Call-by-name versus call-by-value in primitive recursion: storage operator

Pierre Valarcher[*]

Abstract

We study the call-by-value and call-by-name primitive recursion encoded in the framework of system T. It has been shown in [2] and [1] that call-by-name is a better strategy than the call-by-value one when using functionallity (level one of system T) : the inf algorithm can not be well encoded in call-by-value while it has a good complexity in call-by-name. We study the level 0 of system T (encoding only primitive recursion) and compare the two strategies. We establish then, that we can simulate the call-by-value strategy in the call-by-name framework using a storage operator.

1 Introduction

The set of primitive recursion functions is large enough to define most programmable functions (those which may be computed in reasonnable time complexity). If we are interested in programming languages that compute this set of functions, we know that some of them are not expressive enough to implement efficient algorithms:

- first was done by L. Colson (see [1]) that use denotational semantics (the used domain is the lazy integer [6]) and proved that though the function inf that computes the minimum of two integers is obviously a primitive recursive function (see [14] for a formal definition), there is no way to represent (in the model of primitive recursive algorithms which are called PR-combinators) the *good* algorithm, the one which decreases alternatively both arguments. He proved an *ultimate obstinacy theorem* and he showed that every PR-combinator must *choose* one (and only one) of its argument and thus the alternation between arguments is impossible. A constructive proof of this property is in [3];
- this work has been followed by R. David (see [4]) who developed a new semantics (the *trace* of computation) which allows him to prove a new property (the *backtracking property*) which is proved for every primitive recursive algorithm using any kind of data types.

[*]LIFAR (EA 2655), Université de Rouen, France. Electronic address: Pierre.Valarcher@univ-rouen.fr

- L. Colson and D. Fredholm (see [7] and [2]) show that call-by-value strategy (with primitive recursion over lists of integers and with primitive recursion in higher types, called system T of Gödel) does not allow to compute the *good* algorithm of the *inf* function.
- in [12], Y. Moschovakis established a linear lower bounds for the complexity of non-trivial primitive recursive algorithm from *piecewise linear* given functions. His main corollary is that logtime algorithms for the greatest common divisor from such givens (such as *Stein*'s) cannot matched in efficiency by primitive recursive algorithms from the same given functions. He ended by an open problem relative to the classical *Euclidan algorithm* (a partial response has been found by L. Van Den Dries, see [5]).

We consider the call-by-value and call-by-name strategies in the primitive recursive framework. Terms are encoded considering level 0 of system T of Gödel.

We try to simulate the behaviour (the reductions) of call-by-value within the call-by-name strategy using storage operator. A *storage operator* (first considered in [10]) is some term that takes as inputs a function f and its argument a and then computes first this argument (a reduces to some $S^n(0)$) and then $f \; S^n(0)$. This is the opposite of call-by-name strategy.

We will conjecture that one can simulate all call-by-value reductions of primitive recursive term by a call-by-name equivalent term that has the same complexity. We failed to obtain this result but we construct a simulation (a mapping from term to term that "regularly" maps a term in the call-by-value reduction to a term in the call-by-name reduction, the two terms computing the same function). Similarly work has been done by G. Plotkin for the λ–calculus (see [?]).

1.1 Outline of the paper

We first recall system T of Goedel with the two operational semantics: call-by-value and call-by-name. We then construct two storage operators (one in level 0 and one in level 1 of system T) and we establish their complexity. Finally, we give a mapping from terms to terms that allows us to simulate call-by-value reduction in call-by-name strategy.

2 Goedel's system T: syntax

Primitive recursive functionals of finite types, also known as Goedel's T in logic (see [8, 15]), and as typed lambda calculus with primitive recursive recursion in higher types in computer science (see [9]), are quite important in both areas. In the former they are used in proof-theoric studies of Peano arithmetic, in the latter they can be used to give the formal semantics of some modern kind (functional) computer programming languages.

