Nordic Journal of Computing

ON THE EXPRESSIVE POWER OF THE LOOP LANGUAGE

TRISTAN CROLARD
LACL – University of Paris 12
61, avenue du Général de Gaulle
94010 Créteil Cedex, France
crolard@univ-paris12.fr

SAMUEL LACAS
Trusted Labs
5, rue du Bailliage
78000 Versailles, France
Samuel.Lacas@trusted-labs.fr

PIERRE VALARCHER
LIFAR – University of Rouen
Faculté des Sciences et des Techniques
76801 Saint Etienne du Rouvray, France
pierre.valarcher@univ-rouen.fr

Abstract. We define a translation of the elementary imperative language Loop into Gödel's system T with product types and we show that this translation actually defines lock-step simulation on the proviso that we assume a call-by-value evaluation of System T. As an application, we obtain that the Loop language (which is known to be expressive enough to represent all primitive recursive functions) does not allow to write a program that computes the minimum of two natural numbers *(n,m)* with the expected complexity *O(min(n,m))*.

1. Introduction

This paper is devoted to the study of the expressive power of an elementary imperative programming language similar to the Loop language described by Meyer and Ritchie in [19] extended with a *constant-time predecessor* operator.

While the λ-calculus is usually used to describe the denotational semantics of programming languages, we exploit it to encode the *operational semantics* of this imperative language. More precisely, we define a lock-step simulation of imperative programs by pure functional programs under call-by-value evaluation (actually terms of Gödel's system T). The semantics of the Loop language is given in the style of Plotkin's Structured Operational Semantics [22], while the semantics of Gödel's system T is given (as usual for the λ-calculus) as a set of reduction rules. This lock-step simulation is the main contribution of this paper.

This simulation theorem enables us to derive properties concerning this imperative language from previously known results in the λ-calculus. An example of such result which can be derived using our framework is related to the so-called *minimum problem*: there is no program in the Loop language which computes the minimum of two natural numbers *n* and *m* in time *O(min(n,m))*.

The fundamental property (from which such results are derived) in call-by-value system T is a generalization of a result by Colson and Fredholm called the "ultimate obstinacy" property in [3]. Unfortunately, the proof techniques developed in [3] did

Received January 30, 2006.

not generalize easily to system T equipped with product types and a constant-time predecessor operator. This led us to give a new (direct) proof of this property, which is the second contribution of this paper.

The plan of the paper is the following. Section 2 is devoted to the presentation of our target functional language which corresponds to Gödel system T equipped with a one-step predecessor function, product types and the call-by-value strategy. Our variant of the Loop-language (syntax and semantics) is presented in section 3 and the lock-step simulation is described in details in section 4. In section 5, we give a direct proof of the "ultimate obstinacy" theorem (and its corollary) for our extended system T.

Related works

In his pioneering work [2], Colson applies denotational semantics (with the domain of lazy integers [10]) and proves that although the function that computes the minimum of two natural numbers is obviously primitive recursive (see for instance [21] for a formal definition), there is no way to implement it efficiently as a *primitive recursive algorithms* (represented by *PR*-combinators in [2]). His main theorem states the *ultimate obstinacy* property of *PR*-combinators in all reduction strategy. Informally, this property expresses the fact that once an algorithm has started to consume an argument, it will never switch to another argument (a constructive proof of this property by Coquand may be found in [4]).

This idea has been developed further by David [7] who defines a *trace semantics* which allows him to prove a stronger property (the *backtracking property*). The third author has proved in [24] similar results about intensional behavior of other primitive recursive schemes, and most of these results have been extended in [8]. At about the same time, Colson and Fredholm [11] proved that *minimum problem* has still no solution even if we allow recursion over lists of natural numbers (assuming a call-by-value evaluation strategy). On the other hand, David [6] provides solution to the minimum problem, but assuming this time a call-by-name evaluation strategy.

Similar results have been obtained for Gödel's system T. Colson and Fredholm [3] proved that (assuming a call-by-value evaluation strategy) for any algorithm which computes a non-trivial binary function (where trivial means constant or projection plus constant), the time-complexity is at least linear in one of the inputs. This is again in contrast to the call-by-name case.

Related issues concerning non-determinism and sequentiality have been studied by Brookes and Dancannet [1, 5]. The authors proved that for the Berry-Curien programming language of sequential algorithms, the minimum problem has a solution (but not the "natural" one). They also developed an intensional semantics based on abstract circuits. In this framework, a program is mapped to a circuit, whose dimensions tells the quantity of parallel work and time needed to run a program.

In another line of work, Moschovakis [20] established linear lower bounds for the complexity of non-trivial primitive recursive algorithm from *piecewise linear* given functions (division by 2, minimum, lesser than). For instance, he proves that Stein's

algorithm for the greatest common divisor cannot be implemented efficiently by a primitive recursive algorithm.

Recently, a partial solution to an open problem concerning the classical *Euclidan algorithm* mentioned in [20] was found by Van Den Dries [26]. These results seem to indicate that many problems have no adequate solutions as primitive recursive algorithms.

2. Call-by-value system T

Gödel's system T is the extension of the simply typed λ-calculus with a type of natural numbers, and equipped with primitive recursion at all types. System T was first introduced in logic [13, 23] where it was mainly used in proof-theoretic studies of Peano arithmetic. An important result states that functions on the natural numbers definable in this system is correspond exactly to provably total functions in first-order Peano arithmetic.

Our approach is closer to [12], where system T is used to give a formal semantics to constructs found in higher-order programming languages. We recall two main results from [12]: well-typed terms are strongly normalizing and the rewriting system enjoys the Church-Rosser property.

