ON THE REPRESENTATION OF DATA IN LAMBDA–CALCULUS

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

Abstract

We analyse the algorithmic properties of programs induced by the choice of the representation of data in lambda–calculus. From a logical point of view there are two canonical ways of defining the data types: the iterative one and the recursive one. Both define the same mathematical object, but we show that they have a completely different algorithmic content. The essential of the difference appears in the operational properties of two programs: the predecessor and the addition on the type of unary natural numbers (for the type of lists this would be the programs cdr and append). The results we prove in this paper state a fundamental duality between the iterative and recursive representation of data in lambda–calculus.

For the iterative representation of natural numbers (Church numerals) there is a "one–step" addition, but we prove in §3 that there is no "one–step" predecessor (by "one–step" we mean "whose computation requires only number of reduction steps bounded by a constant"). For the recursive representation of natural numbers we have the converse situation: there is a "one–step" predecessor but we prove in §4 that there is no "one–step" addition. For simplicity, we state these results for the type of natural numbers, but they hold in fact for all the usual data types defined as multisorted term algebras. Their practical significance for programming, may be, appears clearer on the type of lists where the predecessor is replaced by the cdr and the addition by append.

In §5, we briefly present a new representation of natural numbers for which we have both, a "one–step" predecessor and a "one–step" addition.

§1 INTRODUCTION

In a programming language based on (second order) typed lambda–calculus, the user will have the possibility to define his own data types, the representation in lambda–calculus (and thus in machine language) being automatically extracted from the logical definition of the data type. This way has been explored by several people (for instance [Le83], [BB85], [KP87]). At the logical level the data types are defined by second order formulas expressing the usual iterative definition of the corresponding algebras of terms, and the data receive the corresponding iterative definition in lambda–calculus. Programs on these data types are constructed from equational specifications using proofs. Well–known metamathematical results say that we can obtain in that manner programs for all the functions whose termination is provable in second order Peano arithmetic, that is in

practice, for all the total functions we need. Unfortunately this extensional result (it concerns functions) doesn't extend to an intensional one (concerning programs). We can get programs for all the functions we need, but not necessary programs having the intended behaviour (in terms of time complexity, for instance).

We show in this paper that the logical definitions of the data types have in fact an algorithmic content, which generates constraints on the possible behaviours of the programs constructed using them. For simplicity we state and prove our results for the type of unary natural numbers, but the proofs easely generalize to all the usual data types defined as multisorted term algebras, such as lists, trees,... .

From a logical point of view, there are two canonical ways to define the data types: the iterative one (used in [Le83], [BB85], [KP87]) and the recursive one (used in [Pa88a]). In the first case we obtain an iterative representation of natural numbers:

$0 = \lambda f.\lambda x.x$ and $n + 1 = \lambda f.\lambda x.(f)((n)f)x$

In the second case we obtain a recursive representation of natural numbers:

$0 = \lambda f.\lambda x.x$ and $n + 1 = \lambda f.\lambda x.(f)n$

Our results show a fundamental duality between these two representations, coming from the corresponding logical definitions:

– For the iterative representation of natural numbers there is a "one–step" addition, but no "one–step" predecessor (theorem 1).

– For the recursive representation of natural numbers there is a "one–step" predecessor but no "one–step" addition (theorem 2).

(by "one–step" we mean "whose computation requires only a number of reduction steps bounded by a constant).

The "negative" part of the results have important consequences from the point of view of programming. The lack of a "one–step" predecessor for the iterative representation means that each time we have to iterate programs using the predecessor (and more generally the destructors of an iterative data type, like cdr for lists) we will get inefficient programs. This situation appears frequently for n–ary functions: for instance, we are not able to get, using a proof, a program comparing two natural numbers n and m in a time inf(n,m) (in fact we conjecture that no such program exists; this would mean that the usual second order typed lambda–calculus is not a realistic model for functional programming). On the other hand the problem disappear with the recursive representation: for instance, we easely obtain, using a proof, a program comparing two natural numbers n and m in a time inf(n,m).

From a theoretical point of view, the results not only concern the particular representation of data in lambda–calculus, but the iterative and recursive definitions themselves. For instance, the results for the recursive case also hold in the Gödel system T and the Martin–Löf theory of types.

At the end of the paper, we briefly present a new representation of natural numbers for which we have both, a "one–step" predecessor and a "one–step" addition. The representation in lambda–calculus is very simple, but its logical meaning is not as immediate as those of the other two.

§2 ITERATIVE VERSUS RECURSIVE REPRESENTATION OF DATA

2.1 Preliminaries.

Our basic system for constructing programs, is a variant of the second order typed lambda–calculus (used in [Le83] and [KP87])

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.

(b) if x is a variable and t is a term, then $\lambda x.t$ is a term.

