Chapter 1

A total functional programming language that computes APRA

David Michel and **Pierre Valarcher**
LACL, EA 4219 Université Paris-Est Créteil
IUT de Sénart-Fontainebleau
route forestière Hurtault
F-77300 Fontainebleau
david.michel@univ-paris12.fr
pierre.valarcher@univ-paris-est.fr

We recall the definition of the class of primitive recursive algorithms (APRA) and we prove that there exists a functional programming language (primitive recursion with variable parameters or fragment of system T_1 of Gödel) that simulates all algorithms of APRA in lock-step (one step is simulated by a constant number of steps). The class APRA is a large class of algorithms defined from abstract state machine (ASM) of Y. Gurevich. The functional language admits a mixed-reduction strategy to be efficient and the simulation through an intermediate language with a bounded-conditional loop.

1. Introduction

We propose to describe a functional programming language that simulates a class of algorithms defined in [1]. We first describe the context of the paper giving intuition of notions of algorithms and the problematic related on.

(a) Algorithms.

There is a growing interest in finding a **formal definition** for *algorithms*. The first definition was proposed by Y. Gurevich in[2], [3] and [4]; an axiomatic definition is mapped to the notion of *abstract*

state machine with a *strict lock-step* simulation (see [5] for a definition of simulation and strict lock-step*). The abstract state machines are a kind of super Turing machine that work not on simple tape (with finitary alphabets) but on multi–sorted algebras (see the point of view in [6]). A program is a finite set of rules that updates terms. For instance, if we consider an algebra with symbols $(res, x, y, <, -)$ then the following program is an algorithm for a *gcd* function:

if x=0 **then** res := y
if ¬(x=0) ∧ y=0 **then** res := x
if ¬(x=0) ∧ ¬(y=0) ∧ (x<y) **then** y := y-x
if ¬(x=0) ∧ ¬(y=0) ∧ ¬(x<y) **then** x := x - y

The second approach, due to Y. Moschovakis [7], 8] and [27]), considers that algorithms are expressed by a definition by mutual recursion and the algorithmic content of a such definition A is given by the semantics of equations (given by the recursor of a special form of A). With the algebra containing $(-, <)$, the following equation represents an algorithm for the *gcd* function, $\mathbf{gcd}(x, y) =$

$$\mathbf{if}(x = 0, y, \mathbf{if}(y = 0, x, \mathbf{if}(x < y, \mathbf{gcd}(x, y - x), \mathbf{gcd}(x - y, y))))$$

(b) Algorithmic weakness.

On the other side, a whole field of research is devoted to study the algorithmic power of some programming languages (essentially total programming languages matching well known class of total functions, see [10], [12],[13][14] [15], 16],[17] and [1,18,19,20] for main results). All approaches (except in [1]) consider the *lacuna* of different languages: for instance, the *minimum* problem is such a well known *lacuna* (it refers to the fact that the usual algorithm that decrements **alternatively** its two inputs and **stops** when it achieves to read one of its inputs cannot be simulated by some programming languages; the number of steps to find the minimum is proportional to the smaller inputs but, in many programming languages this *algorithm* is not representable with the same complexity).

(c) The class of primitive recursive algorithms.

A more *top-bottom* approach is to consider a class of algorithms (thanks to the formal definition of algorithms by Y. Gurevich) and prove

* The algorithms allows to update simultaneously a finite amount of data.

that a programming language can simulate in lock–step all algorithms of the class. The second author defined a class of algorithms for primitive recursive functions in [1]; it is proved that this class of algorithms (named $APRA$) can be represented by an extension of the imperative *Loop* languages ([22]) in *lock-step* (not in *strict lock-step*). The $APRA$ algorithms are all couples (A, c) with A a simple arithmetic abstract state machine: the domains are integer and boolean, structures are composed of 0-ary dynamic functions (variables) and the static unary functions $x \to x+1, x \to x-1$ (successor and predecessor) and c is any primitive recursive functions (it represents the length of the run of A).

(d) Link imperative and functional programming languages.

Following [20] and [18], we extract from the system T of Gödel, the core of the functional language that simulates the class $APRA$. The fragment of system T which captures $APRA$ is exactly the language induced by the definition of *primitive recursion with variable parameters* (see [21]) with a mixed call-by strategy (by value on β–reduction and by-name for rec and if reductions).

(e) Organization of the paper.

The paper is organized as follows : the section 2 is devoted to the presentation of a language *Loop* extended with boolean type, boolean expression, conditional and escape statement or bounded-conditional loops. Section 3 presents the functional language of system T and section 4 is devoted to the main result. We conclude by linking $APRA$ and an equational presentation of algorithms *à la* Moschovakis.

2 *Loop*'s languages.

2.1 The *Loop* language.

We consider an extension of the *Loop* language defined in [22] with a syntax closest to today programming language. The following definitions [20], but bounded loops are written *downto* instead of *up to*. The language has been proved to be extensionaly computing *primitive recursive functions*. We add *boolean* and *integer* types.