In this section, we extend system T with product types (tuples and n-ary functions) and a constant-time predecessor operation. The rewriting system is summed up in figure 1, where we denote by $t[u_1/x_1, \ldots, u_n/x_n]$ the usual capture-avoiding substitution. Note that the reduction rules presented here implements the so-called weak reduction call-by-value strategy (since we do not reduce under an abstraction). We also recall the type system in figure 2 and we shall only consider well-typed terms in the sequel.

2.1 Examples

- $\mathbf{rec}(x_2, x_1, \lambda y.\lambda x.S(y))$ computes the sum of (x_1, x_2) by induction on x_2.
- $\mathbf{rec}(x_1, x_2, \lambda y.\lambda x.S(y))$ computes the sum of (x_1, x_2) by induction on x_1.
- $(\mathbf{rec}(x_1, \lambda y.S(y), \lambda u.\lambda x.\lambda y.\mathbf{rec}(y, (u\ S(0)), \lambda w.\lambda z.(u\ w)))\ x_2)$ is a term which computes Ackermann's function on (x_1, x_2) (which is known not to be primitive recursive [21]).

2.2 Derived form

We write $\mathbf{let}\ (x_1, \ldots, x_n) = u\ \mathbf{in}\ t$ as an abbreviation for the redex $\lambda(x_1, \ldots, x_n).t\ u$ and we obtain thus the following derived typing rule and evaluation rule:

$$\frac{\Gamma \vdash u : (\tau_1 * \ldots * \tau_n) \quad \Gamma, x_1 : \tau_1, \ldots x_n : \tau_n \vdash t : \sigma}{\Gamma \vdash \mathbf{let}\ (x_1, \ldots, x_n) = u\ \mathbf{in}\ t : \sigma}$$

$$C[\mathbf{let}\ (x_1, \ldots, x_n) = (v_1, \ldots, v_n)\ \mathbf{in}\ t] \rightsquigarrow C[t[v_1/x_1, \ldots, v_n/x_n]]$$

Moreover, the following property holds:

Types

$$\begin{array}{rcl} \tau & ::= & N \\ & & \tau_1 \to \tau_2 \\ & & \tau_1 \times \ldots \times \tau_n \end{array}$$

Values

$$\begin{array}{rcl} v & ::= & x \\ & | & 0 \\ & | & S(v) \\ & | & \lambda(x_1, \ldots, x_n).t \end{array}$$

Terms

$$\begin{array}{rcl} t & ::= & x \\ & | & 0 \\ & | & S(t) \\ & | & \mathbf{pred}(t) \\ & | & t_1\ t_2 \\ & | & \lambda(x_1, \ldots, x_n).t \\ & | & (t_1, \ldots, t_n) \\ & | & \mathbf{rec}(t_1, t_2, t_3) \end{array}$$

Contexts

$$\begin{array}{rcl} C[\,] & ::= & [\,] \\ & | & C[\,]\ t \\ & | & v\ C[\,] \\ & | & S(C[\,]) \\ & | & \mathbf{pred}(C[\,]) \\ & | & \mathbf{rec}(C[\,], t_2, t_3) \\ & | & \mathbf{rec}(v_1, C[\,], t_3) \\ & | & \mathbf{rec}(v_1, v_2, C[\,]) \\ & | & (v_1, \ldots, v_{i-1}, C[\,], t_{i+1}, \ldots, t_n) \end{array}$$

Evaluation rules

$$\begin{array}{rcl} C[\mathbf{pred}(0)] & \rightsquigarrow & C[0] \\ C[\mathbf{pred}(S(v))] & \rightsquigarrow & C[v] \\ C[\mathbf{rec}(0, v_2, v_3)] & \rightsquigarrow & C[v_2] \\ C[\mathbf{rec}(S(v_1), v_2, v_3)] & \rightsquigarrow & C[(v_3\ \mathbf{rec}(v_1, v_2, v_3)\ v_1)] \\ C[\lambda(x_1, \ldots, x_n).t\ (v_1, \ldots, v_n)] & \rightsquigarrow & C[t[v_1/x_1, \ldots, v_n/x_n]] \end{array}$$

Fig. 1: Gödel's system T

LEMMA 1. *If* $u \rightsquigarrow u'$ *then* $\mathbf{let}\ (x_1, \ldots, x_n) = u\ \mathbf{in}\ t \rightsquigarrow \mathbf{let}\ (x_1, \ldots, x_n) = u'\ \mathbf{in}\ t$ *for any term* t.

PROOF. Indeed, since $\mathbf{let}\ (x_1, \ldots, x_n) = \bullet\ \mathbf{in}\ t$ is an evaluation context. □

REMARK 1. Let us define the *degree* $\partial(\tau)$ of a type τ inductively with $\partial(N) = 0$, $\partial(\sigma \to \tau) = max(\partial(\sigma) + 1, \partial(\tau))$ and $\partial(\tau_1 \times \ldots \times \tau_n) = max(\partial(\tau_1), \ldots, \partial(\tau_n))$ and the *degree* $\partial(t)$ of a term t as the maximum degree of types τ such that $\mathbf{rec}(t_1, t_2, t_3)$ occurs in t with type τ. If T_n stands for the restriction of system T to terms with degree less than n, it is well-known that functions of type $N^k \to N$ which can be represented by a term of T_0 correspond exactly to primitive recursive function.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \subset \Gamma'}{\Gamma' \vdash t : \tau}$$

$$\frac{}{\Gamma \vdash 0 : N} \qquad \frac{\Gamma \vdash t : N}{\Gamma \vdash S(t) : N} \qquad \frac{\Gamma \vdash t : N}{\Gamma \vdash \mathbf{pred}(t) : N}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_1, \ldots, t_n) : \tau_1 \times \ldots \times \tau_n}$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \sigma}{\Gamma \vdash \lambda(x_1, \ldots, x_n).t : (\tau_1 \times \ldots \times \tau_n) \to \sigma} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ u : \tau}$$

$$\frac{\Gamma \vdash t_1 : N \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau \to N \to \tau}{\Gamma \vdash \mathbf{rec}(t_1, t_2, t_3) : \tau}$$