The set of terms of lambda–calculus is denoted by Λ.

(we use the notation (t)u instead of the more usual one (t u) for technical reasons: we will have to consider terms as words on an alphabet, and for the notation (t u) we have to replace the "blank" between t and u by a letter, for instance ")")

The formulas (or types) of second order logic are constructed using the connective $\rightarrow$, the quantifier $\forall$, individual variables: x, y, z ..., unary predicate variables: X,Y,Z..., predicate, function and individual constants. The individual terms are constructed from the function and individual constants. The formula $A \rightarrow [B \rightarrow C]$ is denoted by $A,B \rightarrow C$.

The usual rules of proof for intuitionistic logic are considered as construction rules for terms as follows (for a set $\mathcal{e}$ of equational axioms depending on the program we construct):

$\Gamma, x : A \vdash_e x : A.$

$$\frac{\Gamma,\ x :\ A \vdash_e t\ :\ B}{\Gamma \vdash_e \lambda x.t\ :\ A \rightarrow B} \qquad \frac{\Gamma \vdash_e u : A \qquad \Gamma\ \vdash_e t : A \rightarrow B}{\Gamma \vdash_e u :\ A}$$

$$\frac{\Gamma\ \vdash_e t :\ A}{\Gamma\ \vdash_e t : \forall x A}\ (1) \qquad \frac{\Gamma\ \vdash_e t :\ \forall x A}{\Gamma\ \vdash_e t : A[\tau/x]}\ (3)$$

$$\frac{\Gamma\ \vdash_e t :\ A}{\Gamma\ \vdash_e t : \forall X A}\ (2) \qquad \frac{\Gamma\ \vdash_e t :\ \forall X A}{\Gamma\ \vdash_e t : A[B/X]}\ (4)$$

$$\frac{\Gamma \vdash_e t : A[u] \qquad e \vdash u = v}{\Gamma \vdash_e t : A[v]}$$

(1) x has no free occurences in Γ; (2) X has no free occurences in Γ; (3) τ is an individual term; (4) B is a formula with a distinguished individual variable.

This system is specially simple: $\rightarrow$ is the only logical symbol having an algorithmic content; the quantifications have just a conceptual role, they allow to describe what the programs do. It is well known that the terms derived in this system satisfy the **strong normalisation property** (follows from [Gi72]): if $\Gamma \vdash_e t : A$, then all the reduction sequences beginning with t are finite.

2.2 Iterative representation of data.

The second order formalism allows natural definitions for all the data types usually defined by induction: numbers, lists, trees ....(see [Le83],[BB85],[KP87]). 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 Nx saying "x is a natural number"

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

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

A representation of the constructors, destructors and recursors of the data type in lambda–calculus (and thus in machine language) is given by the logical definition of the data type, using (normal) proofs.

A representation of the constructor $\underline{0}$ in lambda-calculus is given by a proof of $N\underline{0}$. The proof gives the representation $0 = \lambda f.\lambda x.x$, which is precisely the Church numeral 0. In the same way, a representation $s = \lambda\nu.\lambda f.\lambda x.(f)((\nu)f)x$ of the constructor $\underline{s}$ in lambda-calculus is given by a proof of $\forall x[Nx \rightarrow N\underline{s}x]$ (s is precisely a term for the successor function on the Church numerals). Terms **it** and **rec** for the construction of proofs by induction are given by proofs of $X\underline{0}, \forall y[Xy \rightarrow X\underline{s}y], Nx \vdash Xx$ and $X\underline{0}, \forall y[Ny, Xy \rightarrow X\underline{s}y], Nx \vdash Xx$ respectively.

Due to the fact that the Church numeral $(\underline{s}^n\ 0)$ is the unique term of type $N\underline{s}^n\underline{0}$ (up to $\beta\eta$–equivalence) we can construct programs satisfying equational specifications using the following method (see [Le83], [KP87]). In order to program a function between data types (say the addition from $N \times N$ to N), we have to

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

(b) define $\underline{ad}$ on N by a set E of equations (for instance $\underline{ad}[x,\underline{0}] = x$, $\underline{ad}[x,\underline{s}y] = \underline{s}[\underline{ad}[x,y]]$).

(c) deduce the theorem $\forall x\forall y[Nx, Ny \rightarrow N\underline{ad}[x,y]]$ from the previous set of equations.

The term given by the proof of $\forall x\forall y[Nx, Ny \rightarrow N\underline{ad}[x,y]]$ is a program for addition. This programming method extends to all functions defined by equations on inductive data types. The correctness of the programs is ensured by a general result (see [KP87]).

Well known metamathematical results say that we can obtain in this way programs for all the recursive functions whose termination is provable in second order Peano arithmetic ([Gi72], [FLO83]), and thus in practice, programs for all the total functions we need. But this doesn't mean that we can obtain all the programs we need. Let us develop some crucial examples.

