PROGRAMMING WITH PROOFS: A SECOND ORDER TYPE THEORY

Michel PARIGOT
Equipe de Logique, CNRS UA 753
Université Paris 7, UFR de Mathématiques,
2 place Jussieu, 75251 PARIS Cedex 05

Abstract

We discuss the possibility to construct a programming language in which we can program by proofs, in order to ensure program correctness. The logical framework we use is presented in [13].

The main objection to that kind of approach to programming being the inefficiency of the program produced by proofs, the greater part of the paper is devoted to investigate how to define data types and how to construct programs for combining proofs and efficiency. Several solutions are proposed using recursive data types which lead, in particular, to new representations of natural numbers in lambda-calculus.

Introduction: proofs as programs.

We know that in mathematics the problem of the correctness has been solved for a long time: the correctness of a (detailed) proof is easy to verify, even automatically. We also know that a constructive proof of a theorem has an algorithmic content. Putting these two facts together, we are tempted to consider **(formalized) mathematics as a programming language**. This would mean that writing a program satisfying some specifications becomes writing a proof of a statement expressing these specifications.

The theoretical basis of this approach exists: it can be found in the works of logicians on intuitionistic logic, essentially HEYTING, MARTIN-LOF and GIRARD. The **Heyting semantics** for intuitionistic logic explains proofs in terms of programs, and if we invert the perspective it gives a foundation for a programming language where programs are proofs (the essential point in this semantics is that a proof of $A \to B$ is an algorithm which transform each proof of A into a proof of B.).

There has been a lot of works using this principle, mainly N.G. deBRUIJN [2], R.L. CONSTABLE [5], J.C. REYNOLDS [19], T. COQUAND [3]. We will discuss here an other approach presented in [11] and [13], based on second order intuitionistic logic, which allows to write in a natural way exact specifications and correct programs.

The paper is organized as follows. In §1 we recall the general theory:

syntactic and semantic notions of type; definition of data types by second order formulas and extraction by proof of a representation of the data from the definition; expression of the specifications of a program and extraction by proof of a program which meets these specifications. In §2 we discuss the problem of efficiency and conclude to the necessity of new representations of data. In §3 we presents two kind of solutions using recursive definition of data types, which leads to new representations of the data allowing to write efficient programs: in particular we obtain new representations of natural numbers in lambda-calculus for which there exists a program computing the predecessor in one step. Finally we present in §4 a way of programming by proofs with these recursive data types.

1. INTUITIONISTIC SECOND ORDER LOGIC AS A PROGRAMMING LANGUAGE

1.1 Lambda-calculus as a machine language

To really become programs, proofs which naturally appear as trees, have to be linearly coded as terms of lambda-calculus.

The **terms** of lambda-calculus are obtained from variables x,y,z... by a finite number of applications of the following rules:

(a) if t and u are terms, then (t u) is a term - which represents the application of the function t to the argument u.

(b) if x is a variable and t is a term, then $\lambda x.t$ is a term - which represents the function $x \mapsto t$.

The notion of **computation** for terms is the reduction: a reduction in the term t is the replacement of the leftmost subterm of the form $(\lambda x.u\ v)$ by u[v/x] (i.e the result of substituting v to the occurrences of x in u). We say that a term t is reducible to a term t' (and note $t \Rightarrow t'$) if t' is obtained from t by a finite number of reductions. A term is normal if it does not contain a subterm of the form $(\lambda x.u\ v)$. In order to compute a function t on an argument u we reduce the term (t u); there are two possibilities: either we obtain, after a finite number of reductions, a normal term which is called the result of the computation, or there is an infinite number of possible reductions and the computation does not terminate. Using this notion of computation, all the recursive functions are representable by terms of the lambda-calculus.

It is possible to transform lambda-calculus in a programming language using an implementation of reduction. In fact it can be considered as a real machine language, where the symbols of lambda-calculus (,λ,x are interpreted as elementary instructions. Such an implementation has been realized by J.L. KRIVINE.

1.2. The logical framework

The logical language contains logical symbols, fixed parameters, and additional parameters depending of the data types we consider. The logical symbols are: the connective $\rightarrow$, the quantifier $\forall$, individual variables: x, y, z ..., predicate variables of arbitrary arity: X,Y,Z... . The fixed parameters are: a binary function constant **Ap** (application) and two individual constants **K**,**S** (combinators). Additional parameters contains predicate constants of arbitrary arity (predicate constants of arity 0 are called propositional constants), and function constants of arbitrary arity (function constants of arity 0 are called individual constants).

The individual terms and the (second order) formulas are defined in the usual way using this logical language. Note that in second order (intuitionistic) logic, the logical symbols $\bot, \wedge, \vee, \neg$ and $\exists$, as well as the identity relation $=$, are definable from $\rightarrow$ and $\forall$: for instance $A \wedge B$ is defined by $\forall X[[A \rightarrow [B \rightarrow X]] \rightarrow X]$.

