2$ (the *F* system of Girard in 1971, see Girard [1989]), is obtained when adding quantification by relation to the variables of type. Before speaking of explicit polymorphism we observe that the polymorphism of a program can be seen as an *invariance* property by relation to types as structures. That is, the program $\lambda x.x$, that calculates the identity, or $\lambda xy.x$, which calculates a constant function in the first argument, are invariant by relation to each domain of arguments.

The reading of the end of this paragraph (and its semantics in § 7) requires certain attention, although, formally it only supposes instruments already introduced. Higher order logics are based in fact on an ulterior mathematical abstraction.

The types of $\lambda\beta\eta 2$ are obtained while extending those of $\lambda\beta\eta=$ with variables of type, X, Y, … and with universally quantified types, $\forall X : Tp.A$ where Tp is the collection of types (read for all X, A is valid, where it can appear X in A); the terms are also constructed from abstraction with relation to type variables, $\lambda X : Tp.a$, and the application of terms to types, bA. The rules that are introduced by the new types and terms are the following:

$$(\forall I)\quad \frac{\begin{array}{c}[X:TP]\\ \vdots\\ b:B\end{array}}{(\lambda X:Tp.b):(\forall X:Tp.B)}\quad (*)$$

$$(\forall E)\quad \frac{b:(\forall X:Tp.B)\qquad A:Tp}{(bA):[A/X]B}$$

(*) in b no free variables has the type which depends on X.

The first rule forms functions from the collection of types to terms. The second states that a term b can be (functionally) applied to a type A and give a term bA within type B where it substitutes A in X.

The axioms that need to be added to (β) and (η) in § 3 are the following (observe that they are the second order version):

$(\forall\beta)$	$(\lambda X:Tp.b)A=[A/X]b$	with A free of X in b
$(\forall\eta)$	$(\lambda X:Tp.aX)=a:$	with X not free in a.

For example, $(\lambda X:Tp.\lambda x:X.x):(\forall X.X\to X)$ is the second order identity or explicitly polymorph; $\lambda X:Tp.(\lambda Y:Tp.(\lambda x:X.\lambda y:Y.x))$of type $\forall X\forall Y.X\to(Y\to X)$ is the function that is explicitly polymorph and constant in the second argument. Applying the first to a type A one obtains $(\lambda X:Tp.\lambda x:X.x)A=\lambda x:A.x$, the identity of type $A\to A$. In an analogous fashion for $(\lambda X:Tp.(\lambda Y:Tp.(\lambda x:X.\lambda y:Y.x)))AB:(A\to(B\to A))$.

The types inhabited are exactly the theorems of what we call second order propositional calculus, which is the subjacent logic system of Analysis, as second order Arithmetic (real numbers are sets of integer numbers, this is why we need the quantification over sets). In addition, one can understand the Computer Science side shown in this passing to a second order: types are 'automatically updated' since terms can take types as arguments. That is to say, types are dealt with within the language or manipulated by a formal calculus feasible for a machine, instead of being handled outside the language in a metalinguistic manner.

Observe that the theory of types of $\lambda\beta\eta 2$ is essentially *impredicative* (or non predicative). In other words, while stating that $\forall X:Tp.A$ is a type, formally $(\forall X:Tp.A):Tp$, one defines an element of the collection of types, Tp, and this is done by quantifying on the collection Tp itself (note that the defined type $\forall X:Tp.A$ contains the quantification $\forall X:Tp$.) Such definitions are commonplace in Analysis, or Topology: for instance, when one defines a set as the intersection of a

collection of sets that may include the set that is being defined (least upper or greatest lower bounds, Lebesgue measure, …). The impredicativeness of types is at the base of the expressiveness of the language and constitutes a non negligible logical or semantical challenge. However, even for $\lambda\beta\eta 2$, the normalisation and Church-Rosser theorems are valid, stated as done so in 3.3 and 3.4; the proof, of the first in particular, is rather complex by the impossibility of stratifying the formulas and using any form of induction, due to the implicit circularity in the impredicative definition of types (Girard [1989], see Hindley & Seldin [1986). By means of the analogy between types and propositions (belonging to the second order, now), the normalisation and Church-Rosser theorems prove the results of 'elimination of cuts' and 'uniqueness of canonical proofs (or normal form)' for second order systems, like we observed for propositional calculus in § 3. Moreover, these results guarantee the logical consistency of calculus of types and equation calculus, confirming the robustness of impredicative constructions that are at the base of Analysis as second order Arithmetic. The relation with this last theory is shown by the theorem that characterises the expressiveness of calculus. Because of the normalisation theorem it may be shown that the representable functions are all total (always convergent). In fact, the computable functions in $\lambda\beta\eta 2$ are exactly the recursive functions that can be proved total in second order Arithmetic (Girard Lafont Taylor [1986]). Such a set of functions is much larger than that of recursive primitive functions and it largely includes all total functions which one can need for practical computing. However, current experimental programming languages based on explicit polymorphism extend $\lambda\beta\eta 2$ with fixed point operators or recursive operators. Then, the normalisation theorem is not valid anymore and the correspondence between programs and proofs is lost; however, the effectiveness of recursion, as an instrument to define functions, allows a greater simplicity for programming.