Addition.

For the addition we have to prove the theorem $\forall x \forall y[Nx, Ny \to N\underline{ad}[x,y]]$. Here is a natural proof, using the proof of $\forall x[Nx \to N\underline{s}x]$:

Let $\nu : Nx$, $\mu : Ny$. We prove $N\underline{ad}[x,y]$ by induction on y.

(1) We have $\nu : Nx$ and by the first equation, $\nu : N\underline{ad}[x,\underline{0}]$.

(2) We look for a term of type $\forall z[N\underline{ad}[x,z] \to N\underline{ad}[x,\underline{s}z]]$. By the second equation it suffices to find a term of type $\forall z[N\underline{ad}[x,z] \to N\underline{s}[\underline{ad}[x,z]]]$, and s is precisely a term of this type.

Finally $\lambda\nu.\lambda\mu.((\mu)s)\nu$ is of type $\forall x \forall y[Nx, Ny \to N\underline{ad}[x,y]]$ and thus computes the addition.

The previous term computes the addition of n and m in a number of (reduction) steps proportional to m. But there is a more "direct" proof which gives a "one–step" addition.

Let $\nu : Nx$, $\mu : Ny$, $f : \forall y[Xy \to X\underline{s}y]$, and $a : X\underline{0}$; we prove $X\underline{ad}[x,y]$ by induction on y.

(1) Clearly $((\nu)f)a$ is of type Xx and thus of type $X\underline{ad}[x,\underline{0}]$ (by the first equation).

(2) By R5 and R4, f is of type $\forall z[X\underline{ad}[x,z] \to X\underline{s}[\underline{ad}[x,z]]]$ and by the second equation of type $\forall z[X\underline{ad}[x,z] \to X\underline{ad}[x,\underline{s}z]]$.

Therefore $\lambda\nu.\lambda\mu.\lambda f.\lambda a.((\mu)f)((\nu)f)a$ is of type $\forall x \forall y[Nx, Ny \to N\underline{ad}[x,y]]$ and thus a program for the addition.

The difference between the two proofs is that in the second proof the reasoning for the induction step is purely conceptual (we use just rules for quantifiers) while the first one calls on a previous theorem whose proof has an algorithmic content (the program for the successor). But for the second one we need second order logic. The use of quantifiers allows to write clever programs by proofs which are not directly accessible on the equational specification – say by rewriting.

Predecessor.

We define the predecessor function $\underline{p}$ by the equations $\underline{p0} = \underline{0}$ and $\underline{ps}x = x$, and prove the theorem $\forall x[Nx \to N\underline{p}x]$. The natural proof by induction makes use of a complicated induction hypothesis: instead of proving directly $N\underline{p}x$ we prove $Nx \wedge N\underline{p}x$. The algorithmic counterpart is that the resulting program needs at least n steps to compute the predecessor of n. In fact 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.

Theorem 1 Let $c \in \mathbb{N}$. There is no term of lambda–calculus computing the predecessor of each iterative natural number in less than c steps.

We prove this theorem in §2 (P. Malaria proved independently this result [Mal]). In fact we can prove a better result, using a different proof: all programs for the predecessor require a least a number of steps linear in n to compute the predecessor of n. From a practical point of view this means that each time we have to iterate programs using the predecessor (and more generally the destructors of a data type, like cdr for lists) we will get non efficient programs (for instance, if we iterate n times the predecessor function on n, we will get 0, but only after n^2 steps, and not n steps – as expected). This situation appears frequently because the operator **rec** is in fact based on the predecessor: in the construction of **rec** we need either an explicit use of the predecessor or an induction hypothesis of the form $\mathbf{N}x \wedge Xx$ which has the same algorithmic effect.

Therefore if we want to maintain the possibility of iterating functions, we have to use other representations of the data.

2.3 Recursive representation of data.

We recall the recursive definition of natural numbers of [Pa88a]. We introduce a unary predicate constant $\mathbf{N}^r$ and two constructors $\underline{0}$ (for zero) and $\underline{\sigma}$ (for the successor function). We define the recursive type of natural numbers $\mathbf{N}^r$ as the least fixed point of

$$\Phi[\mathbf{N}^r,x] := \forall X[\forall y[\mathbf{N}^r y \to X\underline{\sigma}y], X\underline{0} \to Xx].$$

Formally $\mathbf{N}^r$ is defined by the rules:

$$\Phi[\mathbf{N}^r,x] \subseteq \mathbf{N}^r x \quad \mu_1$$

$$\mathbf{N}^r x \subseteq \Phi[\mathbf{N}^r,x] \quad \mu_1'$$

$$\frac{\Phi[K,x] \subseteq Kx}{\mathbf{N}^r x \subseteq Kx}\ \mu_2$$

where $\subseteq$ is governed by the rule:

$$\frac{\Gamma \vdash_e t : A \qquad A \subseteq B}{\Gamma \vdash_e t : B}\ \subseteq$$

This definition is a particular case of an extension of the system of §2.1 to formulas with fixed point constructor: if A[X] is a formula where all the free occurences of X are positive, then we can form the formula $\mu XA[X]$; the rules μ_1,μ_1', μ_2 extend in an obvious way to this system. Because of the positivity condition, the rules are sound and the strong normalization theorem easely extends to this system.