The intended model $\mathcal{M}$, representing the programs from a denotational point of view, is the following. The universe is the set Λ of terms of lambda-calculus modulo reduction (more precisely modulo $\beta\eta$-reduction). The function **Ap** is interpreted by the function $u,v \longmapsto (u\ v)$. The constants **K** and **S** are interpreted by $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z))$ respectively. This model is in fact the usual way of coding lambda-calculus into a logical structure used in Combinatory Logic; it will be enriched by interpretations of the additional parameters. In the sequel term will also mean term modulo reduction.

The rules of proof for second order logic are the following (A, B denotes formulas and Γ sequences of formulas):

- R1 $A \vdash A$. (axiom)
- R2 if $\Gamma, A \vdash B$, then $\Gamma \vdash A \rightarrow B$. (abstraction)
- R3 if $\Gamma \vdash A \rightarrow B$ and $\Gamma \vdash A$, then $\Gamma \vdash B$. (application)
- R4 if $\Gamma \vdash A$ and x is an individual variable which does not occur free in Γ, then $\Gamma \vdash \forall x A$. (generalisation)
- R5 if $\Gamma \vdash \forall x A$ and b is an individual term, then $\Gamma \vdash A[b/x]$. (specialization)
- R6 if $\Gamma \vdash A$ and X is a predicate variable which does not occur free in Γ, then $\Gamma \vdash \forall X A$. (generalisation)
- R7 if $\Gamma \vdash \forall X A$ and B is a formula, then $\Gamma \vdash A[B/X]$. (specialization)
- R8 if $\Gamma \vdash A$, and Δ is obtained from Γ by permutation, contraction or extension, then $\Delta \vdash A$. (this rule will be omitted in formal derivations)
- R9 if $\Gamma \vdash \neg\neg A$, then $\Gamma \vdash A$ (RA)

The rules R1 to R8 are the rules for intuitionistic second order logic.

1.3 The syntactic notion of type.

From the point of view of Heyting semantics the rules of proof for intuitionistic logic can be considered as construction rules for programs (terms). Instead of handling formulas we handle expressions like t : A (read **t is a term of type A**), the hypothesis being replaced by variables declarations x : A. The rules become:

R1 $x : A \vdash x : A$.

R2 if Γ, $x : A \vdash t : B$, then $\Gamma \vdash \lambda x.t : A \rightarrow B$.

R3 if $\Gamma \vdash t : A \rightarrow B$ and $\Gamma \vdash u : A$, then $\Gamma \vdash (t\ u) : B$.

R4 if $\Gamma \vdash t : A$ and x is an individual variable which does not occur free in Γ, then $\Gamma \vdash t : \forall xA$.

R5 if $\Gamma \vdash t : \forall xA$ and b is an individual term, then $\Gamma \vdash t : A[b/x]$.

R6 if $\Gamma \vdash t : A$ and X is a predicate variable which does not occur free in Γ, then $\Gamma \vdash t : \forall XA$.

R7 if $\Gamma \vdash t : \forall XA$ and B is a formula, then $\Gamma \vdash t : A[B/X]$.

R8 if $\Gamma \vdash t : A$, and Δ is a sequence obtained from Γ by permutation, contraction or extension, then $\Delta \vdash t : A$.

From a programming point of view the expression "t is of type A", which relies a term to a formula, has to be read **"the program t realizes the specification A"**. Terms of lambda-calculus represent programs in machine language, whereas formu-las represent specifications expressed in a high level language. It remains to explain the relation between the program and the specification.

There are two well-known properties of terms obtained by proofs : **type preservation** and **termination**. The first one, which says that if a term t reduces to a term t' and t is of type A then t' is of type A, is essentially evident. The second one is much deeper: whereas the terms of lambda-calculus allow to represent all algorithms, including those which do not terminate, the terms obtained from proofs represent algorithm which always terminate. More precisely, each time we derive an expression "t : A" we are sure that t reduces to a normal term. This property becomes more and more difficult to prove when the expressive power of the logic increase (the proof for second order logic is due to GIRARD [7]).

There is one more essential property: second order intuitionistic logic can be considered as a programming language allowing to write **exact specifications and correct programs**. We will detail this point using a semantic notion of type.

1.4 The semantic notion of type.

The expression t : A (t is of type A) has been defined syntactically using the deduction rules for intuitionistic second order logic. But we can view a type A as a set, namely the set of terms of type A (or propositional traces of proofs of A). Doing so we can define a kind of intuitionistic semantic in the style of LAUCHLI [14]: a statement being interpreted by a subset of Λ (instead of a boolean value), a unary predicate by a function from Λ into $\mathcal{P}(\Lambda)$ (or equivalently by a binary relation on Λ),.... It can be defined using the classical semantic of formulas in the model Λ in the following way: we associate to each n-ary predicate variable X (resp. n-ary predicate constant P) a (n+1)-ary predicate variable X' (resp. (n+1)-ary predicate constant P'), and define inductively, for each formula A and variable y, a formula $y \in A$ as follows

$$y \in \Pi x_1 \ldots x_n := \Pi' x_1 \ldots x_n y \quad (\text{for } \Pi' \text{ predicate variable or constant})$$
$$y \in A \to B \quad := \forall z[z \in A \to (y\ z) \in B]$$
$$y \in \forall xA \quad := \forall x[y \in A]$$
$$y \in \forall XA \quad := \forall X'[y \in A]$$

Now we have a semantic notion of type $t \in A$ ("t is in the type A") - meaning that the statement $t \in A$ is true in the intended model. The following lemma shows that the semantic notion of type extends the syntactic one in the sense that all the programs typable from the syntactical point of view are also typable with the same type from the semantical point of view.