2.1 Definition

The *types* of system T are given by the following inductive definition:

- $\mathbf{N}$ is a type (the type of natural numbers)

- if T and U are types, then $T \to U$ is a type

As usual, we assume an infinite set of typed variables x^T for each type T. The *typed terms* of system T are given by the following inductive definition:

- $x^T : T$.
- $0 \ : \mathbf{N}$ and $S(t) : \mathbf{N}$ if $t : \mathbf{N}$.
- If $t : T$, then $\lambda x^U t : U \to T$.
- If $t : T \to U$ and $u : T$, then $(tu) : U$.
- If $t : \mathbf{N}, b : T$ and $s : \mathbf{N} \to T \to T$, then $\mathbf{rec}_T(t, b, s) : T$.

We define by induction the *degree* $\partial(T)$ of a type T as follows:

- $\partial(\mathbf{N}) = 0$
- $\partial(T \to U) = \max(\partial(T) + 1, \partial(U))$

We now define the degree $d(t)$ of a term t as the maximum degree of types T such that $\mathbf{rec}_T$ occurs in t. We denote by T_0 the set of all terms of degree 0 (those terms define *primitive recursive functions*) and T_1 the set of all terms of degree 1.

2.2 Examples

- $\mathit{Add}_1\ x_1^N\ x_2^N = \mathbf{rec}_N(x_1^N, x_2^N, \lambda x^N \lambda y^N S(y)) : N$ is a recursive definition of addition.
- $\mathit{Add}_2\ x_1^N\ x_2^N = \mathbf{rec}_N(x_2^N, x_1^N, \lambda x^N \lambda y^N S(y)) : N$ is another recursive definition of addition.
- $\mathit{Ack}\ x_1^N\ x_2^N = \mathbf{rec}_{N \to N}(x_1^N, \lambda y^N S(y), \lambda x^N \lambda u^{N \to N} \lambda y^N w)$ where w is defined by $\mathbf{rec}_N(y, (u\ S(0)), \lambda z^N \lambda w^N (u\ w)))\ x_2)$ is a definition of the Ackerman function (which is not primitive recursive).

We now define the evaluation function that maps a program to a value if it reduces to one.

3 Two operational semantics: call-by-value and call-by-name

Substitutions are defined as usually.

3.1 Call-by-value reductions

We first define the set of *values*:

- variables are values;
- $S^k(0)$ is a value (with $k \geqslant 0$);
- $\lambda x.t$ is a value (to allow for weak reduction)

Then, we define the *call-by-value one-step-reduction* relation between terms as the smallest relation $\Rightarrow$ verifying (v, v_1, v_2 denote values):

$$(\lambda x.t \;\; v) \Rightarrow t[v/x]$$

$$\mathbf{rec}_T(0, v_1, v_2) \Rightarrow v_1$$

$$\mathbf{rec}_T(S(v), v_1, v_2) \Rightarrow (v_2 \;\; v \;\; \mathbf{rec}_T(v, v_1, v_2))$$

compatible with the context: that is if $t \Rightarrow u$ then

- $S(t) \Rightarrow S(u)$
- $(w \;\; t) \Rightarrow (w \;\; u)$
- $(t \;\; v) \Rightarrow (u \;\; v)$
- $\mathbf{rec}_T(t, b, s) \Rightarrow \mathbf{rec}_T(u, b, s)$
- $\mathbf{rec}_T(v, t, s) \Rightarrow \mathbf{rec}_T(v, u, s)$
- $\mathbf{rec}_T(v, v_1, t) \Rightarrow \mathbf{rec}_T(v, v_1, u)$

We note $\Rightarrow^*$ the transitive and reflexive closure of $\Rightarrow$ and $\Rightarrow^n$ the n^{th} power of $\Rightarrow$.

We define the reduction cost ($\mathbf{cost}_v \; (t)$) of a term t as the minimal number of *call-by-value one-step-reduction* needed to reduce t to its normal form. Recall that a term is in *normal form* if there is no more reduction possible.