The representation of the constructors of the type in lambda-calculus are obtained from the definition of the type in the same way as before.

– The representation $0 = \lambda f.\lambda a.a$ of $\underline{0}$ is given by a proof of $\mathbf{N}^r\underline{0}$.

– The representation $\sigma = \lambda\nu.\lambda f.\lambda a.(f)\nu$ of $\underline{\sigma}$ is given by a proof of $\forall x[\mathbf{N}^r x \to \mathbf{N}^r\underline{\sigma}x]$.

We can also obtain an iteration operator and a recursion operator: the recursion operator is $\mathbf{rec}^{\mathbf{r}}[\alpha,\beta,n] = (((n)\rho)\iota)\rho$, with $\iota = \lambda d.\alpha$ and $\rho = \lambda y.\lambda r.((\beta)y)(((y)r)\iota)r$.
Using these operators we can program on the recursive representation of natural numbers (which we call stacks) exactly in the same way as for the iterative one (other ways of programming using rules for the fixed point operator of lambda–calculus are given in [Pa88b]). We will now examine the algorithmic properties of the recursive representation on our two canonical examples, the predecessor and the addition.

Predecessor.

The recursive definition of $\mathbf{N}^{\mathbf{r}}x$ allows to construct by a proof a "one–step" predecessor. We introduce a new function constant $\underline{\pi}$ defined by the equations:

$$\underline{\pi}\underline{0} = \underline{0} \text{ and } \underline{\pi}\underline{\sigma}x = x$$

We prove $\forall x[\mathbf{N}^{\mathbf{r}}x \to \mathbf{N}^{\mathbf{r}}\underline{\pi}x]$. Let $\nu : \mathbf{N}^{\mathbf{r}}x$; by μ_1' we have $\nu : \forall X[\forall y[\mathbf{N}^{\mathbf{r}}y \to X\underline{\sigma}y], X\underline{0} \to Xx]$ and therefore $\nu : \forall y[\mathbf{N}^{\mathbf{r}}y \to \mathbf{N}^{\mathbf{r}}\underline{\pi}\underline{\sigma}y], \mathbf{N}^{\mathbf{r}}\underline{\pi}\underline{0} \to \mathbf{N}^{\mathbf{r}}\underline{\pi}x$; by the equations it follows $\nu : \forall y[\mathbf{N}^{\mathbf{r}}y \to \mathbf{N}^{\mathbf{r}}y], \mathbf{N}^{\mathbf{r}}\underline{0} \to \mathbf{N}^{\mathbf{r}}\underline{\pi}x$.
Clearly $\lambda x.x : \forall y[\mathbf{N}^{\mathbf{r}}y \to \mathbf{N}^{\mathbf{r}}y]$ and $0 : \mathbf{N}^{\mathbf{r}}\underline{0}$; therefore $\lambda\nu.((\nu)\lambda x.x)0 : \forall x[\mathbf{N}^{\mathbf{r}}x \to \mathbf{N}^{\mathbf{r}}\underline{\pi}x]$.

The existence of programs computing the destructors of the type in one step, allows to construct programs with better operational properties. This is the case for programs based on the recursion operator. For instance, consider the program comparing two natural numbers. We introduce a function constant comp defined by the following set of equations:

$$\underline{\text{comp}}[\underline{0},y] = \underline{0}$$
$$\underline{\text{comp}}[\underline{\sigma}x,\underline{0}] = \underline{\sigma}\underline{0}$$
$$\underline{\text{comp}}[\underline{\sigma}x,\underline{\sigma}y] = \underline{\text{comp}}[x,y]$$