Conservation lemma: Let A be a statement. If $t : A$, then $t \in A$.

The equality between types, $A = B$, is defined as $\forall y[y \in A \leftrightarrow y \in B]$.

1.5 Definition of data types.

The second order formalism allows natural definitions for all the data types usually defined by induction: integers, lists, trees .... For instance, the set of natural numbers can be defined as "the smallest set containing zero and closed by the successor operation". Formally we introduce parameters for the constructors of the type: an individual constant $\underline{0}$ (for zero) and a function constant $\underline{s}$ (for the successor operation), and consider the formula Ix saying "x is a natural number"

$$\forall X[\forall y[Xy \to X\underline{s}y],\ X\underline{0} \to Xx].$$

(we use $A, B \to C$ as an abbreviation for $A \to [B \to C]$)

A representation of the constructor $\underline{0}$ in lambda-calculus is given by a term in the type $I\underline{0}$; it can be obtain by a formal derivation of $I\underline{0}$

$$f : \forall y[Xy \to X\underline{s}y],\ a : X\underline{0} \vdash a : X\underline{0} \qquad \text{(by R1)}$$
$$f : \forall y[Xy \to X\underline{s}y],\ \vdash \lambda a.a : X\underline{0} \to X\underline{0} \qquad \text{(by R2)}$$
$$\vdash \lambda f.\lambda a.a : \forall y[Xy \to X\underline{s}y],\ X\underline{0} \to X\underline{0} \qquad \text{(by R2)}$$
$$\vdash \lambda f.\lambda a.a : \forall X[\forall y[Xy \to X\underline{s}y],\ X\underline{0} \to X\underline{0}] \qquad \text{(by R6)}$$

We obtain the representation $0 = \lambda f.\lambda x.x$ for the constructor $\underline{0}$, which is precisely the Church numeral 0.

A representation of the constructor $\underline{s}$ in lambda-calculus is given by a term in the type $\forall x[Ix \to I\underline{s}x]$; it can be obtained by a formal derivation of $\forall x[Ix \to I\underline{s}x]$.

Let $\nu : Ix$. We look for a term of type $I\underline{s}x$, i.e. $\forall X[\forall y[Xy \to X\underline{s}y], X\underline{0} \to X\underline{s}x]$. Let $f : \forall y[Xy \to X\underline{s}y]$, and $a : X\underline{0}$. In this context we have $\nu : \forall y[Xy \to X\underline{s}y], X\underline{0} \to Xx$ and therefore $(\nu\ f\ a) : Xx$; because $f : \forall y[Xy \to X\underline{s}y]$, we have also $(f\ (\nu\ f\ a)) : X\underline{s}x$. Finally $\lambda\nu.\lambda f.\lambda x.(f\ (\nu\ f\ a)) : \forall x[Ix \to I\underline{s}x]$.
We obtain the representation $s = \lambda\nu.\lambda f.\lambda x.(f\ (\nu\ f\ a))$ for the constructor $\underline{s}$, which is precisely a term for the successor function on the Church numerals.

Now we can complete our intended model by interpreting $\underline{0}$ by 0 and $\underline{s}$ by the function generated by s. The crucial property is that the Church numeral $(s^n\ 0)$ is the unique term of type $I\underline{s}^n\underline{0}$. More precisely we define a **formal data type** as a formula A[x] such that there is an interpretation of individual and function constants such that the following holds in the model:

$$y \in A[x] \leftrightarrow y = x \wedge A[x].$$

It is readily seen that Ix is a formal data type. In fact all the usual data types can be defined by formal data types.

1.6 Logic as a high level programming language.

In order to program a function between data types (say the predecessor function from I to I), we have to

(a) introduce a function constant $\underline{p}$.

(b) find a set of equations which uniquely determine $\underline{p}$ on I, for example $\underline{p}\underline{0} = \underline{0}$ and $\underline{p}\underline{s}x = x$, and interpret $\underline{p}$ by a function satisfying these equations.

(c) derive a term of type $\forall x(Ix \to I\underline{p}x)$ using the previous set of equations.

Why do we obtain in that way a program for the predecessor function? Consider a term t of type $\forall x(Ix \to I\underline{p}x)$ and a term u satisfying Ix; because Ix is a formal data type, we have $(t\ u) = p[u]$; therefore t is a program for the function p on I and thus for the predecessor function.

