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 Meyer and Ritchie's Loop language into a subsystem of Gödel's system T (with product types). Then we show that this translation actually provides a lock-step simulation if a call-by-value strategy is assumed for system T. Some generalizations and possible applications are discussed.

1. Introduction

The Loop language has been introduced by Meyer and Ritchie in [11] and then extensively studied in the literature (see for instance the textbooks by Davis and Weyuker [5] or Calude [3]). Each Loop program only consists of assignment statements and loop statements (which implement bounded iteration). Meyer and Ritchie's proved in [11] that Loop programs compute exactly the class $\mathcal{PR}$ of primitive recursive functions. More precisely, they define a hierarchy of functions $\{\mathcal{L}_n\}$ by counting the nesting of loops in Loop programs and prove that $\mathcal{L}_n \subsetneqq \mathcal{L}_{n+1}$ and $\mathcal{PR} = \cup_{n\in\mathbb{N}}\{\mathcal{L}_n\}$.

Gödel's system T is the extension of the simply typed λ-calculus with a type of natural numbers and primitive recursion at all types. System T was first introduced in logic [7, 16] where it was mainly used in proof-theoretic studies of Peano arithmetic. An important result in this area states that functions on the natural numbers that are definable in this system correspond exactly to functions that are provably total in first-order Peano arithmetic (see for instance the survey [1]). However, if we call T_0 the restriction of system T to terms that contain only recursors of type N, it is also well-known [1] that functions of type $N^k \to N$ that can be computed by a term of T_0 are exactly the primitive recursive functions.

The main contribution of this paper is the following. We define a translation of the Loop language into call-by-value system T_0 with product types and we prove that this translation actually provides a lock-step simulation: each evaluation step of a Loop program is mapped to one reduction step of the transformed program.

We would like to stress that although the λ-calculus is usually used to describe the denotational semantics of programming languages, we exploit it here to formalize the *operational semantics* of the Loop language. Our approach in this paper is thus closer to [6], where system T is used to give a formal semantics to constructs found in (higher-order) programming languages. In particular, system T is formulated here as a functional programming language with product types and a call-by-value evaluation strategy. As usual for the λ-calculus, the formal semantics is formulated as a contextual semantics (a set of reduction rules together with an inductive definition of evaluation contexts).

The semantics of the Loop language is given by a transition relation defined using Plotkin's Structured Operational Semantics (SOS) [15]. Plotkin's semantics is the conventional framework for giving operational semantics to programming languages and an alternative to denotational semantics in proving compiler correctness. A major benefit of a transition semantics over denotational semantics is that it provides a natural measure of time-complexity, namely the number of steps required to reach the final state.

The paper is organized as follows. Section 2 is devoted to the presentation of our target functional language, namely Gödel's system T with product types and a 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. Some possible applications and related topics are discussed in section 5.

2. Call-by-value system T

In this section, we give the definition of system T with product types (tuples and n-ary functions) and a constant-time predecessor operation. The rewriting system is summarized in figure 1, where $t[u_1/x_1, ..., u_n/x_n]$ denotes the usual capture-avoiding substitution. Note that the reduction rules presented here implement a weak call-by-value reduction strategy (since we do not reduce under an abstraction). We shall only consider well-typed terms, according to the type system in figure 2. The resulting system has two important properties (see [6] for instance): its terms are strongly normalizing and its rewriting rules enjoy the Church-Rosser property.

Types

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

Values

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

Terms

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

Contexts

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

Evaluation rules

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

Figure 1. Gödel's system T

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.

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

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

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

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

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

Figure 2. Type system

- $\mathbf{rec}(x_1,x_2,\lambda y.\lambda x.S(y))$ computes the sum of (x_1,x_2) by induction on x_1.
- $(\mathbf{rec}(x_1,\,\lambda y.S(y),\,\lambda u.\lambda x.\lambda y.\mathbf{rec}(y,\,(u\,S(0)),\,\lambda w.\lambda z.(u\,w)))\;\,x_2)$ computes Ackermann's function on $(x_1,\,x_2)$ (which is known not to be primitive recursive [14]).