We prove $\forall x\forall y[\mathbf{N}^{\mathbf{r}}x, \mathbf{N}^{\mathbf{r}}y \to \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[x,y]]$. Let $n : \mathbf{N}^{\mathbf{r}}x$; we prove $X[x] = \forall y[\mathbf{N}^{\mathbf{r}}y \to \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[x,y]]$ by induction on x.
(1) We have $\lambda x.x : X[\underline{0}]$ by the first equation.
(2) We look for $G : \mathbf{N}^{\mathbf{r}}z, X[z] \to X[\underline{\sigma}z]$.
Let $u : \mathbf{N}^{\mathbf{r}}z$, $\beta : X[z]$, $m : \mathbf{N}^{\mathbf{r}}y$; we prove $\mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[\underline{\sigma}z,y]$ by induction on y.
(2.1) We have $(\sigma)0 : \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[\underline{\sigma}z,\underline{0}]$ by the second equation.
(2.2) We look for $H : \mathbf{N}^{\mathbf{r}}t, \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[\underline{\sigma}z,t] \to \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[\underline{\sigma}z,\underline{\sigma}t]$.
Let $\alpha : \mathbf{N}^{\mathbf{r}}t$ and $d : \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[\underline{\sigma}z,t]$; we have $\beta : \mathbf{N}^{\mathbf{r}}t \to \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[z,t]$ and $(\beta)\alpha : \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[z,t]$; by the third equation, $(\beta)\alpha : \mathbf{N}^{\mathbf{r}}\underline{\text{comp}}[\underline{\sigma}z,\underline{\sigma}t]$ and we can take $H = \lambda\alpha.\lambda d.(\beta)\alpha$.
We take $G = \lambda u.\lambda\beta.\lambda m.\mathbf{rec}^{\mathbf{r}}[(\sigma)0,H,m]$.
Finally we get $\text{comp} = \lambda n.\mathbf{rec}^{\mathbf{r}}[\lambda x.x,G,n]$.
The program comp has the intended behaviour, in the sense that it decrements alternatively each argument, and compare m and n in inf(m,n) steps.
Because all the specifications of the program are described at the logical level, we can obtain a

program for the comparison on the iterative type of natural numbers just by replacing in comp σ by s and $\mathbf{rec}^r$ by rec. But the program obtained on the iterators doesn't compare m and n inf(m,n) steps! (this is because rec "contains" the predecessor)

Addition.

From the two proofs we have in the iterative case, only the "bad" one can be reproduced in the recursive case (because the "good" one uses the particular structure of the formula). In fact we cannot find a better proof, and even in pure lambda-calculus no one–step addition exists on the recursive representation of natural numbers.

Theorem 2 Let $c \in \mathbb{N}$. There is no term of lambda–calculus computing in less than c steps the addition of n and m, for all recursive natural numbers n, m.

This theorem his proved in §4. It completes the nice duality between the iterative and recursive representation of natural numbers. Note that there are terms allowing a translation between the two, and therefore we can use them together for programming.

§3 PROOF OF THEOREM 1

For the proofs of theorem 1 and 2, the terms of lambda–calculus will be considered as particular words on the alphabet $\{(,),\lambda,.\} \cup V$ where V is a countable set of variables. We say that u is a factor of the word v and write $u \prec v$ if v can be written w_1uw_2 for some words w_1 and w_2. If $u,v \prec w$, we say that u meets v (in w) if there exists a non empty factor w'of w such that $u = u'w'$ and $v = w'v'$, or $v = v'w'$ and $u = w'u'$, and $u'w'v'$ is a factor of w.

We denote the normal form of the iterator n by $\overline{n}$. Let $L = \{(x); x \in V\}$. An element of L^* is called a witness. A witness of a term t is a factor α of t such that $\alpha \in L^*$. The length $|\alpha|$ of a witness α is the natural number n such that $\alpha \in L^n$. A witness α is uniform if $\alpha \in \{(x)\}^*$ for a certain $x \in V$. Note that the iterator $\overline{n}$ contains an uniform witness of length n (in fact $\overline{n} = \lambda f.\lambda x.\alpha x$ with $f,x \in V$ and $\alpha \in \{(f)\}^n$). The idea of the proof will be to show that we cannot directly obtain (using reduction) an uniform witness of length n from an uniform witness of length n + 1, but we have to destroy it first and then construct another one, step by step. Here is the canonical example of the construction of a new witness: if $\alpha_1,\alpha_2,\alpha_3$ are witnesses, then the reduction of $w_1\alpha_1(\lambda x.\alpha_2 x)\alpha_3 w_2$ creates a new witnes $\alpha_2\alpha_1\alpha_3$.

Let $t \in \Lambda$ and c be a natural number greater than the maximal length of a witness of t. We consider a reduction sequence (t_i) starting from $(t)\overline{n}$, i.e.

$t_0 = (t)\overline{n}$

$t_i \triangleright^1 t_{i+1}$

where $\triangleright^1$ denotes the one–step reduction.

Proposition Let α be a witness of t_i.
If $|\alpha| > c.3^i$, then α meets a uniform witness of t_i of length $\geq n$.