3.2 Call-by-name reductions

We first define the set of *values*:

- $S(t)$ is a value for any term $t : \mathbf{N}$
- $\lambda x.t$ is a value

Then, we define the *call-by-name one-step-reduction* relation between terms as the smallest relation $\Downarrow$ verifying :

$$(\lambda x.t \;\; u) \Downarrow t[u/x]$$

$$\mathbf{rec}_T(0, b, s) \Downarrow b$$

$$\mathbf{rec}_T(S(t), b, s) \Downarrow (s \;\; t \;\; \mathbf{rec}_T(t, b, s))$$

compatible with the context: that is if $t \Downarrow u$ then

- $S(t) \Downarrow S(u)$
- $(t \;\; w) \Downarrow (u \;\; w)$
- $\mathbf{rec}_T(t, b, s) \Downarrow \mathbf{rec}_T(u, b, s)$

We denote by $\Downarrow_*$ the reflexive transitive closure of $\Downarrow$ and $\Downarrow^n$ the n^{th} power of $\Downarrow$. We define the reduction cost ($\mathbf{cost}_n\ (t)$) of a term t as the minimal number of *call-by-name one-step-reduction* needed to reduce t to its normal form. We define $\text{cost}_n^S\ (t)$ (for $t : N$) by the minimal number of *call-by-name one-step-reduction* needed to reduce t to the first term beginning by a new S ($S^k(t) \Downarrow^{\text{cost}_n^S\ (t)} S^k(S(t'))$ for $k \geq 0$) or, if $\mathbf{valueOf}(t) = 0$ we put cost_n^S $(t) = \mathbf{cost}_n(t)$.

Let $a_0, \dots, a_p$ be a finite sequence such that $a = a_0$, $a_p = 0$ and for all $i \leqslant p-1$, $a_i \Downarrow^{\mathbf{cost}_{\boldsymbol{n}}^{\boldsymbol{S}}(a_i)} S(a_{i+1})$ and $a_p \Downarrow^{\text{cost}_n\ (a_p)} 0$. Then, we have $\mathbf{cost}_n(a) = \sum_{i=0}^{i=p} \mathbf{cost}_{\boldsymbol{n}}^{\boldsymbol{S}}(a_i)$.

3.3 Example

Let *twice* be defined by $\lambda x^N.\,\text{add}_1\ x\ x\ (= \lambda x.\mathbf{rec}_N(x^N, x^N, \lambda z^N \lambda y^N.S(y)))$ that computes the twice of natural integers. Let's compute *twice(*$S^2(0)$) first with call-by-value:

$$\begin{array}{ll}
\Rightarrow & \mathbf{rec}_N(S^2(0), S^2(0), \lambda z^N \lambda y^N.S(y)) \\
\Rightarrow & (\lambda z \lambda y.S(y))\ S(0)\ \mathbf{rec}_N(S(0), S^2(0), \lambda z^N \lambda y^N.S(y)) \\
\Rightarrow & (\lambda z \lambda y.S(y))\ S(0)\ ((\lambda z \lambda y.S(y))\ 0\ \mathbf{rec}_N(0, S^2(0), \lambda z^N \lambda y^N.S(y))) \\
\Rightarrow & (\lambda z \lambda y.S(y))\ S(0)\ ((\lambda z \lambda y.S(y))\ 0\ S^2(0)) \\
\Rightarrow^2 & (\lambda z \lambda y.S(y))\ S(0)\ S(S^2(0)) \\
\Rightarrow^2 & S(S(S^2(0))
\end{array}$$

We have $\boldsymbol{cost}_v(\textit{twice}(S^2(0))) = 8$ (in fact $3n+2$ for $S^n(0)$).

And second in call-by-name twice($S^2(0)$):

$$\begin{array}{ll}
\Downarrow & \mathbf{rec}_N(S^2(0), S^2(0), \lambda z^N \lambda y^N.S(y)) \\
\Downarrow & (\lambda z \lambda y.S(y))\ S(0)\ \mathbf{rec}_N(S(0), S^2(0), \lambda z^N \lambda y^N.S(y)) \\
\Downarrow^2 & S(\mathbf{rec}_N(S(0), S^2(0), \lambda z^N \lambda y^N.S(y))) \\
\Downarrow & S(\lambda z \lambda y.S(y))\ 0\ \mathbf{rec}_N(0, S^2(0), \lambda z^N \lambda y^N.S(y))) \\
\Downarrow^2 & S(S(\mathbf{rec}_N(0, S^2(0), \lambda z^N \lambda y^N.S(y)))) \\
\Downarrow & S(S(S^2(0)))
\end{array}$$

and we have again $\boldsymbol{cost}_n(\textit{twice}(S^2(0))) = 8$ (also $3n+2$).