This programming method extends to all the usual data types. The correctness of the programs is ensured by the way we derive them (we just have to verify that the deduction rules are well applied, and this can be checked automatically).

Example: a program for the addition

We introduce a binary function constant $\oplus$, and the usual equations defining addition: $x \oplus \underline{0} = x$, $x \oplus \underline{s}y = \underline{s}[x \oplus y]$. Then we look for a term t of type $\forall x \forall y[Ix, Iy \to I[x \oplus y]]$.

Let $\nu : Ix$, $\mu : Iy$, $f : \forall y[Xy \to X\underline{s}y]$, and $a : X\underline{0}$; we have to find a term of type $X[x \oplus y]$ in this context; we proceed by induction i.e. we look for terms of

type $X[x \oplus 0]$ and $\forall z[X[x \oplus z] \to X[x \oplus \underline{s}z]]$.
Clearly $(\nu\ f\ x)$ is of type Xx and thus of type $X[x \oplus \underline{0}]$ (by the first equation).
By R5 and R4, f is of type $\forall z[X[x \oplus z] \to X\underline{s}[x \oplus z]]$ and by the second equation of type $\forall z[X[x \oplus z] \to X[x \oplus \underline{s}z]]$. By R7, with $B = X[x \oplus .]$, μ is of type
$\forall z[X[x \oplus z] \to X[x \oplus \underline{s}z]], X[x \oplus \underline{0}] \to X[x \oplus y]$.
Therefore $(\mu\ f\ (\nu\ f\ x))$ is of type $X[x \oplus y]$. Finally $\lambda\nu.\lambda\mu.\lambda f.\lambda x.(\mu\ f\ (\nu\ f\ x))$ is of type $\forall x \forall y[Ix, Iy \to I[x \oplus y]]$ and thus a program for addition.

2. THE QUESTION OF EFFICIENCY

Intuitionistic logic certainly provides a programming language allowing to write exact specifications and correct programs. But doing so, correctness has a counterpart: programs are often not efficient. There is first a practical reason: programming being reduced to the search of proofs, one can write programs without thinking at "how the program works". But in fact, <u>inefficiency</u> is the main objection to the "programming by proof" approach: we can distinguish three theoretical sources of inefficiency:

(i) the better proofs from a conceptual point of view are not necessary the better ones from an algorithmic point of view.

(ii) the efficient algorithms of pure lambda-calculus do not always come from proofs, even in second order intuitionistic logic.

(iii) the efficient algorithms are not always expressible in pure lambda-calculus.

The order in which these problems are enumerated indicates the increasing difficulties to find remedies for them. The first one does not call our approach in question: it just says that we must learn what are the best proofs from an algorithmic point of view (for instance, an immoderate use of proofs by induction can lead to disastrous algorithms). The second one objects to the choice of the high level language: the crude version of second order intuitionistic logic does not provide a real programming language, and we must at least construct more elaborate versions. The third one is a priori more worrying: our machine language seems too weak; fortunatly there is another possible diagnosis: the existence of some algorithms depends on the coding of the data in lambda-calculus. Let us give simple examples for the last two problems.

In order to program the inf of two natural numbers we have to deduce the statement "if x and y are natural numbers, then inf[x,y] is a natural number" from the set of equations defining the inf; doing this we must choose x or y as base of the induction; if for instance we choose x, then the resulting program will need at least 512 steps to compute inf[512,0]! In fact there exists a program in lambda-calculus

which computes inf[x,y] in inf[x,y] steps, but it doesn't come from a proof (thisresult is proved in J.L. KRIVINE [12]).
The situation get worse if we try to construct a program computing the predecessor of a natural number. The classical definition of natural numbers by induction gives the Church numerals (which are of the form $\lambda f.\lambda x.(f^n\ x)$); whereas the program for the successor function is obtained by a direct proof, all the programs for the predecessor function require a proof by induction, which means from an algorithmic point of view that the computation of the predecessor of 512 uses at least 512 steps! Moreover, the proof by induction which gives a "bad" algorithm is essentially the only possible one, and even in pure lambda-calculus no better algorithm exists.The same phenomena appears for all the usual data types: we are in the situation of a LISP language without direct access to the **cdr** of the lists (each time we need to recalculate the entire list). Such a situation become disastrous when we execute complicated programs such as sorting programs where the call to the **cdr** is iterated.

How to solve these basic difficulties without destroying the essential, i.e. programming with proofs in order to obtain correct programs? There are at least two ways that give enought new algorithms: we can either extend lambda-calculus, or change the definition of the data types. Here we will take the second way and present a solution based on the semantic notion of type. Because of lack of space, we only investigate the example of the predecessor.

3. RECURSIVE DATA TYPES

We look for definitions of data types satisfying the following requirements:

(a) the definition must be a formal data type in order to obtain correct programs.

(b) the representation of data must allow efficient programming (in particular direct access to **cdr**).