Fig. 2: Type system

3. The Loop language

The imperative language we consider is a minor syntactic variant of the Loop language defined by Meyer and Ritchie in [19] (see also the textbook by Davis and Weyuker [9]). Variables (called *registers* in [19]) can contain only natural numbers. Atomic statements are assignments and the only control structure are sequences (lists of statements separated by semi-colons) and for-loops (which implement bounded iteration). The formal syntax of Loop programs is given in the following definition.

Definition 1. *The syntax of expressions e, commands c and sequences s and programs p are defined by the following grammar:*

$$\begin{array}{lcl} e & ::= & \overline{n} \mid x \mid x+1 \mid x-1 \\ c & ::= & x := e \\ & \mid & \mathbf{for}\ x := 1\ \mathbf{to}\ e\ \{s\} \\ & \mid & \{s\} \\ s & ::= & \varepsilon \mid c; s \\ p & ::= & \{s\} \end{array}$$

Metavariables x range over identifiers and $\overline{n}$ over natural numbers literals (which are the only values). The symbol ε denotes the empty sequence and {} is called the empty block (which is a command).

EXAMPLE 1. A simple program that computes the sum of (x_1, x_2) (and stores the result in variable r):

```
{
  r := x1;
  for y := 1 to x2 {
    r := r + 1;
  };
}
```

3.1 Operational semantics

Following [14], the operational semantics of the Loop language is given by a simple transition system. The following rules define a transition relation on configurations, where a configuration is either a value, a pair $\langle e, \mu \rangle$ (respectively $\langle c, \mu \rangle$) consisting of an expression e (respectively a command c) and a store μ. For simplicity, we assume that a store μ is itself represented as a pair $(\vec{x}, \vec{n})$ where $\vec{x}$ is tuple of variables and $\vec{n}$ is a tuple of natural numbers (of same length). Thus a store maps variables to natural numbers in the obvious way, and we write $\mu(x)$ for the value of x in store μ, and $\mu[x := n]$ for an update of the value of x in store μ.

Expressions

$$\langle x_i + 1, \mu \rangle \to \mu(x_i) + 1 \tag{1}$$

$$\langle x_i - 1, \mu \rangle \to \mu(x_i) \dot{-} 1 \tag{2}$$

Assignments

$$\langle \{x_i := \overline{n}; s\}, \mu \rangle \to \langle \{s\}, \mu[x_i := n] \rangle \tag{3}$$

$$\left\langle \{x_i := x_j; s\}, \mu \right\rangle \to \left\langle \{s\}, \mu[x_i := \mu(x_j)] \right\rangle \tag{4}$$

$$\frac{\langle e, \mu \rangle \to n}{\langle \{x_i := e; s\}, \mu \rangle \to \langle \{x_i := \overline{n}; s\}, \mu \rangle} \tag{5}$$

where e is neither a constant nor a variable.

Empty block

$$\langle \{\{\}; s\}, \mu \rangle \to \langle \{s\}, \mu \rangle \tag{6}$$

Sequence

$$\frac{\langle c, \mu \rangle \to \langle c_1, \mu_1 \rangle}{\langle \{c; s\}, \mu \rangle \to \langle \{c_1; s\}, \mu_1 \rangle} \tag{7}$$

where c is neither the empty block nor an assignment.

Loop

$$\langle \textbf{for } x_i := 1 \textbf{ to } e \ \{s\}, \mu\rangle \to \langle \{\}, \mu\rangle \tag{8}$$

where e is either the constant 0 or a variable x_j and then $\mu(x_j) = 0$.

$$\langle \textbf{for } x_i := 1 \textbf{ to } e \ \{s\}, \mu\rangle \to \left\langle \{\textbf{for } x_i := 1 \textbf{ to } \overline{n} \ \{s\}; x_i := \overline{n+1}; s\}, \mu \right\rangle \tag{9}$$

where e is either a constant $n+1$ or a variable x_j and then $\mu(x_j) = n+1$.

Remark 2. In a loop of the form **for** $x_i := 1$ **to** x_j $\{s\}$, both variables x_i and x_j are allowed to be assigned in the body $\{s\}$. However, the variable x_j is looked up before the beginning of the loop and x_i is assigned its new value before executing the body of the loop. The expected semantics of the for-loop (as well as its termination) is thus enforced.

Remark 3. Note that if a variable b_1 contains only boolean values 0 and 1, a loop of the form **for** $i := 1$ **to** $b_1\{s\}$ corresponds to the conditional **if** b **then** $\{s\}$. The control structure **if** b_1 **then** $\{s_1\}$ **else** $\{s_2\}$ can then be simulated by the sequence $\{b_2 := 1;$ **if** b_1 **then** $\{s_1; b_2 := 0\};$ **if** b_2 **then** $\{s_2\}\}$.

We are now able to define the semantics of a program (as usual $\to^\star$ stands for the reflexive and transitive closure of $\to$).

Definition 2. *Given an initial store μ, we say that a program p evaluates to μ' if and only if $\langle p, \mu\rangle \to^\star \langle \{\}, \mu'\rangle$*

Remark 4. This semantics is deterministic since there is always at most one rule which can be applied (depending on the environment and the shape of the program). Moreover, the only case where no rule can be applied corresponds to the final configuration (when the program amounts to an empty block).