But we can in the following term:

$$\begin{array}{rll}
\text{twice}(\text{twice}(S^2(0))) & \Rightarrow^8 & \text{twice}(S^4(0)) \\
 & \Rightarrow^{14} & S^8(0)
\end{array}$$

and $\boldsymbol{cost}_v(\textit{twice(twice}(S^2(0)))) = 22$ (in fact, $\boldsymbol{cost}_v(\textit{twice}^2(S^n(0)))$=$9n+4$) but

$\text{twice}(\text{twice}(S^2(0)))$

$$
\begin{array}{ll}
\Downarrow & \mathbf{rec}_N(\text{twice}(S^2(0)), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y)) \\
\Downarrow & \mathbf{rec}_N(\mathbf{rec}_N(S^2(0), S^2(0), \lambda z^N \lambda y^N.S(y))), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y)) \\
\Downarrow^3 & \mathbf{rec}_N(S(\mathbf{rec}_N(S(0), S^2(0), \lambda z^N \lambda y^N.S(y))), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y)) \\
\Downarrow & (\lambda z^N \lambda y^N.S(y))\mathbf{rec}_N(S(0), S^2(0), \lambda z^N \lambda y^N.S(y)) \\
 & \quad \mathbf{rec}_N(\mathbf{rec}_N(S(0), S^2(0), \lambda z^N \lambda y^N.S(y)), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y)) \\
\Downarrow^2 & S(\mathbf{rec}_N(\mathbf{rec}_N(S(0), S^2(0), \lambda z^N \lambda y^N.S(y)), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y))) \\
\Downarrow^3 & S(\mathbf{rec}_N(S(\mathbf{rec}_N(0, S^2(0), \lambda z^N \lambda y^N.S(y))), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y))) \\
\Downarrow^3 & S(S(\mathbf{rec}_N(\mathbf{rec}_N(0, S^2(0), \lambda z^N \lambda y^N.S(y)), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y)))) \\
\Downarrow & S(S(\mathbf{rec}_N(S^2(0), \text{twice}(S^2(0)), \lambda z^N \lambda y^N.S(y)))) \\
\Downarrow^7 & S(S(S(S(\text{twice}(S^2(0)))))) \\
\Downarrow^8 & S(S(S(S(S^4(0)))))
\end{array}
$$

and we have $\boldsymbol{cost}_n(\textit{twice(twice(}S^2(0)))$=30 (we have $\boldsymbol{cost}_n(\textit{twice}^2(S^n(0))$=$12n+6$).

3.4 Known results

Definition 1 *We say that a term t is* ***strongly normalisable*** *iff all sequences of reduction starting by t are* ***finite****.*

Theorem 1 *All terms of system T are strongly normalisable.*

Theorem 2 *There is no term in T_0 that computes (**by name** or **by value**) the minimum of two natural numbers in O(min).*

Theorem 3 *There is no term in T_1 that computes (**by value**) the minimum of two natural numbers in O(min).*

We can find proofs in [1] and [2]. But one can construct a T_1 term that has the good complexity in the call-by-name strategy. We show that in the framework of imperative programming same behaviours occur (see [16] and [13]).

4 Storage Operator

If $a : N$ reduces to the normal form $S^n(0)$, we define $\mathbf{valueOf}(a) = n$. The notion of storage operator in λ–calculus was introduced by J.L. Krivine in [10] [11]. It is a term that computes first arguments of term application in call-by-name strategy:

Definition 2 *A* ***storage operator*** *$M : (N \to T) \to (N \to T)$ is a term such that for all term $f : N \to T$ and all term $a : N$ if **valueOf**(a) $= n$ then $M\, f\, a \Downarrow^* f\, S^n(0)$.*