Definition. *The syntax is defined by induction as follows:*

$$\begin{array}{ll} e & ::= e_{int} \mid e_{bool} \\ e_{int} & ::= x \mid x\text{-}1 \mid x\text{+}1 \mid \bar{n} \\ e_{bool} & ::= \neg b \mid \textbf{true} \mid \textbf{false} \end{array}$$

$$
\begin{array}{rl}
c & ::= x{:=}e \\
 & \mid \quad \textbf{if } b\ \{s\}\ \textbf{else}\ \{s\} \\
 & \mid \quad \textbf{for } x\ \textbf{in } b'\ \textbf{downto } 1\ \{s\} \\
 & \mid \quad \{s\}
\end{array}
$$

$$
\begin{array}{rl}
b & ::= \textbf{true} \mid \textbf{false} \mid \ x \\
b' & ::= \bar{n} \mid x \\
s & ::= \epsilon \mid c;s \\
p & ::= \{s\}
\end{array}
$$

The command **if** $e\ \{s_1\}$ *is equivalent to the command* **if** $e\ \{s_1\}$ **else** $\{\}$.

We use a *down to* bounded iteration rather than a usual bounded iteration. But, we always can transform an increasing iteration onto a decreasing iteration with a constant cost execution. For example **for** x_i **in** 1 **to** $e\ \{s\}$ is replaced by

$$x_{new} := 1;\ \textbf{for } x_i \textbf{ in } e \textbf{ downto } 1\ \{s[x_{new}/x_i];\ x_{new} := x_{new} + 1;\}$$

where x_{new} is a fresh variable and $s[x/y]$ (x, y variables) stands for the substitution of y by x in s. We have that x_i does not appear is $s[x_{new}/x_i]$.

The operational semantics of *Loop* is given by a simple transition system which is described by a set of rewriting rules over configurations. A configuration is either a value, a pair $\langle e, \mu\rangle$ (respectively $\langle c, \mu\rangle$) consisting on an expression e (respectively a command c) and a store μ. A store maps variable to natural numbers or boolean values in the obvious way, and we write $\mu(x)$ for the value of x in the store μ, and $\mu[x := q]$ for an update of the value of x in store μ.

Operational semantics.

1 – Expressions:

$$
\begin{array}{ll}
\langle x_i + 1, \mu\rangle & \triangleright\ \mu(x_i) + 1 \\
\langle x_i - 1, \mu\rangle & \triangleright\ \mu(x_i) \dot{-} 1 \\
\langle \neg b, \mu\rangle & \triangleright\ \neg\mu(b)
\end{array}
$$

2 – Assignments: q denotes a natural integer or a boolean value.

$$
\begin{array}{ll}
\langle \{x_i := \bar{q}; s\}, \mu\rangle & \triangleright\ \langle \{s\}, \mu[x_i := q]\rangle \\
\langle \{x_i := x_j; s\}, \mu\rangle & \triangleright\ \langle \{s\}, \mu[x_i := \mu(x_j)]\rangle \\
\langle \{x_i := e; s\}, \mu\rangle & \triangleright\ \langle \{x_i := \bar{q}; s\}, \mu\rangle \quad \text{if } \langle e, \mu\rangle \triangleright q
\end{array}
$$

where e is neither a constant nor a variable.

3 – Empty block: $\langle\{\{\};s\},\mu\rangle \triangleright \langle\{s\},\mu\rangle$.

4 – Sequence: $\langle\{c;s\},\mu\rangle \triangleright \langle\{c_1;s\},\mu_1\rangle$ if $\langle c,\mu\rangle \triangleright \langle c_1,\mu_1\rangle$ where c is neither the empty block nor an assignment.

5 – Conditional:

5a. $\langle$**if** e $\{s_1\}$ **else** $\{s_2\},\mu\rangle \triangleright \langle\{s_1\},\mu\rangle$ where e is either the constant **true** or a variable x_j and then $\mu(x_j) =$ **true**.

5b. $\langle$**if** e $\{s_1\}$ **else** $\{s_2\},\mu\rangle \triangleright \langle\{s_2\},\mu\rangle$ where e is either the constant **false** or a variable x_j and then $\mu(x_j) =$ **false**.

6 – Bounded down iteration:

6a. $\langle$**for** x_i **in** e **downto** 1 $\{s\},\mu\rangle \triangleright \langle\{\},\mu\rangle$ where e is either the constant 0 or a variable x_j and then $\mu(x_j) = 0$.

6b. And $\langle$**for** x_i **in** e **downto** 1 $\{s\},\mu\rangle \triangleright$

$$\langle\{x_i := \overline{q+1}; \{s\};\ \ \textbf{for } x_i \textbf{ in } \overline{q} \textbf{ downto } 1\ \{s\}\},\mu\rangle$$

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

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

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

The semantics is deterministic and the only case where no rule can be applied corresponds to the final configuration (when the program amounts to an empty block).