7. Semantics of Polymorphism

The mathematical meaning of polymorphism is relatively simple in the case of implicit polymorphism: it is only needed to correlate the meaning of terms without type to their version when assigned types. Consider thus the model $\underline{\mathbf{PN}} = (\mathbf{PN}, \mathbf{PN}0, \subseteq)$ of calculus without types mentioned at the end of § 5 and recall that in it every element $a \in \mathbf{PN}$ is an endomorphism and vice versa: ab interprets the functional application between terms, considered as elements of the model. To interpret types, construct the category PER of partial relations of equivalence on $\underline{\mathbf{PN}}$ in the following manner. Objects are *partial* equivalence relations, A, B… on $\mathbf{PN}$ (subsets of the product $\mathbf{PN} \times \mathbf{PN}$, which are symmetric and transitive, but need not be reflexive: elements need not be correlated to themselves). PER is Cartesian Closed. The base construction to verify it is quickly given: the internalisation of the morphism space between two objects A and B is the partial equivalence relation $A \to B$ such that $(d,d') \in (A \to B)$ if and only if, for all $(a,a') \in A$, $(da,d'a') \in B$; in other words, d and d' are equivalent in $A \to B$ if they transfom equivalent elements of A into equivalent elements of B. It is precisely this construction of the internal space of morphisms that leads to choose partial relations: take an arbitrary d and a rather small relation target B, then, d does

not associate all equivalent elements in A to equivalent elements in B. Thus, it is not generally the case that $(d,d) \in (A \to B)$, in other words, not all d is equivalent to itself in $A \to B$. In PER the affectation of types to terms without type has the following meaning: if a term c is formally given a type C, then the interpretation in P**N** of c, as a term without type, is equivalent to itself in the interpretation of type C as a partial relation of equivalence. Since this is true for each interpretation of free variables in c with elements in P**N** and variables (of type) free in C with objects of PER, we have given a correct meaning to polymorphism: every term, seen as an element of P**N**, has many relations, and in particular all those that interpret its formal types. One also succeeds in showing the interpretation of a term without type in the equivalence class of its interpretation as a term to which all the types are given. This completes the semantic correlation between terms, types, calculus without types and the assignement of type schemes.

To move on to explicit polymorphism, recall that it is based on a impredicative theory of types, a theory in which the type $\forall X : Tp.A$ is obtained quantifying on the collection of all types. The general mathematical meaning, in fact categorical, of this construction is not obvious. In the first place, it is a question of interpreting the universal quantification $\forall X : Tp$ as an *indexed* (or *generalised*) *product*. In fact, for the rule $(\forall E)$, if a: $\forall X : Tp.A$, then aB : $[B/X]A$; in other words, the term a is interpreted as a function that, taking (the interpretation of) a type B as argument, gives a result (the interpretation of) in a type A in which B is substituted for the variable X. This is precisely the intuitive definition of n product indexed by an arbitrary set, an idea that generalises the ordinary product between sets: a Cartesian product is a product indexed by a finite set. The difficulty lays in finding a category that is closed w. r. to a product indexed over the category itself: in fact, if the collection Tp of types is interpreted by the collection of objects of a category, $\forall X : Tp.A$ has to be interpreted as a product indexed over the category and also has to be an object of the category. The reader must pay attention to the strong reflexiveness or strong property of closure that we require; its rigorous understanding is a typical and beautiful side of Mathematics in which the geometrical and categorical instruments give meaning to symbols that otherwise would not be but a game of signs: we can obviously write $(\forall X : Tp.A) : Tp$, with Tp, the *definiendum* that appears in the *definiens*, it is here that it is interesting to comprehend if we mean something. Many indeed find this writing unacceptably circular, in spite of the important syntactic results of normalisation and Church-Rosser that have been cited. The construction of a sound mathematical model makes it acceptable.