4. Lock-step simulation

In this section, we present the lock-step simulation of Loop programs by λ-terms of system T. We first show how to encode a Loop programs as a λ-term and then we prove that this encoding is such that the evaluation of an imperative program runs in lock-step with the reduction of its image.

4.1 Translation

In order to distinguish the successor S (which is a constructor) from the successor seen as an operation (whose evaluation should imply a reduction step), we use the keyword **succ** as an abbreviation for $\lambda x.S(x)$ in the following definition:

Definition 3. *The translation * of a* Loop *program with variables $\vec{x} = (x_1, \ldots, x_k)$ into a term is defined by induction on expressions and commands as follows:*

- $\overline{n}^\star = S^n(0)$

- $x_i^\star = x_i$
- $(x_i + 1)^\star = \mathbf{succ}(x_i)$
- $(x_i - 1)^\star = \mathbf{pred}(x_i)$
- $\{\}^\star = \vec{x}$
- $\{x_i := e; s\}^\star = \mathbf{let}\ x_i = e^\star\ \mathbf{in}\ \{s\}^\star$
- $\{c; s\}^\star = \mathbf{let}\ \vec{x} = c^\star\ \mathbf{in}\ \{s\}^\star$ *if c is not an assignment*
- $(\mathbf{for}\ x_i := 1\ \mathbf{to}\ e\ \{s\})^\star = \mathbf{rec}(e^\star, \vec{x}, \lambda\vec{x}.\lambda x_i.\{s\}^\star)$
  where e is either a constant or a variable

EXAMPLE 2. Let us translate the following program p which computes the division by two (where $\vec{x} = (x_1, x_2, b_1, b_2, i, j, k)$).

```
{
  r := x1;
  b1 := 1;
  for k := 1 to x1 {
    b2 := 1;
    for i := 1 to b1 {    // if b1 then
      r := r − 1;
      b1 := 0;
      b2 := 0;
    };
    for j := 1 to b2 {    // else
      b1 := 1;
    };
  };
}
```

We obtain $p^\star_{\vec{x}} =$

```
(
  let r  =  x1 in
  let b1  =  1 in
  let x⃗ = rec(x1, x⃗, λx⃗.λk.
    let b2  = 1 in
    let x⃗ =  rec(b1, x⃗, λx⃗.λi.
      let r  =  pred(r) in
      let b1  =  0 in
      let b2  =  0 in
    x⃗) in
    let x⃗ =  rec(b2, x⃗, λx⃗.λj.
      let b1 = 1 in
    x⃗) in
  x⃗)
)
```

REMARK 5. Note that the translation clearly involves only terms with degree 0 (since $\vec{x}$ is always a tuple of natural numbers) and thus the target language is actually T_0 (and not full system T).

4.2 Simulation theorem

Recall that a store μ is by construction a pair $(\vec{x}, \vec{n})$ where $\vec{x}$ is tuple of variables and $\vec{n}$ is a tuple of natural numbers (of same length). The following lemma (respectively theorem) states that the evaluation of an expression (respectively program) runs in lock-step with the reduction of its image.

LEMMA 2. *If* $\langle e, (\vec{x}, \vec{n})\rangle \to q$ *then* $e^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow q^\star$

PROOF. By case on the rule $\langle e, (\vec{x}, \vec{n})\rangle \to q$ being used.

- Rule 1

$$(x_i + 1)^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow (\mu(x_i) + 1)^\star$$

  Indeed, $(x_i + 1)^\star[\vec{n}^\star/\vec{x}] = \mathbf{succ}(x_i)[\vec{n}^\star/\vec{x}] = \mathbf{succ}(n_i^\star) \rightsquigarrow S(n_i^\star) = (n_i + 1)^\star$

- Rule 2

$$(x_i - 1)^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow (\mu(x_i) \dot{-} 1)^\star$$

  Indeed, $(x_i - 1)^\star[\vec{n}^\star/\vec{x}] = \mathbf{pred}(x_i)[\vec{n}^\star/\vec{x}] = \mathbf{pred}(n_i^\star)$ and if $n_i = 0$ then $\mathbf{pred}(n_i^\star) \rightsquigarrow 0$ and $\mathbf{pred}(n_i^\star) \rightsquigarrow (n_i - 1)^\star$ otherwise.

THEOREM 1. *If* $\langle c, (\vec{x}, \vec{n})\rangle \to \langle c_1, (\vec{x}, \vec{n}_1)\rangle$ *then* $c^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow c_1^\star[\vec{n}_1^\star/\vec{x}]$

PROOF. By induction on the derivation of $\langle c, (\vec{x}, \vec{n})\rangle \to \langle c_1, (\vec{x}, \vec{n}_1)\rangle$.

- Rule 3

$$(x_i := \overline{q}; s)^\star[\vec{n}^\star/\vec{x}] = \{s\}^\star[(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_k)^\star/\vec{x}]$$

  Indeed, $\{x_i := \overline{q}; s\}^\star[\vec{n}^\star/\vec{x}]$

$$\begin{array}{rl} = & (\mathbf{let}\ x_i = q^\star\ \mathbf{in}\ \{s\}^\star)[(n_1, \ldots, n_k)^\star/\vec{x}] \\ = & \mathbf{let}\ x_i = q^\star\ \mathbf{in}\ \{s\}^\star[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^\star/\vec{x}] \\ \rightsquigarrow & \{s\}^\star[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^\star/\vec{x}][q^\star/x_i] \\ = & \{s\}^\star[(n_1, \ldots, n_{i-1}, q, n_{i+1}, \ldots, n_k)^\star/\vec{x}] \end{array}$$