2.2 The *LoopE* language.

The *LoopE* language extends the *Loop* language with an *escape* statement allowing to stops the computation anywhere in the program.

Definition.

$$c ::= \ldots \ \mid \textbf{exit when } \ b$$

We give two equivalent operational semantics. The first one is a simple extension of the previous with the rules

First semantics: Exit1.

$$\langle\{\textbf{exit } \textbf{ when } b; s\},\mu\rangle \triangleright \langle\{\},\mu\rangle$$

where b is the constant **true** or a variable x_j and then $\mu(x_j) = \mathbf{true}$.

$$\langle \{\mathbf{exit} \quad \mathbf{when}\ b; s\}, \mu\rangle \triangleright \langle \{s\}, \mu\rangle$$

where b is the constant **false** or a variable x_j and then $\mu(x_j) = \mathbf{false}$.

The second one is a transformation of the previous. We consider now the environment as a couple (α, μ) where μ is the previous notion of environment and $\alpha \in \{true, false\}$. Then in all previous rules we substitute μ by $(true, \mu)$. And then

Second semantics: Exit2.

$$\langle \{\mathbf{exit} \quad \mathbf{when}\ b; s\}, (true, \mu)\rangle \triangleright \langle \{\}, (false, \mu)\rangle$$

where e is the constant **true** or a variable x_j and then $\mu(x_j) = \mathbf{true}$.

$$\langle \{\mathbf{exit} \quad \mathbf{when}\ b; s\}, (true, \mu)\rangle \triangleright \langle \{s\}, (true, \mu)\rangle$$

where b is the constant **false** or a variable x_j and then $\mu(x_j) = \mathbf{false}$.

Remark. We can allways simulate the **exit**; statement of the initial paper [1] by **exit when true**; for instance.

Remark. The semantics remains deterministic.

Remark. There is no rules in the second semantics for $\langle s, (false, \mu)\rangle$.

We define, as in [23], by $Loop_0$ the set of programs with no $< for >$ command and $Loop_{n+1}$ the set of programs containing the command **for** x **in** b **downto** 1 $\{s\}$ with $s \in Loop_n$. We say that c is a $Loop_n$–command: a command which is not under the scope of any **for** command is said to be a $Loop_0$–command. $Loop_{n+1}$–command is a command under the scope of a $n+1$ depth-nested **for** command.

2.3 The *LoopC* language.

We extend the *Loop* language with a more *sophisticated* **for** statement.

$$\text{c} ::= ... \mid \mathbf{for}\ x_i\ \mathbf{in}\ b'\ \mathbf{downto}\ 1\ \mathbf{onlyIf}\ b\ \{s\}$$

The operational semantics of this command is

Conditional for.

$$\langle \textbf{for } x_i \textbf{ in } e \textbf{ downto } 1 \textbf{ onlyIf } b\ \{s\}, \mu\rangle \triangleright \langle\{\}, \mu\rangle$$

where e is either the constant 0 or a variable x_j and then $\mu(x_j) = 0$ or where b is the constant **false** or a variable x_k and then $\mu(x_k) = \textbf{false}$.

$$\langle \textbf{for } x_i \textbf{ in } e \textbf{ downto } 1 \textbf{ onlyIf } b\ \{s\}, \mu\rangle$$

$$\triangleright \langle \textbf{if } b\ \{x_i := \overline{q+1}; \{s\};\ \ \textbf{for } x_i \textbf{ in } \overline{q} \textbf{ downto } 1 \textbf{ onlyIf } b\ \{s\}\}$$
$$\textbf{else } \{x_i := \overline{q+1};\}, \mu\rangle$$

where e is either the constant $q+1$ or a variable x_j and then $\mu(x_j) = q+1$ and where b is either the constant **true** or a variable x_k and then $\mu(x_k) = \textbf{true}$.
Remark. With this new statement, we can encode the predicate $eq0(x)$ in *constant*–step reductions:

$$b := \textbf{true};\ \textbf{for } I \textbf{ in } x \textbf{ downto } 1 \textbf{ onlyIf } b\ \{b := \textbf{false};\}$$

This is also the case with the **exit when** statement, but the step side of the solution (the end of the program) does not appear to be used locally.

Of course, we can simulate the **exit when** statement with conditional loop. But the following translation does not preserve the numbers of reductions: we can not escape form nested loops in a constant step reduction, we must go back through all loops.

Definition. *The translation * of a LoopE program into a LoopC program is defined by induction on commands as follows where w is a fresh variable (expressions are mapped to themselves) :*

1- $(\textbf{exit}\ \ \textbf{when } u)^* = \{\textbf{if } u\ \{w := false;\}\}$
2- $(\textbf{if } e\ \{s_1\} \textbf{ else } \{s_2\})^* = \textbf{if } e\ \{s_1\}^* \textbf{ else } \{s_2\}^*$
3- $(\textbf{for } x_i \textbf{ in } e \textbf{ downto } 1\ \{s\})^* = \textbf{for } x_i \textbf{ in } e \textbf{ downto } 1 \textbf{ onlyIf } \neg w\ \{s\}^*$
4- $\{x := e; s\}^* = \{x := e; s^*\}$
5- $\{c;\ s\}^* = \{c^*;\ \textbf{if } w\ \{s\}^*\}$ when c is not an assignment.
and then $\langle p, (true, \mu)\rangle^* = \langle p^*, \mu \oplus \{w, true\}\rangle$.