Briefly, the category PER has the desired property of closure. To fully prove this however, it is necessary to immerse PER into a larger context, one in which a product indexed by PER itself can be defined. The idea is to find a category in which one can immerse PER as a subcategory, and at the same time, as an object on which the product can be defined. The answer is given thanks to an appropriate *topos*. These are categories with strong closure properties and provide models of intuitionist set theory. In a particular topos, said to be '*effective*' and built in generalising the

construction of the category PER, one can define the product indexed by PER, considered as an object of the topos itself. The result, say $\prod_{A\in \mathrm{PER}} F(A)$, is not only an object of the topos but of PER, which is also a subcategory of the topos. In other words, $(\prod_{A\in \mathrm{PER}} F(A))$, the product indexed over PER, is an object of PER, as a (sub-)category. In carrying out the construction step by step, one 'understands', mathematically, what an obscure impredicative formal definition can mean (see Asperti & Longo [1991]). In particular, the relations or sets of subsets of natural numbers, the objects of PER, are defined and understood independently of the product in question; thus the circularity in the syntax of the collection of types, *Tp*, defined while listing between the types also those obtained by the products indexed by the collection itself (or quantifications: $\forall X : Tp.A$), is reduced to proving a closure property of a predefined mathematical structure. Since this semantics is based on geometrical categories (the topos have their origin in Algebraic Geometry), that can as well provide an interpretation to intuitionist set theory, we have in a certain sense closed the scope of our understanding. We started from a formal propositional calculus and from terms of the intuitionist propositional calculus, both simple and second order calculi, in passing by topological and order structures, the relations on natural numbers, quotient spaces, etc… we have given an interpretation to apparently circular formalisms and found an independent link to other aspects of logic, and intuitionist set theory. The thus established unity between different theories gives a meaning, and adds to the comprehension, of each of them; it shows the proofs as programs, morphisms, calculable functions, while proposing a rigorous mathematical frame for programming in machines. This is only one possible semantic construction, but, as a side effect, it also helps to enrich the empirical methodology of programming: the untranslatable, non generalisable, 'ad hoc' solutions that are only understandable by the programmer, which are sources of mistakes, are "understood" in a unified manner, and in a uniform mathematical style.

8. Automatic Proof

We started this presentation by underlining the role of λ-calculus as a language for proofs: each term of the calculus with types is the coding of a proof and its reduction in normal form leads to a canonical or 'minimum' proof of the proposition corresponding to its type. Moreover, if a term without types admits a simple type, the assignement of a type to a term and its reduction to normal form can be done in an automatic way. They are thus the instruments for automatic proof, and in particular for the proofs of program properties. In fact, it is a program property to admit types or not; but not only this, a type is in reality a *specification* or a way to specify a program. While saying that a program goes from integers to booleans, one specifies a property that partially contributes to define it. In certain cases, the type can univocally determine the program: there exists only one program, that is understood as a term without free variables and that has type $\forall X.(X \to X)$, the identity. In other words, if one specifies a program as a function that applies each type to itself, the formal description of the type, of second order, univocally determines the program that calculates the identity function.

In general, as we have already said, the type assignement to a program is at the same time, a proof of logical propositions, the types, and a synthesis method of programs (which is automatic). Let us return to the second example in 2.1: stqrting with hypotheses on variables, one constructs a program of type $A \to (B \to A)$ step by step. In fact, the only program with type scheme $X \to (Y \to X)$. Thus already, in the current context, we can say that we have a partial method to prove propositions and program properties, and also of program *synthesis* (programs or proofs are synthetized by this proof method). A partial method, because, it requires an external (human) intervention in choosing the hypotheses that can be added if necessary along the deduction process, and also because, by comparison with the statements on programs, one can deduce in this manner only some properties and some programs.

Moreover, to speak of automatic proof as a substitute for human mathematical proof, we need to add at least the instruments that can formalise properties that are more or less commonplace in Mathematics. It is indeed obvious that mathematics is based from the start on the use of *individual* variables, for the elements of sets, and their quantification. In other words, it is necessary to be able to write formulas that describe statements as: given *any* integer *there exists* another integer that is larger than it; every element of a group has an inverse, etc… in the manner $\forall x \exists y (x < y)$, $\forall x \exists y (xy^{-1} = 1)$ … In the formulas, or types, introduced until now, there were no first order variables, or elements and we passed directly from simple propositional calculus to second order systems. These allow us to deal with the logical aspect of mathematical Analysis, not necesseraly its "mathematics".