Proposition 1 *Storage operator exists in system T_0.*

Proof Let M_0 be defined by (in equational notation first):

$$\begin{array}{rcl}
\mathrm{map}(f,0) & = & 0 \\
\mathrm{map}(f,S(n)) & = & f(n) \\
\mathrm{mk}(0) & = & S(0) \\
\mathrm{mk}(S(n)) & = & \mathrm{incr}(\mathrm{mk}(n)) \\
\mathrm{incr}(0) & = & S(0) \\
\mathrm{incr}(S(n)) & = & S(S(n)) \\
M_0 & = & \lambda f \lambda a.\, \mathrm{map}(f,\mathrm{mk}(a))
\end{array}$$

In system T this give us:

$$M_0 \equiv \lambda f \lambda a.\mathbf{rec}_N(\mathbf{rec}_N(a,S(0),\lambda x\lambda y.\mathbf{incr}(y)),0,\lambda x_2\lambda y_2.(f\ x_2))$$

with $\mathbf{incr}(y) = \mathbf{rec}_N(y,S(0),\lambda x_1\lambda y_1.S(S(x_1)))$.

We show first that, if $\mathbf{valueOf}(a) = n$ then:

$$\mathbf{rec}_N(a,S(0),\lambda x\lambda y.\mathbf{incr}(y)) \Downarrow^{\mathrm{cost}_n(a)+6n+1} S^{n+1}(0)$$

1. if $\mathbf{valueOf}(a) = 0$ then

$$\mathbf{rec}_N(a,S(0),\lambda x\lambda y.\mathbf{incr}(y)) \Downarrow^{\mathrm{cost}_n(a)} \mathbf{rec}_N(0,S(0),\lambda x\lambda y.\mathbf{incr}(y)) \Downarrow S(0)$$

2. If $\mathbf{valueOf}(b) = n$ and $a \Downarrow^{\mathrm{cost}_n^S(a)} S(b)$, then $\mathbf{rec}_N(a,S(0),\lambda x\lambda y.\mathbf{incr}(y))$

$$\begin{array}{ll}
\Downarrow^{\mathrm{cost}_n^S(a)} & \mathbf{rec}_N(S(b),S(0),\lambda x\lambda y.\mathbf{incr}(y)) \\
\Downarrow^{3} & \mathbf{incr}(\mathbf{rec}_N(b,S(0),\lambda x\lambda y.\mathbf{incr}(y))) \\
\Downarrow^{\mathrm{cost}_n(b)+6n+1} & \mathbf{incr}(S^{n+1}(0)) \text{ by HR} \\
\Downarrow^{3} & S(S(S^n(0)))
\end{array}$$

And finally, we can prove that M_0 is a storage operator: $M_0 f a$

$$\begin{array}{ll}
\Downarrow^{2} & \mathbf{rec}_N(\mathbf{rec}_N(a,S(0),\lambda x\lambda y.\mathbf{incr}(y)),0,\lambda x_2\lambda y_2.(f x_2)) \\
\Downarrow^{\mathrm{cost}_n(a)+6n+1} & \mathbf{rec}_N(S^{n+1}(0),0,\lambda x_2\lambda y_2.(f x_2)) \\
\Downarrow & \lambda x_2\lambda y_2.(f x_2)S^n(0)\mathbf{rec}_N(S^n(0),0,\lambda x_2\lambda y_2.(f x_2)) \\
\Downarrow^{2} & f S^n(0)
\end{array}$$

□

Corollary 1 *The storage operator M_0 is in T_0 and if* $\mathbf{valueOf}(a) = n$ *then*

$$M_0 f a \Downarrow^{\mathrm{cost}_n(a)+6n+6} f S^n(0)$$

One can reduce the complexity of the reduction using one level of system T.