- Rule 4

$$(x_i := x_j; s)^\star[\vec{n}^\star/\vec{x}] = \{s\}^\star[(n_1, \ldots, n_{i-1}, n_j, n_{i+1}, \ldots, n_k)^\star/\vec{x}]$$

  Indeed, $\{x_i := x_j; s\}^\star[\vec{n}^\star/\vec{x}]$

$$\begin{array}{rl} = & (\mathbf{let}\ x_i = c_j^\star\ \mathbf{in}\ \{s\}^\star)[(n_1, \ldots, n_k)^\star/\vec{x}] \\ = & \mathbf{let}\ x_i = c_j^\star\ \mathbf{in}\ \{s\}^\star[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^\star/\vec{x}] \\ \rightsquigarrow & \{s\}^\star[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^\star/\vec{x}][n_j^\star/x_i] \\ = & \{s\}^\star[(n_1, \ldots, n_{i-1}, n_j, n_{i+1}, \ldots, n_k)^\star/\vec{x}] \end{array}$$

- Rule 5

$$\frac{e^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow q}{\{x_i := e; s\}^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{x_i := \overline{q}; s\}^\star[\vec{n}^\star/\vec{x}]}$$

Indeed, $\{x_i := e; s\}^\star[\vec{n}^\star/\vec{x}]$

$$\begin{array}{ll} = & (\textbf{let } x_i = e^\star \textbf{ in } \{s\}^\star)[(n_1, \ldots, n_k)^\star/\vec{x}] \\ = & \textbf{let } x_i = e^\star[\vec{n}^\star/\vec{x}] \textbf{ in } (\{s\}^\star[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^\star/\vec{x}]) \end{array}$$

By induction hypothesis, $e^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow q$ and then by lemma 1:

$\textbf{let } x_i = e^\star[\vec{n}^\star/\vec{x}] \textbf{ in } (\{s\}^\star[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^\star/\vec{x}])$

$$\begin{array}{ll} \rightsquigarrow & \textbf{let } x_i = q^\star \textbf{ in } (\{s\}^\star[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^\star/\vec{x}]) \\ = & (\textbf{let } x_i = q^\star \textbf{ in } \{s\}^\star)[(n_1, \ldots, n_k)^\star/\vec{x}] \\ = & \{x_i := q; s\}^\star[\vec{n}^\star/\vec{x}] \end{array}$$

- Rule 6

$$\{\{\}; s\}^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{s\}^\star[\vec{n}^\star/\vec{x}]$$

Indeed, $\textbf{let } \vec{x} = \{\}^\star \textbf{ in } \{s\}^\star[\vec{n}^\star/\vec{x}]$

$$\begin{array}{ll} = & \textbf{let } \vec{x} = \{\}^\star \textbf{ in } \{s\}^\star[\vec{n}^\star/\vec{x}] \\ = & \textbf{let } \vec{x} = \vec{n}^\star \textbf{ in } \{s\}^\star \\ \rightsquigarrow & \{s\}^\star[\vec{n}^\star/\vec{x}] \end{array}$$

- Rule 7

$$\frac{c^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow c_1^\star[\vec{n}^\star/\vec{x}]}{\{c; s\}^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{c_1; s\}^\star[\vec{n}^\star/\vec{x}]}$$

Indeed, since c is neither an assignment not the empty block,

$$\begin{array}{rcl} \{c; s\}^\star[\vec{n}^\star/\vec{x}] & = & \textbf{let } \vec{x} = c^\star \textbf{ in } \{s\}^\star[\vec{n}^\star/\vec{x}] \\ & = & \textbf{let } \vec{x} = c^\star[\vec{n}^\star/\vec{x}] \textbf{ in } \{s\}^\star \end{array}$$

By induction hypothesis, $c^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow c_1^\star[\vec{n}_1^\star/\vec{x}]$ and then by lemma 1:

$$\begin{array}{rcl} \textbf{let } \vec{x} = c^\star[\vec{n}^\star/\vec{x}] \textbf{ in } \{s\}^\star & \rightsquigarrow & \textbf{let } \vec{x} = c_1^\star[\vec{n}_1^\star/\vec{x}] \textbf{ in } \{s\}^\star \\ & = & \textbf{let } \vec{x} = c_1^\star \textbf{ in } \{s\}^\star[\vec{n}_1^\star/\vec{x}] \\ & = & \{c_1; s\}^\star[\vec{n}_1^\star/\vec{x}] \end{array}$$

- Rule 8

$$(\textbf{for } x_i := 1 \textbf{ to } e\ \{s\})^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{\}^\star[\vec{n}^\star/\vec{x}]$$

Indeed, since e is either the constant 0 or a variable x_j and $n_j = 0$:

$$\begin{array}{rcl} (\textbf{for } x_i := 1 \textbf{ to } e\ \{s\})^\star[\vec{n}^\star/\vec{x}] & = & \textbf{rec}(e^\star, \vec{x}, \lambda\vec{x}.\lambda x_i.\{s\}^\star)[\vec{n}^\star/\vec{x}] \\ & = & \textbf{rec}(0, \vec{n}^\star, \lambda\vec{x}.\lambda x_i.\{s\}^\star) \\ & \rightsquigarrow & \vec{n}^\star \\ & = & \{\}^\star[\vec{n}^\star/\vec{x}] \end{array}$$

- Rule 9

  $(\mathbf{for}\ x_i := 1\ \mathbf{to}\ e\ \{s\})^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{\mathbf{for}\ x_i := 1\ \mathbf{to}\ \overline{q}\ \{s\}; x_i := \overline{q+1}; s\}^\star[\vec{n}^\star/\vec{x}]$