Martin-Löf's (predicative) Intuitionist Type theory and Coquand and Huet's Calculus of Constructions (see Coquand & Huet [1988]), in different ways, extend the types of λ-calculus with the possibility to also define first order formulas as types, that is with the structure $\forall x{:}A.B$, where x is a variable of type A (see Hindley & Seldin [1986]). Indeed, λ-calculus already has first order variables: those that appear in the terms. As we have already said, λ-abstraction is a quantification over the terms, analogous to set abstraction: $\{x \mid \mathrm{P}(x)\}$ is the set of all elements x, of a certain universe, that satisfy property P. But, the variables are also terms, in full right, and they don't only have the meaning of elements as in Mathematics. In Mathematics, 'for every integer x…' only means that x can be particularised by an arbitrary integer. On the contrary, variable x, in λ-calculus, is *also* a term in normal form that can be manipulated, treated as the other terms, closed, constants,… The systems just mentioned use this linguistic richness of λ-calculus to give a unified treatment of formulas and types. Briefly, mathematical formulas are the first order terms and types, on which on can carry out computations in an automatic and uniform manner. For example, the formula $x(x + 1)=12$ becomes, by obvious calculation, $x^2 + x = 12$, where one has declared x as integer (a non erased assumption: x : Int). The solution of the equation is given by the particularisation or substitution of x by the integer 3. Note that the formulas or mathematical

propositions are not but "restrictions" to be satisfied: that is, propositions are types as specifications and a program is a proof of a specification or type, as in the already discussed cases. One can note here a substantial difference with Logic Programming where a program is a proposition and its evaluation is the proof of the proposition. The current interest of an integration of functional programming methods and logical programming ones is precisely due to the possibility of studying the synthesis of functional programs, from logical specifications, as a compilation method for logical programs in functional programming. Note though, as it has already been announced in the introduction, that we have omitted the principal instrument to deal with first order formulas or the restrictions that describe usual mathematical properties: unification techniques that handle uniform particularisation of first order formulas. These techniques, crucial in Logic Programming, are introduced in the Calculus of Constructions, both first and second order, the latter being what is new.

As a whole, the research direction that we have been detailing has given good results, if they are read with care. De Bruijn, for example, has developed mathematical jargon in the extensions of λ-calculus of an important intuitive effectiveness (see Hindley & Seldin [1986]). However, the truly important properties that can be handled are the properties of programs. The purely mathematical assertions proved in a strictly automatic fashion are not numerous. The problem is that the implicit richness in a true mathematical proof is found in the language changes, in the bridges and in the indirect analogies, in the superposition of methods. Their sterilising reduction in a single language, poor and static, can be of some effectiveness if enriched by the man-machine interaction. In this perspective, more instruments, to make proofs, are welcomed, including the automatic ones. Various computational algebras, or systems based on the language 'Mathematica' for example, provide greater effectiveness to the work of the algebrist by intervening when long calculations are needed, when innumerable explorations are required and in other tasks in which practical complexity renders computation impossible for a human being. The methods presented here, inspired by λ-calculus develop the complementary approach, while aiming to deduce the most for Mathematics from the logical formalisation. In spite of the limitative character and the failures of any "completeness program" (mathematics is completely formalisable), considered as a base project, from the practical standpoint, the interaction between man and machine can create miracles, but only if one accentuates the interactive character of proofs. Take, for example a proof by induction. λ-calculus perfectly describes the induction scheme and the inductive proofs are a typical candidate for automatic treatment. However, every one knows that in non trivial cases, the true mathematical problem, in a proof by induction, lays in the choice of induction hypothesis. Often the 'inductive load' has to be much stronger than the thesis; in other words, to inductively prove $\forall x.\mathrm{P}(x)$, one does not always succeed in proving that for every n, $\mathrm{P}(n)$ can be deduced from $\mathrm{P}(n-1)$ and it is necessary to turn to a property Q, stronger than P, to have that $\mathrm{Q}(n-1)$ implies $\mathrm{Q}(n)$.