Proposition 2 *There exists a strorage operator in T_1.*

Proof Let $M_1 \in T_1$ defined by (in equational definition first):

$$\begin{array}{rcl} M_1 & = & \lambda f \lambda a. M_1'(f, a, 0) \\ M_1'(f, 0, p) & = & f(p) \\ M_1'(f, S(n), p) & = & M_1'(f, n, S(p)) \end{array}$$

Which, in T_1, gives[1]:

$$M_1 \equiv \lambda f \lambda a.(\mathbf{rec}_{N \to N}(a, \lambda z.fz, \lambda x \lambda y \lambda z.y\ S(z))\ 0)$$

Let $M_1' = \lambda f \lambda a.\mathbf{rec}_{N \to N}(a, \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))$ then we prove that, if

$$\mathbf{valueOf}(a) = n$$

then

$$\text{for all } p\ \mathbf{rec}_{N \to N}(a, \lambda z.fz, \lambda x \lambda y \lambda z.y\ S(z))\ p \Downarrow^{\text{cost}_n(a)+4n+2} f\ S^n(p)$$

by induction on $\mathbf{valueOf}(a)$.

1. If $\mathbf{valueOf}(a) = 0$ then $\mathbf{rec}_{N \to N}(a, \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))\ p$

$$\begin{array}{ll} \Downarrow^{\text{cost}_n(a)} & \mathbf{rec}_{N \to N}(0, \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))\ p \\ \Downarrow & (\lambda z.f\ z)\ p \\ \Downarrow & f\ p \end{array}$$

2. if $\mathbf{valueOf}(b) = n$ and $a \Downarrow^{\text{cost}_n^S(a)} S(b)$ with

$$\mathbf{rec}_{N \to N}(b, \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))\ q \Downarrow^{\text{cost}_n(b)+4n+2} S^n(q)$$

then $\mathbf{rec}_{N \to N}(a, \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))\ p$

$$\begin{array}{cl} \Downarrow^{\text{cost}_n^S(a)} & \mathbf{rec}_{N \to N}(S(b), \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))\ p \\ \Downarrow & (\lambda x \lambda y \lambda z.y\ S(z))\ b\mathbf{rec}_{N \to N}(b, \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))\ p \\ \Downarrow^3 & \mathbf{rec}_{N \to N}(b, \lambda z.f\ z, \lambda x \lambda y \lambda z.y\ S(z))\ S(p) \\ \Downarrow^{\text{cost}_n(b)+4n+2} & fS^n(S(p)) \text{ by HR} \\ \equiv & fS^{n+1}(p) \end{array}$$

And since $M_1\ f\ a = M_1'\ f\ a\ 0$ we get $M_1\ f\ a \Downarrow^{\text{cost}_n(a)+4n+4} fS^n(0)$. □

Corollary 2 *The storage operator M_1 is in T_1 and if* $\mathbf{valueOf}(a) = n$ *then*

$$M_1 f a \Downarrow^{\text{cost}_n(a)+4n+4} fS^n(0)$$

[1]In fact, we can translate a primitive recursive definition with variable parameters (PRV for short, [14]) in a term representable in T_1. Let f be defined by:

$$\begin{array}{rcl} f(0, y) & = & g(y) \\ f(S(n), y) & = & h(n, f(n, j(n, y)), y) \end{array}$$

with g, h and j in PRV. Then f is computable by the term: $\hat{f} = \mathbf{rec}_{N \to N}(n, \lambda z.g\ z, \lambda x \lambda y \lambda z.h\ x\ (y\ (j\ x\ z))\ z$.

Remark 1 It is an open problem to find a *better* storage operator in this framework.

Example 1 Let us recall the twice term, then we have:

$$\lambda x.\,\text{twice}(\text{twice}(S^p(0)) \Rightarrow^{3p+2} \lambda x.\,\text{twice}(S^{2p}(0))$$

$$\Rightarrow^{6p+2+1} S^{4p}(0)(\text{which gives } 9p+5 \text{ steps})$$

and

$$M_1 \lambda x.\,\text{twice}(\text{twice}(S^p(0))) \Downarrow^{3p+2+8p+4} \lambda x.\,\text{twice}\, S^{2p}(0)$$
$$\Downarrow^{6p+2+1} S^{4p}(0)(17p+9 \text{ steps})$$

Notice that

$$\lambda x.\,\text{twice}(\text{twice}(S^p(0)) \Downarrow^{12p+6} S^{4p}(0)$$

As we can note, our storage operator doesn't allow us to simulate call-by-value without loss of complexity in the call-by-name framework. That is a challenge to find a map ϕ from term to term such that

$$\mathbf{cost}_{\boldsymbol{n}}(\phi(t)) \in O(\mathbf{cost}_{\boldsymbol{v}}(t))$$

Despite this, we show a translation from term to term that allows us to simulate the call-by-value reduction in the call-by-name one in T_0 using this storage operator (Primitive recursion framework).