  Indeed, since e is either a constant $q+1$ or a variable x_j and $n_j = q+1$:

  $(\mathbf{for}\ x_i := 1\ \mathbf{to}\ e\ \{s\})^\star[\vec{n}^\star/\vec{x}]$

  $$\begin{array}{ll}
  = & \mathbf{rec}(e^\star, \vec{x}, \lambda\vec{x}\lambda x_i\{s\}^\star)[\vec{n}^\star/\vec{x}] \\
  = & \mathbf{rec}(S^{q+1}(0), E^\star, \lambda\vec{x}.\lambda x_i.\{s\}^\star) \\
  \rightsquigarrow & \mathbf{let}\ \vec{x} = \mathbf{rec}(S^q(0), E^\star, \lambda\vec{x}.\lambda x_i.\{s\}^\star)\ \mathbf{in\ let}\ x_i = S^{q+1}(0)\ \mathbf{in}\ \{s\}^\star \\
  = & (\mathbf{let}\ \vec{x} = \mathbf{rec}(S^q(0), \vec{x}, \lambda\vec{x}.\lambda x_i.\{s\}^\star)\ \mathbf{in\ let}\ x_i = S^{q+1}(0)\ \mathbf{in}\ \{s\}^\star)[\vec{n}^\star/\vec{x}] \\
  = & (\mathbf{let}\ \vec{x} = \mathbf{rec}(S^q(0), \vec{x}, \lambda\vec{x}.\lambda x_i.\{s\}^\star)\ \mathbf{in}\ \{x_i := \overline{q}+1; s\}^\star)[\vec{n}^\star/\vec{x}] \\
  = & (\mathbf{let}\ \vec{x} = (\mathbf{for}\ x_i := 1\ \mathbf{to}\ \overline{q}\ \{s\})^\star\ \mathbf{in}\ \{x_i := \overline{q}+1; s\}^\star)[\vec{n}^\star/\vec{x}] \\
  = & \{\mathbf{for}\ x_i := 1\ \mathbf{to}\ \overline{q}\ \{s\}; x_i := \overline{q}+1\ ; s\}^\star[\vec{n}^\star/\vec{x}]
  \end{array}$$

5. Ultimate Obstinacy

Colson and Fredholm proved in [3] that (non-trivial) terms of system T evaluated by value are ultimately obstinate. In this section, we show that this property still holds for system T in the presence a constant-time predecessor function and product types. However, since the techniques developed in [3] do not seem to generalize easily to this case, we present here a new direct proof of this result.

We first extend the syntax of terms and values with an infinite number of constants $\bot_i^k$ of type N and we supplement our set of reduction rules with the following rule:

$$C[\mathbf{pred}(\bot_i^k)] \rightsquigarrow C[\bot_i^{k+1}]$$

We denote by $\rightsquigarrow_\bot$ this extended rewriting system. Intuitively, a constant $\bot_i^k$ represents a "sufficiently large" natural number (on which the predecessor can be applied at least k times). Thus, we could view $\bot_i^k$ as $\mathbf{pred}^k(\bot_i^0)$ or, in other words, if we look at the i-th argument as an input channel, the superscript k tells us how far we may need to look ahead in this input channel.

The strong normalization of reduction $\rightsquigarrow_\bot^*$ for well-typed terms is a consequence of the following lemma (since the reduction $\rightsquigarrow$ is strongly normalizing for well-typed terms).

Lemma 3. *For any terms t, u, if $t[\bot_1^{k_1}/x_1, \ldots, \bot_n^{k_n}/x_n] \rightsquigarrow u[\bot_1^{k_1}/x_1, \ldots, \bot_n^{k_n}/x_n]$ then $t[0/x_1, \ldots, 0/x_n] \rightsquigarrow u[0/x_1, \ldots, 0/x_n]$.*

Since terms now possibly contain constants $\bot_i^k$ of type N, even a well-typed term can get stuck during evaluation (the term cannot be evaluated further) although it is not yet a value. This lemma already splits the set of terms into constant, trivial and ultimately obstinate terms.

LEMMA 4. *Given a well-typed term t of type N with free variables $x_1, \dots, x_n$ of type N, if $t[\bot_1^0/x_1, \dots, \bot_n^0/x_n] \rightsquigarrow_\bot^p v$ where v is in normal form then one of the following cases holds:*

- *v is a value $S^k(0)$ (we say that t is **constant**)*
- *v is a value $S^k(\bot_i^m)$ with $m \leqslant p$ (we say that t is **trivial**)*
- *there is an evaluation context $C[\ ]$ such that $v = C[\mathbf{rec}(\bot_i^m, v_2, v_3)]$ and $m \leqslant p$ (we say that t is **ultimately obstinate**)*

PROOF. By inspection of normal forms (and then by induction on the derivation to show that $m \leqslant p$). □

DEFINITION 4. *The S-substitution $t[\![S(\bot_i)/\bot_i]\!]$ is the homomorphic extension of the basic function defined as follows:*

- $\bot_i^0[\![S(\bot_i)/\bot_i]\!] = S(\bot_i^0)$
- $\bot_i^{k+1}[\![S(\bot_i)/\bot_i]\!] = \bot_i^k$

We write $t[\![S^p(\bot_i)/\bot_i]\!]$ for the generalized S-substitution defined by induction on p with $t\left[\!\left[S^0(\bot_i)/\bot_i\right]\!\right] = t$ and $t\left[\!\left[S^{p+1}(\bot_i)/\bot_i\right]\!\right] = t[\![S(\bot_i)/\bot_i]\!][\![S^p(\bot_i)/\bot_i]\!]$.