Remark. We need only one variable to encode the escape statement. We don't memorize the specific condition of the escape. Of course, it may be of some interests to memorize the condition of the escape to adapt the behaviors of the program.

We extend, trivially, the definition of $LoopE_n$ and $LoopC_n$ (resp. $LoopE_n$–command and $LoopC_n$–command) form $Loop_n$(resp. $Loop_n$–command). Then the transformation doesn't modify the depth of a command:

Lemma. *If $p \in LoopE_n$ then $p^* \in LoopC_n$.*

From the translation, we have that if an **exit when** u statement is raised (*i.e.* u is $true$) as a $LoopC_n$–command, the simulation needs to execute at most one assignment and $2n$ conditionals:

Lemma. *Let* (**exit when** u) *be a $LoopE_n$–command then*

$$\langle\{\textbf{exit when } u; s\}, (true, \mu)\rangle^* \rightarrow^{2n+1} \langle\{\}, \mu \oplus \{w, false\}\rangle$$

with $u = true$.

From now the simulation is *lock-step* (not strict) and do not depend of the inputs:

Proposition. *If $pe \in LoopE_n$ and $\langle pe, (true, \mu)\rangle \rightarrow^m \langle\{\}, (\alpha, \mu')\rangle$ then there exists $pc \in LoopC$ such that $\langle pc, \mu \oplus \{w, true\}\rangle \rightarrow^{O(m)} \langle\{\}, \mu' \oplus \{w, \alpha\}\rangle$ where α is $true$ or $false$ depending of the fact that the last command executed in the execution of $\langle pe, \mu\rangle$ is a* **exit when** *statement or not.*

The GCD sample. Let `GCDcore` be the following program. It comes from the *gcd* program of the *ASM* of section 1 (see [1] for an explanation of the transformation from *ASM* program to *Loop* program). For sake of simplicity, we preserve the function *substraction* $(-)$ and the predicate *lower than* $(<)$, but clearly they may be encoded in the *LoopC* language:

```
if (x = 0) {
   res := y;
   stop := true;
} else {
        if (y = 0) {
           res := x;
           stop := true;
        } else {
                if (x < y) {
                   y := y - x;
                } else { /* x >= y */
                        x := x - y;
                }
        }
}
```

Now let $add(N, M)$ be the function that sums N and M, then by inserting `GCDcore` after each command (the principle is that `GCDcore` must run enough times to be computed; this is done by inserting the core program, one that represents the program of the ASM, in a program that implements a higher lower bound of the algorithm) we obtain a *LoopC* program that computes the *gcd* function with the expected complexity:

```
stop := false;
X := 0;
GCDcore;
for I in N downto 1 onlyIf !stop {
      X := X + 1;
      GCDcore;
}
if !stop {
   for J in M downto 1 onlyIf !stop {
        X := X + 1;
        GCDcore;
   }
}
```

3. The system T language

The system T language is an extension of the usual primitive recursive with variable parameters ([21]). We present it as a λ–calculus with integer types and a recursor.

Primitive recursive functionals of finite types, also known as Gödel system T in logic (see [24]), and as typed lambda calculus with primitive recursive recursion in higher types in computer science, are quite important in both areas. In the former they are used in proof-theoretical studies of Peano arithmetic, in the latter they can be used to give the formal semantics of some modern kind of (functional) computer programming languages.

3.1 Types and syntax.

We give the definition of system T with product types (tuples and n-ary functions), a constant-time predecessor operation and a conditional:

Definition. *We define types and typed-terms by induction:*

Types are defined inductively by: B is the type of booleans, N is the type of natural numbers, and if U and V are types then are $U \to V$ and $U \times V$;

Terms $t : T$ are defined inductively

- for each types T, the set of typed variables $x^T : T, y^T : T, \ldots$ are typed–terms,
- $true : B, false : B$ are typed–terms,
- $0 : N$ is a typed–term,
- $S(t) : N$ and $pred(t) : N$ are typed–terms if $t : N$ is a typed–term,
- $(t_1^{T_1}, \cdots, t_n^{T_n}) : (T_1 \times \cdots \times T_n)$ is a typed–term if $t_1^{T_1}, \cdots, t_n^{T_n}$ are typed–terms,
- $\lambda(x_1^{U_1}, \cdots, x_n^{U_n}).t : (U_1 \times \cdots \times U_n) \to T$ is a typed–term if $t : T$ is a typed–term,
- $t\ u : T$ is a typed–term if $t : U \to T$ et $u : U$ are typed–terms,
- $if_T(c, u_1, u_2) : T$ is a typed–term if $c : B$ and $u_1, u_2 : T$ are typed–terms,
- $rec_T(t,\ b,\ [x^N, y^T]\ s) : T$ is a typed–term if $t : N$, $b, s : T$ are typed–terms.