(c) the formula defining the data type must remain closely related to our intuition of the data type.

Two kinds of solutions have a particular interest: we will present them for the type of natural numbers, but they easely extend to all usual data types. They are based on a deep use of the semantic notion of type: instead of considering a type as a formula, we will consider a type as predicate defined by axioms.

3.1 The type Number

We introduce a unary predicate constant **N** and two constructors $\underline{0}$ (for zero) and $\underline{\sigma}$ (for the successor function). We define the intuitionistic interpretation $y \in \mathbf{N}x$ of the type Number $\mathbf{N}$ as the minimal solution K of the equation

$$Kx = \forall X[\forall y[Ky,\ Xy \to X\underline{\sigma}y],\ X\underline{0} \to Xx]$$

Note that because K occurs positively in $\forall X[\forall y[Ky,\ Xy \to X\underline{\sigma}y],\ X\underline{0} \to Xx]$,this solution exists.

One possible motivation for this recursive definition of the type Number is the following: the equation $\mathbf{N}x = \forall X[\forall y[\mathbf{N}y,\ Xy \to X\underline{\sigma}y],\ X\underline{0} \to Xx]$ is a possible formulation of induction where the induction step is not formulated for arbitrary elements but just for natural numbers.

Representation of the constructors.

A representation 0 of the constructor $\underline{0}$ in lambda-calculus is given by a term in the type $\mathbf{N}\underline{0}$, i.e. in the type $\forall X[\forall y[\mathbf{N}y,\ Xy \to X\underline{\sigma}y],\ X\underline{0} \to X\underline{0}]$. Clearly $\lambda f.\lambda a.a$ is in this type, and we have the same representation as for the type Iterator.

A representation σ of the constructor $\underline{\sigma}$ in lambda-calculus is given by a term in $\forall x[\mathbf{N}x \to \mathbf{N}\underline{\sigma}x]$. Let $\nu \in \mathbf{N}x$, $f \in \forall y[\mathbf{N}y,\ Xy \to X\underline{\sigma}y]$ and $a \in X\underline{0}$; then $(\nu\ f\ a) \in Xx$ and $(f\ \nu) \in Xx \to X\underline{\sigma}x$; therefore $((f\ \nu)\ (\nu\ f\ a)) \in X\underline{\sigma}x$ and finally $\lambda\nu.\lambda f.\lambda a.((f\ \nu)\ (\nu\ f\ a)) \in \forall x[\mathbf{N}x \to \mathbf{N}\underline{\sigma}x]$.

We complete our intended model by interpreting $\underline{0}$ by 0 and $\underline{\sigma}$ by the function generated by σ. Because we have a predicate instead of a formula we must also define the classical interpretation $\mathbf{N}x$ of the type Number $\mathbf{N}$. Of course, we take "the smallest set containing 0 and closed by the function generated by σ"; this mean that the following holds in the model: $\mathbf{N}x \leftrightarrow \forall X[\forall y[Xy \to X\underline{s}y],\ X\underline{0} \to Xx]$. It is easy to see that with this interpretation $\mathbf{N}x$ is a formal data type.

An equivalent definition of the type Number.

We can give an inductive definition of the type Number using a universal formal data type $\mathbf{U}$ defined as follows: the intuitionistic interpretation is defined in the model by $\forall x \forall y[x \in \mathbf{U}y \leftrightarrow x = y]$, and the classical one is just Λ (note that this type is not syntactically definable). The new definition of the type Number $\mathbf{N}x$ is just $\forall X[\forall y[\mathbf{U}y,\ Xy \to X\underline{\sigma}y],\ X\underline{0} \to Xx]$.

This definition gives a new intuition of the type Number: take the definition $\mathbf{I}x$ of the type of iterators but replace the trivial interpretation of the universal quantification by the traditional constructivist one: a proof of $\forall xA$ is a function which associates to each element u of the domain a proof of $A[u/x]$.

Example: a program for the predecessor

The type Number has an essential property which is due to its recursive definition: there exists a program which compute the predecessor in one step (to be more precise: by five elementary reductions). We introduce a new function constant $\underline{q}$ and two axioms which define $\underline{q}$ semantically

$\underline{q}\underline{0} = \underline{0}$

$\underline{q}\underline{\sigma}x = x$

We have to find a term $t \in \forall x[Nx \to N\underline{q}x]$. Let $\nu \in Nx$; by specialization to $N\underline{q}.$, we obtain $\nu \in [\forall y[Ny, N\underline{q}y \to N\underline{q}\underline{\sigma}y], N\underline{q}\underline{0} \to N\underline{q}x]$, and by the equations

$$\nu \in [\forall y[Ny, N\underline{q}y \to Ny], N\underline{0} \to N\underline{q}x].$$

Clearly $\lambda x.\lambda y.x \in \forall y[Ny, N\underline{q}y \to Ny]$ and $\lambda x.\lambda y.y \in N\underline{0}$;
therefore $(\nu\ \lambda x.\lambda y.x\ \lambda x.\lambda y.y) \in N\underline{q}x$ and $\lambda\nu.(\nu\ \lambda x.\lambda y.x\ \lambda x.\lambda y.y) \in \forall x(Nx \to N\underline{q}x)$.