Theorem 1 is an immediate consequence of this proposition. Let $k \in \mathbb{N}$, $t \in \Lambda$ and c be a natural number greater than the maximal length of a witness of t. We prove that t doesn't compute the predecessor of $\overline{n+1}$ in less than k steps, for each $n > 3^k$. We consider a reduction sequence (t_i) starting from $(t)\overline{n+1}$. By the previous proposition, for each $l \leq k$, if α is a witness of t_l, then either $|\alpha| \leq c.3^l < n$ or meets a witness β of length $\geq n+1$; therefore t_k is distinct from $\overline{n}$.

lemma 1. Let t be a term, u a subterm of t which is not a variable and β a witness of t which meets u. Then $\beta = \beta_1\gamma\beta_2$ with $\beta_1,\beta_2 \in L^*$, $\gamma \in L$, β_1 doesn't meet u and $\gamma\beta_2 \prec u$.
proof. Because u is not a variable, it has a terminal segment of length 2 of the form .x or)x with $x \in V$. But .x and)x cannot be factors of β (recall that β is a concatenation of words of the form (x) with $x \in V$). Let β_1 be the greatest initial segment of β such that $\beta_1 \in L^*$ and β_1 doesn't meet u. We have $\beta = \beta_1\gamma\beta_2$ with $\beta_2 \in L^*$, $\gamma \in L$, $\beta_2 \prec u$ and γ meets u; because u is a term and γ meets u, we have $\gamma \prec u$, and therefore $\gamma\beta \prec u$.

Proof of the proposition.

We call n–witness of a term t a uniform witness of t of length n. The proof proceeds by induction on i. For $i = 0$, we see that a witness of $(t)\overline{n}$ is a witness of one of the terms t or $\overline{n}$, and therefore satisfies the required condition.

Suppose now that the result holds for t_i and consider a witness α of t_{i+1} of length $> c.3^{i+1}$. We have

$t_i = w_1(\lambda x.u)vw_2$ and

$t_{i+1} = w_1u[v/x]w_2$

We have to prove that there exists a n–witness of t_{i+1} which meets α.

Fact 1 If β is a witness of length $> c.3^i$ such that $\beta \prec w_1$ or $\beta \prec w_2$, then there exists a n–witness of t_{i+1} which meets β.
Proof. By induction hypothesis there is a n–witness β' of t_i which meets β in t_i. By lemma 1, β' cannot meet $(\lambda x.u)v$, and therefore is also a witness of t_{i+1}.

If α doesn't meet u[v/x], then fact 1 gives the desired result. Therefore we may assume that α meets u[v/x].
case 1: u[v/x] is a variable y.

We have $\alpha = \alpha_1(y)\alpha_2$ with $\alpha_1,\alpha_2 \in L^*$, $\alpha_1 \prec w_1$ and $\alpha_2 \prec w_2$. Because $|\alpha| > c.3^{i+1}$, we have $|\alpha_1| > c.3^i$ or $|\alpha_2| > c.3^i$; in each case we get by fact 1 a n–witness of t_{i+1} which meets α.
case 2: u[v/x] is not a variable.

The term u[v/x] is not a variable and meets α; by lemma 1, we can write α as $\alpha_1\alpha_2$ with $\alpha_1,\alpha_2 \in L^*$, $\alpha_1 \prec w_1$ and $\alpha_2 \prec u[v/x]$. If $|\alpha_1| > c.3^i$ we get by fact 1 a n–witness of t_{i+1} which meets α. Otherwise we have $|\alpha_2| > 2c.3^i$ and the result follows from fact 2 below.

Fact 2 Let β be a witness of u[v/x]. If $|\beta| > 2c.3^i$, then there exist a n–witness of u[v/x] which meets β.

Proof. Suppose that v is a variable y. Let β' be a witness of u such that $\beta = \beta'[y/x]$. By the induction hyptothesis, there exists a n–witness θ of t_i which meets β'; because $\beta' \prec \lambda x.u$, θ meets $\lambda x.u$ and by lemma 1, $\theta \prec u$; therefore $\theta[y/x]$ is a n–witness of u[y/x] which meets β.

Suppose now that v is not a variable and write u as $w_1x_1w_2...x_nw_n$, where $x_1,...,x_n$ are all the free occurrences of x in u. Using lemma 1, we see that there exists an occurence v_j of v in u[v/x] and $\beta_1,\beta_2 \in L^*$ such that $\beta = \beta_1\beta_2$, $\beta_1 \prec w_j$ and $\beta_2 \prec v_j$.

If $|\beta_1| > c.3^i$, then there is a n–witness θ of t_i which meets β_1 and hence meets $\lambda x.u$; by lemma 1, $\theta \prec u$; because θ is uniform, it cannot meet an occurence of x (otherwise β_1 itself meets an occurence of x); it follows that θ is a n–witness of t_{i+1} which meets β.