5 Application : Simulating cbv Primitive Recursion

We say that a term t' *simulates* another term t if sufficiently enough the reduces of t "match" reduces of t'. More formally

Definition 3 *Let $\to_a$ and $\to_b$ be two distincts reductions (strategies), let t and t' be two closed terms and ϕ a mapping from term to term. We say that t'* ***simulates*** *t if there exists $t_1, \ldots, t_n$ such that $t = t_1 \to_a^* t_2 \to_a^* \ldots \to_a^* t_n =$ $\mathbf{valueOf}(t)$ then there exists $u_1, \ldots, u_n$ such that if $t' = u_1 \to_b^* u_2 \to_b^* \ldots \to_b^*$ $u_n = \mathbf{valueOf}(t)$ and $u_i = \phi(t_i)$ (we say then that u_i ϕ-matches t_i).*

Remark 2 Two terms that are extensionally equals simulate one each other. More is the size of the set of matches terms more the simulation is fine. (See [16] for a step-lock simulation of call-by-value LOOP programming and call-by-value T).

From now on, we try to translate a term t in T_0 into another term t' (in T_0 or T_1) such that t' evaluated by name has the similar behaviour than term t evaluated by value. We define the * translation from β-normal form term (of type $\mathbf{N}$) to term by induction:

$$\begin{aligned}
x_i^* &= x_i \\
0^* &= 0 \\
S(t)^* &= S(t^*) \\
\mathbf{rec}_{\boldsymbol{N}}(n, b, \lambda y \lambda z.s)^* &= M_0(\lambda n.M_0(\lambda b.\mathbf{rec}_{\boldsymbol{N}}(n, b, \lambda y.M_0(\lambda z.s^*)))b^*)n^*
\end{aligned}$$

Theorem 4 *If $t : N$ is a closed β-normal term in T_0 then t^* is in T_0 and t^* by-name simulates t by-value.*

Proof by induction on t:

$t = x_i$ it's OK by definition of *.

$t = S(t')$ idem.

$t = \mathrm{rec}_N(n, b, \lambda y \lambda z.s)$ Recall that $\bar{n} = \mathbf{valueOf}(n)$,

$$\begin{array}{rcl}
\mathbf{rec}_{\boldsymbol{N}}(n, b, \lambda y\lambda z.s) & \Rightarrow^* & \mathbf{rec}_{\boldsymbol{N}}(\bar{n}, \bar{b}, \lambda y\lambda z.s) \\
& \Rightarrow & \lambda y\lambda z.s(\overline{n-1})\mathbf{rec}_{\boldsymbol{N}}(\overline{n-1}, \bar{b}, \lambda y\lambda z.s) \\
& \Rightarrow^* & \lambda y\lambda z.s(\overline{n-1})\bar{m} \\
& \Rightarrow^2 & s[y \leftarrow \overline{n-1}, z \leftarrow \bar{m}]
\end{array}$$

and

$t^* = M_0(\lambda n.M_0(\lambda b.\mathbf{rec}_{\boldsymbol{N}}(n, b, \lambda y.M_0(\lambda z.s^*)))b^*)n^*$

$$\begin{array}{ll}
\Downarrow^* & M_0(\lambda b.\mathbf{rec}_{\boldsymbol{N}}(\bar{n}, b, \lambda y.M(\lambda z.s^*)))b^* \text{ by HR} \\
\Downarrow^* & \mathbf{rec}_{\boldsymbol{N}}(\bar{n}, \bar{b}, \lambda y.M_0(\lambda z.s^*)) \\
\Downarrow^* & M_0(\lambda z.s^*[y \leftarrow \overline{n-1}])\mathbf{rec}_{\boldsymbol{N}}(\overline{n-1}, \bar{b}, \lambda y.M_0(\lambda z.s^*)) \\
\Downarrow^* & \lambda z.s^*[y \leftarrow \overline{n-1}]\bar{m} \\
\Downarrow^* & s^*[y \leftarrow \overline{n-1}, z \leftarrow \bar{m}]
\end{array}$$

□

We can establish a complexity result, that indicates the problem with the * mapping.