LEMMA 5. *For any terms t, u, if $t \rightsquigarrow u$ then $t[\![S(\bot_i)/\bot_i]\!] \rightsquigarrow u[\![S(\bot_i)/\bot_i]\!]$*

PROOF. It is easy to check that S-substitution commutes with the reduction rules given in figure 1. It remains to show that S-substitution commutes with the rule $C[\mathbf{pred}(\bot_i^k)] \rightsquigarrow C[\bot_i^{k+1}]$. Indeed, if $C'[\ ] = C[\ ][\![S(\bot_i)/\bot_i]\!]$ then:

$C[\mathbf{pred}(\bot_i^0)][\![S(\bot_i)/\bot_i]\!] = C'[\mathbf{pred}(S(\bot_i^0))] \rightsquigarrow C'[\bot_i^0] = C[\bot_i^1][\![S(\bot_i)/\bot_i]\!]$

$C[\mathbf{pred}(\bot_i^{k+1})][\![S(\bot_i)/\bot_i]\!] = C'[\mathbf{pred}(\bot_i^k)] \rightsquigarrow C'[\bot_i^{k+1}] = C[\bot_i^{k+2}][\![S(\bot_i)/\bot_i]\!]$

□

The following lemma (together with lemma 3) allows us to apply lemma 4 on genuine natural numbers.

LEMMA 6. *Given a well-typed term t with constants $\bot_1^0, \dots, \bot_n^0$ of type N, if $t \rightsquigarrow_\bot v$ then $t[\![S^p(\bot_1)/\bot_1, \dots, S^p(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot v[\![S^p(\bot_1)/\bot_1, \dots, S^p(\bot_n)/\bot_n]\!]$*

PROOF. Apply lemma 5 repeatedly p times for each $\bot_i$. □

LEMMA 7. (CONSTANT TERMS) *Given a well-typed constant term $t : N$ with free variables $x_1, \dots, x_n$ of type N, there exist $k, p \in \mathbb{N}$ such that for any $p_1 \geqslant 0, \dots, p_n \geqslant 0$, $t[S^{p_1}(0)/x_1, \dots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(0)$.*

PROOF. Let $t' = t[\bot_1^0/x_1, \dots, \bot_n^0/x_n]$ and p, k be such that $t' \rightsquigarrow^p S^k(0)$ by lemma 4. Since $t[S^{p_1}(\bot_1^0)/x_1, \dots, S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1, \dots, S^{p_n}(\bot_n)/\bot_n]\!]$ and by lemma 6, $t'[\![S^{p_1}(\bot_1)/\bot_1, \dots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(0)$. We conclude by lemma 3 that $t[S^{p_1}(0)/x_1, \dots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(0)$. □

LEMMA 8. (TRIVIAL TERMS) *Given a well-typed trivial term $t : N$ with free variables $x_1, \ldots, x_n$ of type N, there exist i, m, p, k with $1 \leqslant i \leqslant n$ and $m \leqslant p$ such that for any $p_1 \geqslant 0, \ldots, p_n \geqslant 0$, $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^{p_i \dot{-} m+k}(0)$.*

PROOF. Let $t' = t[\bot_1^0/x_1, \ldots, \bot_n^0/x_n]$ and p, k, m be such that $t' \rightsquigarrow^p S^k(\bot_i^m)$ by lemma 4. Since $t[S^{p_1}(\bot_1^0)/x_1, \ldots, S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!]$ and by lemma 6, $t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(S^{p_i-m}(\bot_i^0))$ if $p_i \geqslant m$, and $t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p S^k(\bot_i^{m-p_i})$ if $p_i < m$. We conclude by lemma 3 that $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p S^k(S^{p_i \dot{-} m}(0))$. □

REMARK 6. Clearly, trivial terms compute only functions definable (with the usual mathematical notation) as $(x_1, \ldots, x_n) \mapsto (x_i \dot{-} m) + k$ (for some constants m, k).

LEMMA 9. *Given an evaluation context $C[\ ]$, if $C[\mathbf{rec}(S^k(0), v_2, v_3)]$ is well-typed and $k > 0$ then $C[\mathbf{rec}(S^k(0), v_2, v_3)] \rightsquigarrow C'[\mathbf{rec}(S^{k-1}(0), v_2, v_3)]$ where $C'[\ ]$ is again an evaluation context.*

PROOF. Indeed, $C' = C[(v_3 \bullet S^k(0))]$. □

THEOREM 2. (ULTIMATELY OBSTINATE TERMS) *Given a well-typed ultimately obstinate term t of type N with free variables $x_1, \ldots, x_n$ of type N, there exist $i, m \in \mathbb{N}$ with $1 \leqslant i \leqslant n$ such that for any $p_1 \geqslant p, \ldots, p_n \geqslant p$, $t[S^{p_1}(0)/x_1, \ldots, S^{p_n}(0)/x_n]$ reaches its normal form in at least p_i reductions steps.*

PROOF. Indeed, if $t' = t[\bot_1^0/x_1 \ldots, \bot_n^0/x_n]$ then $t' \rightsquigarrow_\bot^p C[\mathbf{rec}(\bot_i^m, v_2, v_3)]$ by lemma 4. Now, $t[S^{p_1}(\bot_1^0)/x_1, \ldots, S^{p_n}(\bot_n^0)/x_n] = t'[\![S^{p_1}(\bot_1)/\bot_1, \ldots, S^{p_n}(\bot_n)/\bot_n]\!]$ and $t'[\![S^{p_1}(\bot_1)/\bot_1 \ldots, S^{p_n}(\bot_n)/\bot_n]\!] \rightsquigarrow_\bot^p C[\mathbf{rec}(\bot_i^{p_i-m}, v_2, v_3)]$ by lemma 6 (since $p_i \geqslant m$), and $t[S^{p_1}(0)/x_1 \ldots, S^{p_n}(0)/x_n] \rightsquigarrow^p C[\mathbf{rec}(S^{p_i-m}(0), v_2, v_3)]$ by lemma 3. Thus, by lemma 9, $C[\mathbf{rec}(S^{p_i-m}(0), v_2, v_3)] \rightsquigarrow^{p_i-m} C[\mathbf{rec}(0, v_2, v_3)]$. We conclude since $p \geqslant m$ and thus $p + p_i - m \geqslant p_i$. □