If $|\beta_1| \leq c.3^i$, then $|\beta_2| > c.3^i$ and considering β_2 as a factor of v we get by the induction hypothesis a n–witness θ of t_i which meet β_2; in this case θ meets $(\lambda x.u)v$ and by lemma 1, $\theta \prec v$; therefore θ is a n–witness of t_{i+1} which meets β.

§4 PROOF OF THEOREM 2

We denote the normal form stack n by $\overline{n}$ and introduce a new notion of witness adapted to stacks. A witness is an element of L^*, where $L = \{\lambda f.\lambda x.(f);\ x,f \in V\}$. A witness of a term t is a factor α of t such that $\alpha \in L^*$. The length $|\alpha|$ of a witness α is the natural number n such that $\alpha \in L^n$. Note that the stack $\overline{n}$ can be written $\alpha\overline{0}$ with $\alpha \in L^n$.

Let t be a term and α a witness of t. The complement of α is the unique v of t such that the factor αv is a term (in that case v is itself a term). We say that α is closed if its complement is a stack $\overline{n}$, and open in the other case.

Let $t \in \Lambda$ and c be a natural number greater than the maximal length of a witness of t. We consider a reduction sequence (t_i) starting from $((t)\overline{n})\overline{m}$, i.e.

$t_0 = ((t)\overline{n})\overline{m}$

$t_i \triangleright^1 t_{i+1}$

Proposition Let α be a witness of t_i.

(i) If α is open, then $|\alpha| < c.3^i$.

(ii) If α is closed, then $|\alpha| < c.3^i + \sup(n,m)$.

Theorem 2 is an immediate consequence of this proposition. Let $k \in \mathbb{N}$, $t \in \Lambda$ and c be a natural number greater than the maximal length of a witness of t. We prove that t doesn't compute the addition of $\overline{n}$ and $\overline{n}$ in less than k steps, for each $n > c.3^k$. We consider a reduction sequence (t_i) starting from $((t)\overline{n})\overline{n}$. By the proposition, for each $1 \leq k$, if α is a witness of t_l, then $|\alpha| < c.3^l + n < n + n$; therefore t_k is distinct from $\overline{n+n}$.

Proof of the proposition.

The proof proceeds by induction on i. For $i = 0$, we see that a witness of $((t)\overline{n})\overline{m}$ is a witness of one of the terms t, $\overline{n}$, $\overline{m}$, and therefore satisfies the conditions (i) and (ii).

Suppose now that the result holds for t_i and consider a non empty witness α of t_{i+1}. We have

$t_i = w_1(\lambda x.u)vw_2$

$t_{i+1} = w_1u[v/x]w_2$

If α doesn't meet u[v/x], then α is a witness of t_i and the conditions (i) and (ii) hold. Therefore we may assume that α meets u[v/x].

case 1: u[v/x] is a variable y.

We have $t_{i+1} = w_1yw_2$ and y meets α. Therefore $\alpha = \alpha_1\lambda y.\lambda x.(y)\alpha_2$ with $\alpha_1,\alpha_2 \in L^*$, $\alpha_1 \prec w_1$ and $\alpha_2 \prec w_2$; in particular α_1 and α_2 are witnesses of t_i.

Clearly α_1 is open in t_i or empty and by induction hypothesis $|\alpha_1| < c.3^i$. If α_2 is open or empty we also have $|\alpha_2| < c.3^i$ and $|\alpha| < c.3^i + 1 + c.3^i \leq c.3^{i+1}$. If α_2 is closed then $|\alpha_2| < c.3^i + \sup(n,m)$; in this case α is closed and

$|\alpha| < c.3^i + 1 + c.3^i + \sup(n,m) \leq c.3^{i+1} + \sup(n,m)$.

case 2: u[v/x] is not a variable.

We first prove a lemma.

lemma 1. Let β be a witness of a term t, u a subterm of t which is not a variable and meets β. Then $\beta = \beta_1\gamma\beta_2$ with $\gamma \in L$, $\beta_1,\beta_2 \in L^*$, $\beta_2 \prec u$, β_1 doesn't meet u and γ meets u.

proof. Because u is not a variable, it has a terminal segment of length 2 of the form .x or)x with $x \in V$. But .x and)x cannot be factors of β (recall that β is a concatenation of words of the form $\lambda f.\lambda x.(f)$). It follows that β can be written $\beta = \beta_1\gamma\beta_2$ with $\gamma \in L$, $\beta_1,\beta_2 \in L^*$, $\beta_2 \prec u$, β_1 doesn't meet u and γ meets u.

By the previous lemma we write α as $\alpha_1\gamma\alpha_2$ with $\gamma \in L$, $\alpha_1,\alpha_2 \in L^*$, $\alpha_2 \prec u[v/x]$, $\alpha_1 \prec w_1$ and γ meets u[v/x]. The witness α_1 is either empty or an open witness of t_i and therefore $|\alpha_1| < c.3^i$. The result follows from the next lemma:

lemma 2. (i) if α_2 is an open witness, then $|\alpha_2| < 2c.3^i$.