Remark 1. Let us define the *degree* $\partial(\tau)$ of a type τ inductively by $\partial(N)=0$, $\partial(\sigma\to\tau)=max(\partial(\sigma)+1,\partial(\tau))$ and $\partial(\tau_1\times...\times\tau_n)=max(\partial(\tau_1),...,\partial(\tau_n))$ 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 $N^k\to N$ that can be represented by a term of T_0 correspond exactly to primitive recursive functions. Note that the outermost **rec** in the above definition of Ackermann's function has type $N\to N$ and thus has degree 1. Consequently this term belongs to T_1.

2.2. A derived form.

Let the expression $\mathbf{let}\,(x_1,\,...,\,x_n)=u\;\mathbf{in}\;t$ be an abbreviation for the redex $\lambda(x_1,\,...,\,x_n).t\;u$. The following derived typing rule and evaluation rule can be obtained:

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

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

Moreover, the following property holds:

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

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

3. The Loop language

The imperative language we consider is a minor syntactic variant of the Loop language defined by Meyer and Ritchie in [11]. Variables (called *registers* in [11]) may contain only natural numbers. The only atomic statements are assignments and the control structures are sequences (lists of statements separated by semi-colons) and 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, bounds b, sequences s and programs p are defined by the following grammar:*

$$\begin{array}{rcl}
e & ::= & \bar{n} \mid x \mid x+1 \mid x-1 \\
c & ::= & x := e \\
 & \mid & \textbf{for } x := 1 \textbf{ to } b \ \{s\} \\
 & \mid & \{s\} \\
b & ::= & \bar{n} \mid x \\
s & ::= & \varepsilon \mid c; s \\
p & ::= & \{s\}
\end{array}$$

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

Remark 2. The symbol ε shall be omitted in concrete examples. For instance, the empty block (which is a command) is denoted $\{\}$. Note also that for simplicity the bound of a loop can only be a variable or a constant.

Example 1. The following simple program computes the sum of (x_1, x_2) (and stores the result in variable r):

```
{
  $r := x_1$;
  for $y := 1$ to $x_2$ {
    $r := r + 1$;
  };
}
```

3.1. Operational semantics.

Following [8], 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 a 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.

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

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

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

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.

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

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

$$\langle \textbf{for } x_i := 1 \textbf{ to } e\ \{s\}, \mu\rangle \rightarrow \langle\{\textbf{for } x_i := 1 \textbf{ to } \bar{q}\ \{s\}; x_i := \overline{q+1}; s\}, \mu\rangle \tag{9}$$

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

Remark 3. 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 bound x_j is computed once at the beginning of the loop. Moreover, although the loop index variable x_i can also be assigned within the loop, these assignments do not affect the value of the variable at the beginning of the next loop iteration. The expected semantics of the for-loop (as well as its termination) is thus enforced.

Remark 4. 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** $\{s_2\}\}$ where b_2 is a fresh variable.

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

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

Remark 5. This semantics is deterministic since there is always at most one rule that may 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 show how to encode a Loop program as a λ-term of system T. Then we prove that this encoding is such that the evaluation of a Loop program runs in lock-step with the reduction of the transformed program.

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 shall 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 $\vec{x} = (x_1, ..., x_k)$ into a term is defined by induction on expressions and commands as follows:*

- $\bar{n}^\star = S^n(0)$
- $x_i^\star = x_i$
- $(x_i + 1)^\star = \textbf{succ}(x_i)$
- $(x_i - 1)^\star = \textbf{pred}(x_i)$
- $\{\}^\star = \vec{x}$
- $\{x_i := e; s\}^\star = \textbf{let } x_i = e^\star \textbf{ in } \{s\}^\star$
- $\{c; s\}^\star = \textbf{let } \vec{x} = c^\star \textbf{ in } \{s\}^\star$ *if c is not an assignment*
- $(\textbf{for } x_i := 1 \textbf{ to } e\ \{s\})^\star = \textbf{rec}(e^\star, \vec{x}, \lambda\vec{x}.\lambda x_i.\{s\}^\star)$
  where e is either a constant or a variable

Remark 6. Note that the translation clearly involves only terms of 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).