We choose a special syntax for the *rec* that has been, first, introduced by L. Colson in [25, 11]. Despite the theoretical (philosophical) point of view* this presentation allows us to consider, easily, two different strategies of reductions : one for the β–reduction and one for the *rec*.

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

$$\partial(B) = \partial(N) = 0, \partial(T \to U) = \max(\partial(T) + 1, \partial(U))$$

and

$$\partial(T_1 \times \cdots \times T_n) = max(\partial(T_1), \ldots, \partial(T_n))$$

We now define the degree $d(t)$ of a term t as the maximum degree of types T such that 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.

---

* As notice by L. Colson, in the usual presentation of system T, to deduce that $\Gamma \vdash rec_T(n, b, s) : N$ we need the three hypothesis $\Gamma \vdash n : N$, $\Gamma \vdash b : T$ and $\Gamma \vdash s : N \to T \to T$ then we need the $\to$ to define the rule for recursion. In our case, the rule is $\Gamma, x : N, y : T \vdash s : T$ no arrow appears in the rule.

Remark. Let denote by $let\ (x_1, \cdots, x_n) = u\ in\ t$ the redex:

$$(\lambda(x_1, \cdots, x_n).t)\ u.$$

3.2 Mixed call–by reductions.

The substitution denoted by $t[u/x]$ is defined as usually. We define two reductions of one step $\rightarrow_\beta$ and $\rightarrow$. The reduction rules presented implement the weak call-by-value reduction for the λ–application and call-by-name reduction for rec and if:

Values:

$$\text{v} ::= x \mid 0 \mid S(v) \mid true \mid false \mid \lambda(x_1, \ldots, x_n).t$$

Contexts:

$$\begin{array}{rl} C[\,] ::= & [] \\ \mid & C[]t \\ \mid & vC[] \\ \mid & S(C[]) \\ \mid & pred(C[]) \\ \mid & if(C[], t_2, t_3) \\ \mid & rec(C[], t_2, [x, y]t_3) \\ \mid & (v_1, \ldots, v_{i-1}, C[], t_{i+1}, \ldots, t_n) \end{array}$$

Evaluation rules:

$$\begin{array}{lll} C[\lambda(x_1, \cdots, x_n).t\ \ (v_1, \cdots, v_n)] & \rightarrow_\beta & t[(v_1, \cdots, v_n)/(x_1, \cdots, x_n)] \\ C[pred(0)] & \rightarrow_\beta & C[0] \\ C[pred(S(v))] & \rightarrow_\beta & C[v] \\ C[rec(0, t_2, [x, y]t_3)] & \rightarrow & C[t_2] \\ C[rec(S(v_1), t_2, [x, y]t_3)] & \rightarrow & C[t_3[v_1/x, rec(v_1, t_2, [x, y]t_3)/y]] \\ C[if(true, t_2, t_3)] & \rightarrow & C[t_2] \\ C[if(false, t_2, t_3)] & \rightarrow & C[t_3] \end{array}$$

4. Lock-step simulation.

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. *The translation* $^+$ *of LoopC program with variables* $\vec{x} = (x_1, \ldots, x_n)$ *into a term is defined by induction on expression and command as follows:*

$$\overline{n}^+ = S^n(0)$$
$$\overline{true}^+ = true \text{ and } \overline{false}^+ = false$$
$$x_i^+ = x_i$$
$$(x_i + 1)^+ = succ(x_i)$$
$$(x_i - 1)^+ = pred(x_i)$$
$$\{\}^+ = \vec{x}$$
$$\{x_i = e;\ s\}^+ = let\ x_i\ =\ e^+\ in\ \{s\}^+$$
$$\{c;\ s\}^+ = let\ \vec{x}\ =\ c^+\ in\ \{s\}^+ \text{ if } c \text{ is not an assignment}$$
$$(\textbf{if } b\ \{s_1\} \textbf{ else } \{s_2\})^+ = if(b^+, \{s_1\}^+, \{s_2\}^+)$$
$$(\textbf{for } x_i \textbf{ in } e \textbf{ downto } 1 \textbf{ onlyIf } b\ \{s\})^+ =$$

$$rec_{N\to N}(e^+, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.if(b^+, let\ \vec{x} = \{s\}^+\ in\ y\ \vec{x},\ \vec{x}))\ \ \vec{x}$$

The following lemma states that the translation preserves the lock-step simulation of expression:

Lemma. *If* $\langle e, (\vec{x}, \vec{n})\rangle \triangleright q$ *then* $e^+[\vec{n}^+/\vec{x}] \to q^+$.