Example: inductive programming on the type number.

Though the type Number has a recursive definition it allows to program using proofs by induction. As an example we will construct a program which translates numbers into iterators. We introduce an unary function constant $\underline{h}$ and axioms semantically defining $\underline{h}$

$\underline{h}\underline{0} = \underline{0}$

$\underline{h}\underline{\sigma}x = \underline{s}\underline{h}x$

We have to find a term $t \in \forall x(Nx \to I\underline{h}x)$. Let $\nu \in Nx$. By specialization to $I\underline{h}.$, we obtain $\nu \in [\forall y[Ny, I\underline{h}y \to I\underline{h}\underline{\sigma}y], I\underline{h}\underline{0} \to I\underline{h}x]$ and using the equations
$\nu \in [\forall y[Ny, I\underline{h}y \to I\underline{s}\underline{h}y], I\underline{0} \to I\underline{h}x]$. We have $\lambda u.\lambda v.(s\ v) \in \forall y[Ny, I\underline{h}y \to I\underline{s}\underline{h}y]$ and $\lambda f.\lambda x.x \in I\underline{0}$. Therefore $(\nu\ \lambda u.\lambda v.(s\ v)\ \lambda f.\lambda x.x) \in I\underline{h}x$ and
$\lambda\nu.(\nu\ \lambda u.\lambda v.(s\ v)\ \lambda f.\lambda x.x)$ is the term we looked for.

A comparison.

There is an interesting formal analogy between this new representation of the natural numbers and the Von Neumann's representation of natural numbers in Set Theory:

n+1 in Lambda-calculus	$\lambda f.\lambda x.(f\ n\ (n\ f\ x))$
n+1 in Set Theory	$\{n\} \cup n$

In the first part of the representation of n+1, we have n as a complete entity, whereas in the second part n is executed: it is precisely the first part of the representation which allows to compute directly the predecessor.

From a computational point of view, this representation has an apparent inconvenient from the programming point of vue: as for the Von Neumann's representation, the developed form of the natural number n has lenght 2^n. We will see later that this is no real problem; however, it would be nice to have a type for natural numbers with direct access to the predecessor and a developed representation of n of lenght n. This is in fact possible: in the same way the iterators are obtained from the numbers by just keeping the second part of the representation, we can obtain other natural numbers (called "stacks"), which will have the required properties, by just keeping the first part of the representation.

3.2 The type Stack

We introduce a unary predicate constant S and two constructors $\underline{0}$ (for zero) and $\underline{\tau}$ (for the successor function). We define the intuitionistic interpretation $y \in Sx$ of the type Stack S as the minimal solution K of the equation

$$Kx = \forall X[\forall y[Ky \to X\underline{\tau}y],\ X\underline{0} \to Xx]$$

Note that the definitions of the types Iterator and Stack are both obtained from that of the type Number by removing a part of the definition: either Ky or Xy.

Representation of the constructors.

The representation of the constructors of the type Stack is obtained as for the type Number: the representations of $\underline{0}$ and $\underline{\tau}$ are respectively $0 = \lambda f.\lambda a.a$ and $\tau = \lambda\nu.\lambda f.\lambda a.(f\ \nu)$. We interpret $\underline{0}$ by 0 and $\underline{\tau}$ by the function generated by τ, and define the classical interpretation Sx of the type Stack in the model by: $Sx \leftrightarrow \forall X[\forall y[Xy \to X\underline{\tau}y],\ X\underline{0} \to Xx]$. With this interpretation, Sx is a formal data type.

Example: a program for the predecessor

We introduce a new function constant $\underline{r}$ and two axioms which define $\underline{r}$ semantically: $\underline{r0} = \underline{0}$ and $\underline{r\tau}x = x$.

We have to find a term $t \in \forall x[Sx \to S\underline{\tau}x]$. Let $\nu \in Sx$; by specialization to $S\underline{r}$. we obtain $\nu \in [\forall y[Sy \to S\underline{r\tau}y],\ S\underline{r0} \to S\underline{r}x]$, and by the equations

$$\nu \in [\forall y[Sy \to Sy],\ S\underline{0} \to S\underline{r}x].$$

Clearly $\lambda x.x \in [\forall y[Sy \to Sy]$ and $\lambda x.\lambda y.y \in S\underline{0}$;

therefore $(\nu\ \lambda x.x\ \lambda y.\lambda x.y) \in S\underline{r}x$ and $\lambda\nu.(\nu\ \lambda x.x\ \lambda y.\lambda x.y) \in \forall x(Sx \to S\underline{r}x)$.

In the case of stacks we again obtain a program which compute the predecessor in one step, and this time we have a representation of lenght n for the natural number n. There is however an apparent problem: how can we make proofs by induction with this purely recursive definition of the type?