(i) if α_2 is an closed witness, then $|\alpha_2| < 2c.3^i + \sup(n,m)$.

proof. Write $u = w_1x_1w_2...x_nw_n$, where $x_1,...,x_n$ are all the free occurrences of x in u. If $\alpha_2 \prec w_j$ for a certain j, then $\alpha_2 \prec t_i$ and the result follows from the induction hypothesis. Otherwise α_2 meets an

occurence v_j of v in u[v/x], and v cannot be a variable because a witness doesn't contain free occurrences of variables. By lemma 1 we have $\alpha_2 = \beta_1\theta\beta_2$ with $\theta \in L$, $\beta_1,\beta_2 \in L^*$,$\beta_1 \prec w_j$, $\beta_2 \prec v_j$, and therefore β_1 and β_2 are both witness of t_i.
The witness β_1 is either empty or an open witness of t_i, and $|\beta_1| < c.3^i$. If β_2 is empty or an open witness we also have $|\beta_2| < c.3^i$ and $|\alpha_2| < 2c.3^i$. If β_2 is closed then $|\beta_2| < c.3^i + \sup(n,m)$; in this case α_2 is closed and $|\alpha_2| < 2c.3^{i+1} + \sup(n,m)$.

5 ANOTHER REPRESENTATION OF DATA

The lack of one–step addition is certainly less serious, from a programming point of view, than the lack of a one–step predecessor. But it would be interesting to have a representation of data with both facilities. In pure lambda–calculus, a simple modification of stacks gives a solution:

$0 = \lambda x.x$

$n + 1 = \lambda x.\lambda f.(f)(n)x$

For this representation of natural numbers we have

– a one–step predecessor: $pred = \lambda n.\lambda x.((n)x)\lambda x.x$

– a one–step addition: $ad = \lambda n.\lambda m.\lambda x.(n)(m)x$

– an iteration operator (for induction): $it[\alpha,\beta,n] = (((n)\iota)\rho)\rho$ with $\rho = \lambda u.\lambda r.(\beta)((u)r)r$ and $\iota = \lambda d.\lambda d.\alpha$.

– a recursion operator: $rec[\alpha,\beta,n] = ((((n)\iota)\rho)\rho)(pred)n$ with $\rho = \lambda u.\lambda r.\lambda y.((\beta)y)((u)r)r$ and $\iota = \lambda d.\lambda d.\lambda d.\alpha$

The iteration and recursion operators can be obtained exactly in the same way as for stacks. It is to be noted that the term pred doesn't satisfy $(pred)0 = 0$; but using the operator rec we can find by a proof, in a standard way, a one–step predecessor

$pred' = \lambda n.(((n)\lambda d.\lambda d.0)\lambda d.\lambda x.x)\lambda x.((n)x)\lambda x.x$ such that $(pred')0 = 0$.

It is also possible to type this representation of natural numbers using fixed points, but the meaning of this typing is not clear – contrary to those of iterators or stacks. In fact it seems that a representation cannot be "at the same time" iterative and recursive.

This representation generalizes to all the usual data types. For instance, for the type "list" we will have:

$nil = \lambda x.x$

$cons(a,l) = \lambda x.\lambda f.((f)a)(l)x$

For example the list (a,b,c) is represented by $\lambda x.\lambda f.((f)a)\lambda f.((f)b)\lambda f.((f)c)x$.

REFERENCES

[BB85] C. BOHM, A. BERARDUCCI, **Automatic synthesis of typed Λ–programs on term algebras**, TCS 39 (1985), pp 135–154.

[FLO83] S. FORTUNE, D. LEIVANT, M. O'DONNELL, **Expressiveness of simple and second–order type structures**, J.ACM vol 30 (1983), pp 151–185.

[Gi72] J.Y. GIRARD, **Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur**, Thèse d'état, Université Paris 7, 1972.

[Kr87] J.L. KRIVINE, **Un algorithme non typable dans le systeme F**, CRAS 1987.

[KP87] J.L. KRIVINE, M. PARIGOT **Programming with proofs**, FCT 87, Berlin 1987, (to appear in EIK 1990).

[Le83] D. LEIVANT, **Reasoning about functional programs and complexity classes associated with type disciplines**, FOCS, 1983, pp 460–469.

[Ma84] P. MARTIN–LØF, **Intuitionistic type theory**, Bibliopolis, 1984.

[Mal] P. MALACARIA, personal commucation.

[Pa88a] M. PARIGOT, **Programming with proofs: a second order type theory**, ESOP'88, LNCS 300, pp 145–159.

[Pa88b] M. PARIGOT, **Recursive programming with proofs**, preprint 1988.