The problem of choosing inductive loads is today a crucial problem in automatic proof. It is not clear in fact that it suffices to explore a finite number of possible inductive hypotheses or that, to obtain relatively complete methods, it is necessary to consider an infinite number. That is, if in the standard sectors of Mathematics, one has only to do long induction proofs or if, rather, such proofs are essentially difficult, due to the choice of inductive load: the choice criteria between an infinity of possible inductive hypotheses are difficult because they are generally external to a pre-established methodology, to a language and a formal frozen frame. They are based on 'intentional' choices where man integrates many experiences, uses analogy, refers to metaphors,… The analyses of these methods, as a part of the proof, is one of the stakes of the future, if one does not want to be restricted to a solely formal analysis of proofs that can be fully mechanised. The project then is the development of interactive programs of automatic synthesis of proofs where, for example, the user tells the machine the propositions to be proved in the induction (the inductive load) and lets the machine do the base work. The study of interactive or heuristic methods that are as automated as possible can lead to useful or acceptable systems, even if they remain incomplete.

9. Conclusion

In this presentation of λ-calculus we have sought with insistence to describe λ-terms, at the same time as programs, as codification of proofs and as category-theoretic morphisms. That is, we explained and enriched syntax by semantics and viceversa. By this, we underlined that the "cognitive" aspects of Mathematics are also present in proofs, if not in a principal role, but yet in bridges, in correlation between diverse contexts, where the informal suggestion, for instance, that allows to extend a language by a construction inspired by a model have a great practical and gnoseological interest. The proposal of new ideas and structures, the formulation of conjectures often are made possible thanks to a reflexive equilibrium of theories that integrate and explain each other mutually and are developed and modified in interaction. The unity of mathematics is given, not by a logical, linguistic or metaphysical unity but by the relations between theories and diverse languages. This is crucial, if we want to go further in automatizing as much as possible the deductive processes as well as in foundational analysis.

Luckily, the time is over for a ultimate and unique purely formal foundation of Mathematics, which inventors of λ-calculus and Combinatory Logic took as starting point in the Thirties. Only extreme formalists and reductionists are still pleased by heavy totalling programs. Their efforts sometimes are not useless, evven today, thanks to Computer Science, where pure minimal computations of signs can suggest more languages adapted to machines. Furthermore, this is exactly what is achieved by the examined systems: the formalist attitude at their origin has found an application, on the one hand within the rigidity of machines and it has been enriched, on the other, by its reencounter with mathematical Platonism or realism. In fact, this is the implicit or explicit

vision of researchers who seek meaning in geometric and categorical structures, while contributing to the intertwining of meaning to which they refer to.

More in general, the formalist approach and the Platonism spread in Mathematics, apparently completely different, have a common justification and origin in the observation that a mathematical concept, leaving aside the specific structure from where it emerges, acquires a generality and an independence that renders it applicable to many structures. Logical computations of which we spoke of here, in their perfect linguistic autonomy, that include provable consistency (see § 3), are a paradigmatic example of independence, generality and abstraction that make us 'forget' the structures that suggested them: algebras without variables for Combinatory Logic (Curry), the idea of (effective) function for λ-calculus (Church). Then it was not trivial to re-construct mathematical models that interpret the formalisations that have finally been reached. These structures branch out into many connections and applications, often well beyond expectation.

It is necessary to underline, to this effect, that it is precisely the generality and the independence from specific meanings that is the origin of this generality and objectivity of Mathematics: the importance of an idea and a theorem reside in their invariance by relation to linguistic notation and to particular mathematical structures. The concrete historical experience of this invariance w. r. to a plurality of practical universes, often as real as counting with numbers, is at the base of formalist foundations or ontological visions in Mathematics, as we said. Both, each in its own manner, attribute universality and existence to the pure, although very refined, linguistic and geometrical constructions of man, and are due to the mathematician's amazement when faced with the generality of these constructions nevertheless arisen from life and real world conceptual reconstructions. By mathematics we made the (physical) world intelligible and, then, by an abuse, we later detached one from the other.

One of our tasks today is to fix this formalism/Platonism breach and to surpass it, to understand mathematical proof not as pure formal calculus without meaning nor as a 'vision' of an external reality to man, given by concepts without conceptors. It is especially necessary to analyse them as conceptual constructions, rich with the plurality of human experiences, that go from a purely formal calculus to the practice of geometrical construction. As a matter of fact, meaning is involved in a non removable way in proofs: certain passages are possible only due to references to logical or geometrical structures rich in meaning to us. It is thus that certain recent incompleteness results point out a shift between formal principles of proof and principles of geometric construction (see Longo [1999a and b, 2001a]). Every analysis of proofs has well to deal with computations, but it also needs to be reorganised around a new unity between formal symbols and meaningful structures, where the organisation of space and time, with their symmetries, connectivity etc. give meaning, allow and structure the proof itself (see Longo [2001a and b] and Girard's program, Girard [1997, 2001]).