4. PROGRAMMING USING RECURSIVE DATA TYPES

In the case where a type is defined by induction, like the type Iterator, we obtain directly a term for the proofs by induction. For instance the term $ind = \lambda x.\lambda f.\lambda\nu.(\nu\ f\ x)$ is in the type $\forall X[X0,\ \forall y[Xy \to X\underline{s}y] \to \forall x[Ix \to Xx]]$, and comes directly from a proof of this statement. But for types having a purely recursive definition, we have to construct such a term using a metareasoning.

Induction on the type Stack

We look for a term rec in the type $\forall X[X0,\ \forall y[Xy \to X\underline{\tau}y] \to \forall x[Sx \to Xx]]$;

assuming $\alpha \in X0$ et $\beta \in \forall y[Xy \to X\underline{\tau}y]$, we have to find a term γ in the type $\forall x[Sx \to Xx]$.

<u>lemma</u>: if γ satisfies the equations

$$(\gamma\ 0) = \alpha$$

$$(\gamma\ \underline{\tau}y) = (\beta\ (\gamma\ y)),$$

then $\gamma \in \forall x[Sx \to Xx]$.

<u>proof</u>. We have to prove that $(\gamma\ y) \in Xx$ follows from the hypothesis $y \in Sx$; because Sx is a formal data type, it suffices to prove that $(\gamma\ x) \in Xx$ follows from the hypothesis Sx, or equivalently from the hypothesis $\forall X[\forall y[Xy \to X\underline{\tau}y],\ X\underline{0} \to Xx]$. We proceed by induction (formally this means that we specialize the previous formula to $(\gamma\ .) \in X.$): for $x = 0$, the first equation gives the result; now assume that $(\gamma\ y) \in Xy$; because $\beta \in \forall y[Xy \to X\underline{\tau}y]$ we have $(\beta\ (\gamma\ y)) \in X\underline{\tau}y$ and by the second equation $(\gamma\ \underline{\tau}y) \in X\underline{\tau}y$.

It remains to find γ satisfying the equations of the lemma. Because elements of the type Stack are binary functions, we look for a term γ of the form $\lambda x.t[(x\ \rho\ \iota)]$ where t,ρ,ι are unknown (intuitively, ι is the initial condition and ρ the recursive one). It follows

$$(\gamma\ \underline{0}) = t[(\underline{0}\ \rho\ \iota)] = t[\iota]$$

$$(\gamma\ \underline{\tau}y) = t[(\underline{\tau}y\ \rho\ \iota)] = t[(\rho\ y)]$$

and the equations become

$$t[\iota] = \alpha$$

$$t[(\rho\ y)] = (\beta\ t[(y\ \rho\ \iota)]).$$

Because ρ appears in the second part of the equation, we will have a recursive call of ρ. The simplest possible form for t is $t[z] = (z\ \rho)$. In this case the equations are

$$(\iota\ \rho) = \alpha$$

$$(\rho\ y\ \rho) = (\beta\ (y\ \rho\ \iota\ \rho)).$$

We can take

$$\iota = \lambda d.\alpha$$

$$\rho = \lambda y.\lambda r.(\beta\ (y\ r\ \iota\ r)).$$

Finally we have $\gamma = \lambda x.(x\ \rho\ \iota\ \rho)$, with the previous values for ρ and ι.

<u>Example</u>.

Having the possibility of reasoning by induction on stacks, we are able to construct all the programs we want using the general method presented for the type iterator. As an example we give a program translating stacks into iterators. Consider a unary function constant f and the following set of equations which semantically define the translation

$f\underline{0} = \underline{0}$

$f\underline{\tau}x = \underline{s}fx$.

We have to find a term t in $\forall x[Sx \rightarrow Ifx]$. It follows directly from the equations that

$0 \in If\underline{0}$

$s \in \forall y[Ify \rightarrow If\underline{\tau}y]$.

(recall that 0 and s are the programs for the constructors of the type I)

Therefore $\lambda x.(x\ \rho\ 0\ \rho)$ with $\rho = \lambda y.\lambda r.(s\ (y\ r\ 0\ r)$ is in the type $\forall x[Sx \rightarrow Ifx]$.

Two kinds of induction on the type Number.

In the case of the type **N**, we have two terms for proofs by induction, which gives two completely different programming methods. The first one, which is the analogue of the term ind for the type I, follows the inductive definition of the type **N**: a direct proof shows that $\lambda x.\lambda f.\lambda \nu.(\nu\ \lambda d.f\ x)$ is in the type $\forall X[X\underline{0}, \forall y[Xy \rightarrow X\underline{\tau}y] \rightarrow \forall x[Ix \rightarrow Xx]]$. The second one is obtained using the same reasoning as in the case of the term rec for the type **S**, and has the same recursive nature: this term is $\lambda x.\lambda f.\lambda \nu.(\nu\ \rho\ \iota\ \rho)$, with $\iota = \lambda d.x$ and $\rho = \lambda y.\lambda z.\lambda r.(f\ (y\ r\ \iota\ r))$.