Lemma. *If* $w = S^n(0)$ *then*

$$let\ x = t[w/x]\ in\ u\ \equiv\ let\ x = w\ in\ let\ x = t\ in\ u$$

The following theorem states that the simulation is not a *strict* lock step simulation.

Theorem. *If* $\langle c, (\vec{x}, \vec{n})\rangle \triangleright \langle c_1, (\vec{x}, \vec{n_1})\rangle$ *then* $c^+[\vec{n}^+/\vec{x}] \to^{1\ or\ 2} c_1^+[\vec{n_1}^+/\vec{x}]$.

Proof. By induction on the derivation of $\langle c, (\vec{x}, \vec{n})\rangle \triangleright \langle c_1, (\vec{x}, \vec{n_1})\rangle$.

- If $\langle \{x_i := \overline{q}; s\}, \mu\rangle \triangleright \langle \{s\}, \mu[x_i := q]\rangle$ then $\{x_i := \overline{q}; s\}^+[\vec{n}^+/\vec{x}]$

$$\begin{aligned}
&= (let\ \ x_i\ \ =\ \ \overline{q}^+\ \ in\ \ \{s\}^+)\ \ \ [(n_1, \ldots, n_k)^+/\vec{x}]\\
&= let\ x_i = \overline{q}^+\ in\ \{s\}^+[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^+/\vec{x}]\\
&\to \{s\}^+[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^+/\vec{x}][\overline{q}^+/x_i]\\
&= \{s\}^+[(n_1, \ldots, n_{i-1}, \overline{q}, n_{i+1}, \ldots, n_k)^+/\vec{x}]
\end{aligned}$$

- If $\langle\{x_i := x_j; s\}, \mu\rangle \triangleright \langle\{s\}, \mu[x_i := \mu(x_j)]\rangle$ then $\{x_i := x_j; s\}^+[\vec{n}^+/\vec{x}]$

$$\begin{array}{l} = \ let\ x_i = x_j^+\ in\ \{s\}^+)\ [(n_1, \ldots, n_k)^+/\vec{x}] \\ = \ let\ x_i = n_j^+\ in\ \{s\}^+[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^+/\vec{x}] \\ \rightarrow \{s\}^+[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^+/\vec{x}][n_j^+/x_i] \\ = \ \{s\}^+[(n_1, \ldots, n_{i-1}, n_j, n_{i+1}, \ldots, n_k)^+/\vec{x}] \end{array}$$

- If $\langle e, \mu\rangle \triangleright \overline{q}$ and then $\langle\{x_i := e; s\}, \mu\rangle \triangleright \langle\{x_i := \overline{q}; s\}, \mu\rangle$ then $\{x_i := e; s\}^+[\vec{n}^+/\vec{x}]$

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

By induction hypothesis $e^+[\vec{n}^+/\vec{x}] \rightarrow q$ and by β–reduction (call-by-value):

$$\begin{array}{l} \rightarrow let \quad x_i = q^+ \quad in \quad (\{s\}^+[(n_1, \ldots, n_{i-1}, x_i, n_{i+1}, \ldots, n_k)^+/\vec{x}]) \\ = \ (let\ x_i = q^+\ in\ \{s\}^+)[\vec{n}^+/\vec{x}] \\ = \ \{x_i := \overline{q}; s\}^+[\vec{n}^+/\vec{x}] \end{array}$$

- If $\langle\{\{\}; s\}, \mu\rangle \triangleright \langle\{s\}, \mu\rangle$ then $\{\{\}; s\}^+[\vec{n}^+/\vec{x}]$

$$\begin{array}{l} = \ (let\ \vec{x} = \{\}^+ in\ \{s\}^+)[\vec{n}^+/\vec{x}] \\ = \ let\ \vec{x} = \vec{n}\ in\ \{s\} \\ \rightarrow \{s\}^+[\vec{n}^+/\vec{x}] \end{array}$$

- If $\langle c, \mu\rangle \triangleright \langle c_1, \mu_1\rangle$ and then $\langle\{c; s\}, \mu\rangle \triangleright \langle\{c_1; s\}, \mu_1\rangle$ then $\{c; s\}^+[\vec{n}^+/\vec{x}]$

$$\begin{array}{l} = (let\ \vec{x} = c^+ in\ \{s\}^+)[\vec{n}^+/\vec{x}] \\ = let\ \ \vec{x}\ =\ c^+[\vec{n}^+/\vec{x}]\ \ in\ \ \{s\} \end{array}$$

By induction hypothesis $c^+[\vec{n}^+/\vec{x}] \rightarrow^2 c_1^+[\vec{n}^+/\vec{x}]$ and by β–reduction (call-by-value):

$$\begin{array}{l} \rightarrow^2 \ let\ \vec{x} = c_1^+[\vec{n_1}^+/\vec{x}]\ in\ \{s\} \\ = \quad (let\ \vec{x} = c_1^+\ in\ \{s\})[\vec{n_1}^+/\vec{x}] \\ = \quad \{c_1;\ s\}^+[\vec{n_1}^+/\vec{x}] \end{array}$$