Some Bibliography

Abramsky, S. et al. (eds.) [1992] *Handbook of logic in computer science. Vol.* I and 2, Oxford: Clarendon Press.

Asperti A., Longo G. [1991] *Categories, Types, and Structures: an Introduction to Category Theory for the Working Computer Science*, MIT Press.

Barendregt H. [1986] *The Lambda Calculus, Its Syntax and Semantics*, revised edition, North-Holland.

Böhm C., Berarducci A. [1985] "Automatic synthesis of typed lambda-programs on term algebras" *Theor. Comp. Sci.* 39, pp. 135-154.

de Bruijn N. [1980] "A survey of the project AUTHOMATH" in *To H.B. Curry: essays in Combinatory Logic, lambda calculus and formalism* (Hindley, Seldin eds.), Academic Press.

Cardelli L., Longo G. [1991] "A semantic basis for Quest", *Journal of Functional Programming*, vol.1, n.2, 1991 (pp.417-458).

Church A. [1941] *The Calculi of Lambda Conversion*, Princeton University Press

Coquand T., Huet G. [19881 'The Calculus of Constructions' *Information and Computation*, 76, pp. 95-120.

Girard J.-Y. [1987] "Linear Logic" *Theoretical Comp. Sci.*, 50 (1-102).

Girard J.Y., Lafont Y., Taylor R. [1989] *Proofs and Types*, Cambridge University Press.

Girard J.-Y. [2001] "Locus Solum" *Mathematical Structures in Computer Science*, vol.11, n.3.

Goubault-Larrecq J., Mackie I. [1997] *Proof theory and automated deduction.* Dordrecht; Boston; London: Kluwer, 1997.

Hindley J.R., Seldin J.P. [1980] *To H.B. Curry: Essays on combinatory logic, Lambda Calculus and Formalism*, Academic Press.

Hindley J.R., Seldin J.P. [1986] *Introduction to Combinators and Lambda-Calculus*, Cambridge University Press.

Lambek J., Scott P.J. [1986] *Introduction to higher order Categorical Logic*, Cambridge University Press.

Longo G. [1983] "Set-Theoretical Models of Lambda-Calculus: Theories, Expansions, Isomorphis ms", *Annals Pure Applied Logic*, 24.

Longo G. [1999a] 'The mathematical continuum, from intuition to logic' in *Naturalizing Phenomen ology: issues in contemporary Phenomenology and Cognitive Sciences, (J.* Petitot et al., eds) Sta nford U.P..

Longo G. [1999b] "Mathematical Intelligence, Infinity and Machines: beyond the Gödelitis" *Journal of Consciousness Studies*, special issue on Cognition, vol. 6, 11-12..

Longo G. [2001a] "On the proofs of some formally unprovable propositions and Prototype Proofs in Type Theory" *Types' 00*, Durham, December 2000, invited lecture, to appear.

Longo G. [2001b], "The reasonable effectiveness of Mathematics and its Cognitive roots", in *New Interactions of Mathematics with Natural Sciences* (L. Boi ed.), Springer, to appear.

Mitchell J.C. [1993] *Introduction to Programming Language Theory*, M.I.T. Press.

Martin-Löf P. [1982] "Constructive logic and computer programming," *In Logic, Methodology and Philosophy of Science VI*, ed. L.J. Cohen et al. (eds.) North-Holland (pp. 153-175).

Martin-Löf P. [1984] *Intuitionistic Type Theory* Bibliopolis, Napoli.

Robinson J. [1965] 'A machine-oriented logic based on the resolution principle' *Journal of the Association for Computing Machinery*, 12, pp. 23-41.

Scott D. [1972] "Continuous lattices" *Toposes, algebraic Geometry and Logic*, (Lavwere ed.), SLNM 274, (pp.97-136) Springer-Verlag.

Scott D. [1976] "Data types as lattices," *SIAM Journal of Computing*, 5 (pp. 522-587).

Scott D. [1980] "Lambda-calculus, some models, some philosophy" *The Kleene Symposium* (Barwise et al. eds.) North-Holland.

Smyth M., Plotkin G. [1982] "The category-theoretic solution of recursive domain equations" *SIAM Journal of Computing 11*.