Definition 4 *We define the mapping $^\circ$ from term to term by induction on t:*

$$\begin{array}{rcl}
0^\circ & = & 0 \\
x_i^\circ & = & x_i \\
S(t)^\circ & = & S(t^\circ) \\
\mathbf{rec}_{\boldsymbol{N}}(n, b, \lambda y\lambda z.s)^\circ & = & \mathbf{rec}_{\boldsymbol{N}}(n^\circ, b^\circ, \lambda y.\lambda z.R\ s^\circ))
\end{array}$$

The term R has the effect to reconstruct a natural: $\mathbf{rec}_{\boldsymbol{N}}(n, 0, \lambda y\lambda z.S(z))$. If n is in normal form then $\mathbf{cost}_{\boldsymbol{v}}(R\ n) \in O(n)$. Then we have:

Theorem 5 *Let $t : N$ a closed β-normal term in T_0, then t^* by-name simulates t° by-value and $\mathbf{cost}_{\boldsymbol{n}}(t^*) \in O(\mathbf{cost}_{\boldsymbol{v}}(t^\circ))$.*

6 Conclusion and future works

We construct a storage operator in the framework of primitive recursion (represented by level 0 terms of Gödel system T) and we show that call-by-value strategy may be simulated by call-by-name strategy using this storage operator. Unfortunatly, we failed to construct a simulation that preserve complexity.

It is yet an open problem to establish if call-by-name strategy is a better strategy or not in the framework of primitive recursion or if the two stategies are incompatible.

References

[1] L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83, pp57-69, 1991.

[2] L. Colson and D. Fredholm. System t, call-by-value and the minimum problem. *Theoretical Computer Science*, 206, 1998.

[3] T. Coquand. Une preuve directe du théclvorème d'ultime obstination. *Compte Rendus de l'Académie des Sc.*, 314, Serie I, 1992.

[4] R. David. On the asymptotic behaviour of primitive recursive algorithms. *Theor. Comput. Sci.*, 266(1-2):159–193, 2001.

[5] L. Van Den Dries. Generating the greatest common divisor, and limitations of primitive recursive algorithms. *to appear in Foundations of Computational Mathematics*, 2003.

[6] Martin Hotzel Escardo. On lazy natural numbers with applications. *SIGACT News*, 24(1), 1993.

[7] D. Fredholm. Computing minimum with primitive recursion over lists. *Theoretical Computer Science*, 163, 1996.

[8] K. Gödel. On a hitherto unexploited extension of the finitary standpoint. *J. Philos. Logic*, 9, 1980.

[9] P. Taylor J.Y. Girard, Y. Lafont. *Proofs and Types*, volume 7. Cambridge Tracts in Theorical Comp. Sci., 1989.

[10] Jean-Louis Krivine. A general storage theorem for integers in call-by-name lambda-calculus. *Theoretical Computer Science*, 129(1):79–94, 1994.

[11] Jean-Louis Krivine. About classical logic and imperative programming. *Annals of Mathematics and Artificial Intelligence*, 16:405–414, 1996.

[12] Yiannis N. Moschovakis. On primitive recursive algorithms and the greatest common divisor function. *Theor. Comput. Sci.*, 301(1-3):1–30, 2003.

[13] B. Patrou P. Andary and P. Valarcher. About implementation of primitive recursive algorithms. *in ASM 2005 workshop*, 2005.

[14] Roza Peter. *Recursive Functions.* Academic Press, 1968.

[15] G. Plotkin. Call-by-name, call-by-value and the lambda–calculus. *TCS*, 1975.

[16] K. Schütte. *Proof theory.* Addison Wesley, 1967.

[17] S. Lacas T. Crolard and P. Valarcher. On the expressive power of FOR-language. *to be submitted to Science of Computer Programming.*