- If $\langle$**if** $e\ \{s_1\}$ **else** $\{s_2\}, \mu\rangle \triangleright \langle\{s_1\}, \mu\rangle$ where e is either the constant $true$ or a variable x_j and then $n_j = true$ then (**if** $e\ \{s_1\}$ **else** $\{s_2\})^+[\vec{n}^+/\vec{x}]$

$$\begin{array}{l} = \ if(e^+, \{s_1\}^+, \{s_2\}^+)[\vec{n}^+/\vec{x}] \\ = \ if(e^+[\vec{n}^+/\vec{x}], \{s_1\}^+[\vec{n}^+/\vec{x}], \{s_2\}^+[\vec{n}^+/\vec{x}]) \\ = \ if(true, \{s_1\}^+[\vec{n}^+/\vec{x}], \{s_2\}^+[\vec{n}^+/\vec{x}])[\vec{n}^+/\vec{x}] \\ \rightarrow \{s_1\}^+[\vec{n}^+/\vec{x}] \end{array}$$

- same with s_2 instead of s_1 for when e is either the constant $false$ or a variable x_j and then $n_j = false$
- If $\langle$**for** x_i **in** e **downto** 1 **onlyIf** x_b $\{s\}, \mu\rangle \triangleright \langle\{\}, \mu\rangle$ where e is either the constant 0 or a variable x_j and then $n_j = 0$ then (**for** x_i **in** e **downto** 1 **onlyIf** x_b $\{s\}$)$^+[\vec{n}^+/\vec{x}]$

$= (rec(e^+, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.if(x_b^+, let\ \vec{x} = \{s\}^+\ in\ y\ \vec{x},\ \vec{x}))\ \ \vec{x})\ \ [\vec{n}/\vec{x}]$
$= rec(0, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.if(x_b^+, let\ \vec{x} = \{s\}^+\ in\ y\ \vec{x},\ \vec{x}))\ \ \vec{n})$
$\rightarrow_{rec}\ (\lambda\vec{x}.\vec{x})\ \vec{n}$
$\rightarrow_{\beta}\ \vec{x}[\vec{n}/\vec{x}]$
$= \{\}^+[\vec{n}/\vec{x}]$

- If $\langle$**for** x_i**in** e **downto** 1 **onlyIf** x_b $\{s\}, \mu\rangle\triangleright$

$$\langle\textbf{if}\ x_b\ \{x_i := \overline{q+1}; \{s\}\ \ \textbf{for}\ x_i\ \textbf{in}\ \overline{q}\ \textbf{downto}\ 1\ \textbf{onlyIf}\ x_b\ \{s\}\}$$

$$\textbf{else}\ \{x_i := \overline{q+1};\}, \mu\rangle$$

where e is either the constant q+1 or a variable x_j and then $n_j = q$+1 then

(**for** x_i **in** e **downto** 1 **onlyIf** x_b $\{s\}$)$^+[\vec{n}/\vec{x}]$

$= (rec(e^+, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.if(x_b^+, let\ \vec{x} = \{s\}^+\ in\ y\ \vec{x},\ \vec{x}))\ \ \vec{x})\ \ [\vec{n}/\vec{x}]$
$= (rec(S^{n+1}, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.if(x_b, let\ \vec{x} = \{s\}^+\ in\ y\ \vec{x},\ \vec{x}))\ \ \vec{n})$
$\rightarrow_{rec}\ (\lambda\vec{x}.if(x_b^+, let\ \vec{x} = \{s\}^+\ in\ y\ \vec{x},\ \vec{x})[rec(S^n, \_)/y, S^{n+1}/x_i])\ \ \vec{n}$
$= (\lambda\vec{x}.if(x_b^+, let\ \vec{x} = \{s\}^+[S^{n+1}/x_i]\ in\ rec(S^n, \_)\ \vec{x},\ \vec{x}[S^{n+1}/x_i])\ \ \vec{n}$

by lemma, we have

$$\begin{array}{rl} = & (\lambda\vec{x}.if(x_b^+, \\ & \quad let\ x_i = S^{n+1}\ in\ let\ \vec{x} = \{s\}^+\ in\ rec(S^n, \_)\ \vec{x}, \\ & \quad let\ x_i = S^{n+1}\ in\ \vec{x}) \\ & \ \vec{n} \end{array}$$

$$\begin{array}{rl} = & (\lambda\vec{x}.if(x_b, \\ & \quad (\{x_i := \overline{q+1}; \{s\};\ \textbf{for}\ x_i\ \textbf{in}\ \overline{q}\ \textbf{downto}\ 1\ \textbf{onlyIf}\ x_b\ \{s\}\})^+, \\ & \quad \{\ x_i := \overline{q+1};\}^+)\ \ \vec{n} \\ \rightarrow_{\beta} & (\textbf{if}\ x_b\ \{x_i := \overline{q+1}; \{s\};\ \textbf{for}\ x_i\ \textbf{in}\ \overline{q}\ \textbf{downto}\ 1\ \textbf{onlyIf}\ x_b\ \{s\}\} \\ & \quad \textbf{else}\ \{\ x_i := \overline{q+1};\})^+[\vec{n}/\vec{x}] \end{array}$$