Example 2. A program p and its transform $p^\star_{\vec{x}}$ (where $\vec{x} = (x_1, b_1, b_2, i, j, k, r)$) are given below. Program p computes the division by two. We assume it receives its input in x_1 and gives the output in r.

```
{
  $r := x_1$;
  $b_1 := 1$;
  for $k := 1$ to $x_1$ {
    $b_2 := 1$;
    for $i := 1$ to $b_1$ {   // if $b_1$ then
      $r := r - 1$;
      $b_1 := 0$;
      $b_2 := 0$;
    };
    for $j := 1$ to $b_2$ {   // else
      $b_1 := 1$;
    };
  };
}
```

The transform $p^\star_{\vec{x}}$ is:

```
(
  let $r = x_1$ in
  let $b_1 = 1$ in
  let $\vec{x} = \mathbf{rec}(x_1, \vec{x}, \lambda\vec{x}.\lambda k.$
    let $b_2 = 1$ in
    let $\vec{x} = \mathbf{rec}(b_1, \vec{x}, \lambda\vec{x}.\lambda i.$
      let $r = \mathbf{pred}(r)$ in
      let $b_1 = 0$ in
      let $b_2 = 0$ in
    $\vec{x})$ in
    let $\vec{x} = \mathbf{rec}(b_2, \vec{x}, \lambda\vec{x}.\lambda j.$
      let $b_1 = 1$ in
    $\vec{x})$ in
  $\vec{x})$
)
```

4.2. Simulation theorem.

Recall that a store μ is by construction a pair $(\vec{x}, \vec{n})$ where $\vec{x}$ is a tuple of variables and $\vec{n}$ is a tuple of natural numbers (of the same length). By abuse of notation, if n is a natural number, we write $n^\star$ for $\bar{n}^\star$ and similarly if $\vec{n} = (n_1, ..., n_k)$ we write $\vec{n}^\star$ for the tuple $(\bar{n}_1^\star, ..., \bar{n}_k^\star)$. The following easy lemma states that for any expression e the reduction of $e^\star$ simulates the evaluation of p in lock-step.

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.

□

We are now able to prove the main theorem which states that for any command c the reduction of $c^\star$ simulates the evaluation of c in lock-step.

Theorem 1. *If* $\langle c, (\vec{x}, \vec{n})\rangle \rightarrow \langle c_1, (\vec{x}, \vec{n}_1)\rangle$ *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 $\langle c, (\vec{x}, \vec{n})\rangle \rightarrow \langle c_1, (\vec{x}, \vec{n}_1)\rangle$.

- Rule 3
$$(x_i := \bar{q}\,; s)^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{s\}^\star[(n_1, ..., n_{i-1}, q, n_{i+1}, ..., n_k)^\star/\vec{x}]$$
Indeed, $\{x_i := \bar{q}\,; s\}^\star[\vec{n}^\star/\vec{x}]$
$$\begin{aligned} &= (\mathbf{let}\ x_i = \bar{q}^\star\ \mathbf{in}\ \{s\}^\star)[(n_1, ..., n_k)^\star/\vec{x}] \\ &= \mathbf{let}\ x_i = \bar{q}^\star\ \mathbf{in}\ \{s\}^\star[(n_1, ..., n_{i-1}, x_i, n_{i+1}, ..., n_k)^\star/\vec{x}] \\ &\rightsquigarrow \{s\}^\star[(n_1, ..., n_{i-1}, x_i, n_{i+1}, ..., n_k)^\star/\vec{x}][\bar{q}^\star/x_i] \\ &= \{s\}^\star[(n_1, ..., n_{i-1}, q, n_{i+1}, ..., n_k)^\star/\vec{x}] \end{aligned}$$