Therefore, for the type number we can choose, depending of the function we want to compute, either an inductive programming method or a recursive one or even mix them together, and this flexibility increases our ability to write efficient programs.

We will now briefly explain why the fact that the number n has a normal representation of lenght 2^n is not an objection for programming. We have to distinguish between execution (which is not a rewriting for the implementation we have in mind) and output storage. For input and output, we can choose a representation of lenght n for the number n, for instance $(\sigma^n\ 0)$. During the execution the developped form of n, which is a binary tree of height n, never appears: execution corresponds to a run along a branch of the tree (the two programming methods correspond to runs along the left-most branch and the right-most branch).

A TEMPORARY CONCLUSION

In the case of inductively defined data types, it is possible to program just using proofs. But doing so we are condemned to write inefficient programs. On the other hand with recursive data types, we have to construct preliminary tools using a different method; but then we can write efficient programs just using proofs (and thus ensuring correctness).

Once we have found the term rec, we can give an alternative presentation of the type **S** in the style of P. MARTIN-LOF [16]:

S-introduction

$$0 \in \mathbf{S}\underline{0} \qquad \frac{y \in \mathbf{S}x}{(\tau\ y) \in \mathbf{S}\underline{\tau}y}$$

S-elimination

$$\frac{c \in \mathbf{S}x \qquad \alpha \in X\underline{0} \qquad \begin{matrix} u \in Xy \\ (\beta\ u) \in X\underline{\tau}y \end{matrix}}{(\mathrm{rec}\ \alpha\ \beta\ c) \in Xx}$$

An essential difference with MARTIN-LOF's approach is that **the definition of the type S gives the implementation of 0, τ and rec**.

The point which seems to be a difficulty for programming with recursive data types in comparison with inductive data types, namely the fact that for each data type we have to construct preliminary tools like rec, can be overcome using an other programming method based on the universal data type **U** and a fixed point operator (see [18]).

BIBLIOGRAPHIE

[1] H.P. BARENDREGT, *The Lambda Calculus*, Studies in Logic, North-Holland, 1981.

[2] N. DE BRUIJN, *A survey of the project Automath*, to H.B. CURRY: essays on combinatory logic, λ-calculus and formalism, Seldin/Hindley (eds), pp 579-606, Academic Press, 1980.

[3] T. COQUAND, *Une théorie des constructions*, Thèse de 3eme cycle, Université Paris 7, 1985.

[4] T. COQUAND, G. HUET, *Constructions: a higher order proof system for mechanizing mathematics*, Proc. EUROCAL 85, LNCS 203.

[5] R.L. CONSTABLE et. al., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[6] G. COUSINEAU, P.L. CURIEN, M. MAUNY, *The categorical abstract machine*, LNCS 201, 1985.

[7] J.Y. GIRARD, *Une extension de l'interprétation de Gödel à l'analyse, et son application àl'élimination des coupures dans l'analyse et dans la théorie des types*, Proc. 2nd Scandinavian Logic Symp., pp 63-92, North-Holland, 1970.

[8] J.Y. GIRARD, *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Thèse d'état, Université Paris 7, 1972.

[9] J.Y. GIRARD, *The System F of variable types, fifteen years later*, Theoretical Computer Science, 1987.

[10] W.A. HOWARD, *The formulae as types notion of construction*, manuscript, 1969 (published in Seldin/Hindley (eds), To H.B. CURRY: essays on combinatory logic, λ-calculus and formalism, pp 479-490, Academic Press, 1980).

[11] J.L. KRIVINE, *Programmation en Arithmetique Fonctionnelle du Second Ordre*, manuscript.

[12] J.L. KRIVINE, *Un algorithme non typable dans le système F*, CRAS, 1987.

[13] J.L. KRIVINE, M. PARIGOT, *Programming with proofs*, preprint, presented at 6th Symposium on Computation Theory, Wendisch-Rietz, November 1987.

[14] H. LAUCHLI, *An abstract notion of realizability for which intuitionistic predicate calculus is complete*, in Kino/Myhill/vesley (eds), Intuitionism and proof theory, pp 227-234, North-Holland, 1970.

[15] P. MARTIN-LOF, *Constructive Mathematics and Computer programming*, Proc. 6th Cong. Logic, Methodology and Philosophy of Science, pp 153-175, North-Holland, 1982.

[16] P. MARTIN-LOF, *Intuitionistic type theory*, Bibliopolis, 1984.

[17] M. PARIGOT, *Preuves et programmes: les mathématiques comme langage de programmation*, *Images des Mathématiques*, Courrier du CNRS (à paraître).

[18] M. PARIGOT, *Recursive programming with proofs*, preprint, december 1987.

[19] J.R. REYNOLDS,*Three approaches to type structure*, LNCS 185, 1985, pp 97-138.