COROLLARY 1. *There is no term in system T with product types and constant-time predecessor which computes the minimum of two natural numbers n,m with time-complexity $O(min(n, m))$.*

REMARK 7. Note that the proof of the ultimate obstinacy property relies crucially on lemma 9. For instance, the property does not hold if we consider the following reduction rule for **rec** (which is derivable, but not in call-by-value):

$$\mathbf{rec}(S(n), v, \lambda x.\lambda y.t) \quad \rightsquigarrow \quad t[n/x, \mathbf{rec}(n, b, \lambda x.\lambda y.t)/y]$$

Indeed, since we consider weak reduction and since y may occurs under the scope of a λ-abstraction in t, there no reason that $\mathbf{rec}(n, b, \lambda x.\lambda y.t)$ be the next redex to contract. In fact, one can easily show that call-by-name evaluation can be simulated under call-by-value evaluation using this rule and the usual thunk-based encoding [16].

We are now able to prove the result claimed in the introduction:

COROLLARY 2. *There is no program in the* LOOP*-language which computes the minimum of two natural numbers n,m with time-complexity* $O(min(n, m))$.

PROOF. By corollary 1 and theorem 1. □

6. Conclusion and future work

In the one hand, we have defined a lock-step simulation of LOOP programs by λ-terms of call-by-value system T extended with product types and constant-time predecessor operation. On the other hand, we have proved that Colson and Fredholm ultimate obstinacy property still holds in this extension of system T. As a corollary of both results, we obtained that the LOOP language does not allow to write a program that computes the minimum of two natural numbers *(n,m)* with the expected complexity *O(min(n,m))*.

It is known [2] that such a program may be represented in call-by-name system T. Similarly, it is clear that the following imperative program computes the minimum *(n,m)* with the good time-complexity (where **exit** corresponds to a new control structure with the obvious semantics):

```
{
  m' := m; r := n;
  for y := 1 to n {
    m := m − 1;
    if m = 0 then {r := m'; exit; };
  }
}
```

A question is then to determine whether there is a natural functional system and a lock-step simulation which accounts for this new control structure. This question is interesting since we conjecture that such a language would far more expressive from an algorithmic standpoint. For instance, *Stein*'s algorithm can be written in this language but likely not in the LOOP language (we refer to [20] for details concerning this issue). A similar result (but formulated in the framework of Abstract States Machines [15]) which also supports this conjecture may be found in [25].

Another future work is suggested by remark 5: programs of the LOOP language are actually mapped by our translation to λ-terms of T_0. In order to reach the full hierarchy of system T, a forthcoming paper shall be devoted to an extension of the LOOP language with higher-order procedures and procedural variables.

As a concluding remark, note that there is a growing interest in comparing the expressive power of different programming paradigms. For instance, recent works by Kristiansen, Niggl and Voda [17, 18] exhibit a connection between the classical complexity hierarchies found in imperative and functional languages. We believe that our translation may turn out to be convenient tool for this kind of investigation.

References

[1] S. Brookes and D. Dancanet. Sequential algorithms, deterministic parallelism, and intensional expressiveness. In *22nd Annual Symposium on POPL*, 1995.

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

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

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

[5] D. Dancanet and S. Brookes. Programming language expressiveness and circuit complexity. In *Internat. Conf. on the Mathematical Foundations of Programming Semantics*, 1996.

[6] R. David. Un algorithme primitif récursif pour la fonction inf. *Compt. Rend. de l'Aca. des Scie.*, 317 (Série I), 1993.

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

[8] R. David. Decidability results for primitive recursive algorithms. *Theor. Comput. Sci.*, 300(1-3):477–504, 2003.

[9] M. Davis and E. Weyuker. *Computability, Complexity and Languages*. Academic Press, 1983.

[10] M. H. Escardo. On lazy natural numbers with applications. *SIGACT News*, 24(1), 1993.

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

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

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

[14] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[15] Y. Gurevich. The sequential ASM thesis. *Bulletin of the EATCS*, 67:93, 1999.

[16] J. Hatcliff and O. Danvy. Thunks and the lambda-calculus. *J. Funct. Program.*, 7(3):303–319, 1997.

[17] L. Kristiansen and K.-H. Niggl. On the computational complexity of imperative programming languages. *Theor. Comput. Sci.*, 318(1-2):139–161, 2004.

[18] L. Kristiansen and P. J. Voda. Programming languages capturing complexity classes. In *Nordic Journal of Computing*, volume 12. 2005.

[19] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proc. ACM Nat. Meeting*, 1976.

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

[21] R. Peter. *Recursive Functions*. Academic Press, 1968.

[22] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, September 1981.

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

[24] P. Valarcher. Intensionality vs extensionality and primitive recursion. volume 1179, 1996.

[25] P. Valarcher, P. Andary, and B. Patrou. About the implementation of primitive recursive algorithms. In *Proceedings of the 12th International ASM Workshop*, Paris, March 2005.

[26] L. van den Dries. Generating the greatest common divisor, and limitations of primitive recursive algorithms. *Foundations of Computational Mathematics*, 3:297–322, 2003.