- Rule 4
$$(x_i := x_j; s)^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{s\}^\star[(n_1, ..., n_{i-1}, n_j, n_{i+1}, ..., n_k)^\star/\vec{x}]$$
Indeed, $\{x_i := x_j; s\}^\star[\vec{n}^\star/\vec{x}]$
$$\begin{aligned} &= (\mathbf{let}\ x_i = x_j^\star\ \mathbf{in}\ \{s\}^\star)[(n_1, ..., n_k)^\star/\vec{x}] \\ &= \mathbf{let}\ x_i = \vec{n}_j^\star\ \mathbf{in}\ \{s\}^\star[(n_1, ..., n_{i-1}, x_i, n_{i+1}, ..., n_k)^\star/\vec{x}] \\ &\rightsquigarrow \{s\}^\star[(n_1, ..., n_{i-1}, x_i, n_{i+1}, ..., n_k)^\star/\vec{x}][n_j^\star/x_i] \\ &= \{s\}^\star[(n_1, ..., n_{i-1}, n_j, n_{i+1}, ..., n_k)^\star/\vec{x}] \end{aligned}$$

- Rule 5
$$\frac{e^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow q}{\{x_i := e; s\}^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{x_i := \bar{q}\,; s\}^\star[\vec{n}^\star/\vec{x}]}$$
Indeed, $\{x_i := e; s\}^\star[\vec{n}^\star/\vec{x}]$
$$\begin{aligned} &= (\mathbf{let}\ x_i = e^\star\ \mathbf{in}\ \{s\}^\star)[(n_1, ..., n_k)^\star/\vec{x}] \\ &= \mathbf{let}\ x_i = e^\star[\vec{n}^\star/\vec{x}]\ \mathbf{in}\ (\{s\}^\star[(n_1, ..., n_{i-1}, x_i, n_{i+1}, ..., n_k)^\star/\vec{x}]) \end{aligned}$$
By induction hypothesis, $e^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow q$ and then by lemma 1:
$\mathbf{let}\ x_i = e^\star[\vec{n}^\star/\vec{x}]\ \mathbf{in}\ (\{s\}^\star[(n_1, ..., n_{i-1}, x_i, n_{i+1}, ..., n_k)^\star/\vec{x}])$
$$\begin{aligned} &\rightsquigarrow \mathbf{let}\ x_i = q^\star\ \mathbf{in}\ (\{s\}^\star[(n_1, ..., n_{i-1}, x_i, n_{i+1}, ..., n_k)^\star/\vec{x}]) \\ &= (\mathbf{let}\ x_i = q^\star\ \mathbf{in}\ \{s\}^\star)[(n_1, ..., n_k)^\star/\vec{x}] \\ &= \{x_i := q; s\}^\star[\vec{n}^\star/\vec{x}] \end{aligned}$$

- Rule 6
$$\{\{\}; s\}^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{s\}^\star[\vec{n}^\star/\vec{x}]$$
Indeed, $\mathbf{let}\ \vec{x} = \{\}^\star\ \mathbf{in}\ \{s\}^\star[\vec{n}^\star/\vec{x}]$
$$\begin{aligned} &= \mathbf{let}\ \vec{x} = \vec{x}\ \mathbf{in}\ \{s\}^\star[\vec{n}^\star/\vec{x}] \\ &= \mathbf{let}\ \vec{x} = \vec{n}^\star\ \mathbf{in}\ \{s\}^\star \\ &\rightsquigarrow \{s\}^\star[\vec{n}^\star/\vec{x}] \end{aligned}$$

- Rule 7
$$\frac{c^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow c_1^\star[\vec{n}_1^\star/\vec{x}]}{\{c; s\}^\star[\vec{n}^\star/\vec{x}] \rightsquigarrow \{c_1; s\}^\star[\vec{n}_1^\star/\vec{x}]}$$

Indeed, since c is neither an assignment nor the empty block,

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

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:

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

- 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 $n_j = 0$:

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

- Rule 9

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

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

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

□

5. Possible applications and related topics

We have already mentioned (see remark 6) that programs of the Loop language are actually mapped by our translation to λ-terms of T_0 (we do not exploit the full hierarchy of system T). On the other hand, in our system T_0 recursors with type N^k are allowed. Clearly, we can construct an effective translation of T_0 into the representations of the primitive recursive functions in which simultaneous primitive recursion is allowed, such that a recursor with depth n is always mapped into simultaneous recursion with the same depth n. Thus, our compilation scheme is consistent with the intuition that Loop programs naturally correspond with simultaneous recursion. More formally, if $\{K_n\}_{n\geqslant 0}$ is Heinermann-Axt's hierarchy [2, 9] (which is based on the "nesting depth" of primitive recursions) and $\{K_n'\}_{n\geqslant 0}$ is obtained by allowing the more general simultaneous primitive recursion, it is known that the hierarchies $\{L_n\}_{n\geqslant 0}$ and $\{K_n'\}_{n\geqslant 0}$ coincide (whereas $L_n = K_n$ only for $n \geqslant 4$). Besides, remember that the Heinermann-Axt's hierarchy is related to the Grzegorczyk hierarchy: $\mathcal{E}_{n+1} = K_n$ for $n \geqslant 2$ (see [17] and [12]).

Kristiansen and Niggl [10, 13] recently improved these results. They defined finer syntactic measures, the μ-measures, for both the Loop language and a subsystem of Gödel system T called PR_1 which features only ground type variables and ground type recursion. The μ-measure for PR_1 allows to distinguish between "top recursion" (which may increase complexity) and "side recursion" (which is harmless). They showed that programs of μ-measure $n \geqslant 1$ compute exactly the functions in Grzegorczyk class $\mathcal{E}_{n+1}$ (recall that $\mathcal{E}_2$ corresponds to FLINSPACE and $\mathcal{E}_3$ is the class of Csillag-Kalmar elementary functions). Similarly, for a different notion of μ-measure (adapted for imperative programs), they proved that Loop programs of μ-measure $n \geqslant 0$ compute exactly the functions in $\mathcal{E}_{n+2}$.

A subject for future work is to generalize the μ-measure for PR_1 to account for product types. One would thus get a new measure μ' for a Loop program π defined by $\mu'(\pi) = \mu(\pi^\star)$. We believe that it is worth investigating how this new measure would be related to Kristiansen and Niggl's orginal μ-measure for Loop programs.

As another application, the simulation theorem enables us to derive properties concerning Loop programs from previously known results about system T. An example of such result is related to the so-called *minimum problem*: there is no Loop program that computes the minimum of two natural numbers n and m in time *O(min(n,m))*. Indeed, Colson and Fredholm [4] proved that in call-by-value system T, any algorithm which computes a non-trivial binary function (where trivial means constant or projection plus constant), has a time-complexity which is at least linear in one of the inputs. Note that, as suggested by a referee, this result can easily be proved directly for the Loop language. However, we intend to generalize our translation to an extension of the Loop language with higher-order procedures and procedural variables. We conjecture that such an imperative language is as expressive as full system T, and our translation may turn out to be a convenient tool in this higher-order framework.

Acknowledgments.

We would like to express our gratitude to L. Colson for helpful comments on an earlier version of this paper and to A. Durand and E. Polonowski for carefully proofreading this paper. We would like also to thank an anomynous referee for useful suggestions.

Bibliography

[1] J. Avigad and S. Feferman. Gödel's functional ("dialectica") interpretation. In S. R. Buss, editor, *Handbook of Proof Theory*, pages 337–405. Elsevier Science Publishers, Amsterdam, 1998.

[2] P. Axt. Iteration of primitive recursion. *Z. Math. Logik Grundlagen Math.*, 11:253–255, 1965.

[3] C. Calude. *Theories of computational complexity*. Elsevier Science Inc., New York, NY, USA, 1988.

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

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

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

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

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

[9] W. Heinermann. *Uber die Rekursionszahlen rekursiver Funktionen*. PhD thesis, Münster, 1961.

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

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

[12] H. Müller. *Klassifizierungen der primitiv rekursiven Funktionen.* PhD thesis, Münster, 1974.

[13] K.-H. Niggl. The μ-measure as a tool for classifying computational complexity. *Archive for Mathematical Logic*, 39:515–539, 2000.

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

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

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

[17] H. Schwichtenberg. Rekursionszahlen und die grzegorczyk-hierarchie. *Archiv für Mathematische Logik und Grundlagenforschung*, 12:85–97, 1969.