□

GCD's sample. The *LoopC* program for the *gcd* function is then translated to the following term (`GCDcore'` is the translated of `GCDcore`):

```
let (x,y,r,s) = GCDcore' in
    let (x, y,r,s) =
        rec(N
            ,λ(x,y,r,s). (x,y,r,s),
            ,[y,I]λ(x,y,r,s). if (s
                                  ,let (x, y,r,s) =
                                  GCDcore'
                                  in y (x, y,r,s)
                                  ,(x, y,r,s)))
    in
        let (x, y,r,s) =
        if (!s
            ,rec(N
            ,λ(x,y,r,s). (x,y,r,s),
            ,[y,I]λ(x,y,r,s). if (s
                                  ,let (x, y,r,s) =
            GCDcore'
            in y (x, y,r,s)
            ,(x, y,r,s)))
        ,(x,y,r,s))
    in
        (x,y,r,s)
```

5. Conclusion : from $LoopC^+$ to PRV.

It has been shown in [1], that an extension of *Loop* language with an escape statement is complete relatively to *APRA*, a large class of algorithms (the class of simple arithmetic *abstract state machine* with a primitive recursive run).

By simulation, we show that there exists a total functional language that is complete relatively to *APRA* too. The functional language comes with an evaluation strategy that mixes call-by-value (for β–reduction) and call-by-name (for rec–reduction and if–reduction).

The translation of the **for**–rule leads to the primitive recursive program to level one of system T. But clearly, the term obtained is not as powerful as a term of the full level one of system which may represent the *Ackermann* function (that is not primitive recursive, see [26]). In fact, the term obtained is representable in the *primitive recursion with variable parameters* framework (PRV, see [21]).

Definition. *The set of PRV functions contains the basic functions ($\vec{x}$ stands for $(x_1, \cdots, x_n)$)*

$$\begin{array}{ll} f(x) = x+1 & \textit{(successor function)} \\ f(x) = x-1 & \textit{(predecessor function)} \\ f(\vec{x}) = q & \textit{(constant functions)} \\ f(\vec{x}) = x_i & \textit{(projection functions)} \end{array}$$

and is closed by the schema of composition with $F, G_1, \cdots, G_m$ in PRV

$$f(\vec{x}) = F(G_1(\vec{x}), \cdots, G_m(\vec{x}))$$

and by the primitive recursion with variable parameters schema with g, j and h in PRV

$$\begin{array}{ll} f(0, \vec{x}) & = g(\vec{x}) \\ f(y+1, \vec{x}) & = h(y, f(y, j(\vec{x})), \vec{x}) \end{array}$$

A function in PRV can be view as a recursive program (a system of recursive term equations, see [27]).

The terms of $LoopC^{+}$ are those obtained by the translation $^{+}$ of $LoopC$ programs. One can then define a translation from terms of $LoopC^{+}$ to the PRV term as follows:

Definition.

$$(let\ \vec{x} = v\ in\ u)^* \equiv \left| \begin{array}{ll} f_u(\vec{x}) & = u^* \\ f_v(\vec{x}) & = v^* \\ f(\vec{x}) & = f_u(f_v(\vec{x})) \end{array} \right.$$

$$(\vec{y})^* \equiv \vec{y}$$

$$(if(b,u,v))^* \equiv \left| \begin{array}{ll} f_u(\vec{x}) & = u^* \\ f_v(\vec{x}) & = v^* \\ f(\vec{x}) & = if(b^*, f_u(\vec{x}), f_v(\vec{x})) \end{array} \right.$$

$$\begin{array}{lll} (S(x))^* & \equiv x & + \quad 1 \\ pred(x))^* & \equiv x & - \quad 1 \\ (0)^*, (true)^*, (false)^* & \equiv 0, true, false & \end{array}$$

$$(u\ v)^* \equiv \left| \begin{array}{ll} f_u(\vec{x}) & = u^* \\ f_v(\vec{x}) & = v^* \\ f(\vec{x}) & = f_v(f_u(\vec{x})) \end{array} \right.$$

$$(rec(e, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.if(b, let\ \vec{x} = s\ in\ y\ \vec{x},\ \vec{x}))\ \ \vec{x})^*$$

$$\equiv \left| \begin{array}{ll} f_s(\vec{x}) & = s^* \\ f_b(\vec{x}) & = b^* \\ f(0, \vec{x}) & = \vec{x} \\ f(n+1, \vec{x}) & = if(f_b(\vec{x}), f(n, f_s(\vec{x})), \vec{x}) \end{array} \right.$$

So we rely a subclass of *abstract state machine* to an equational presentation of algorithms by simulation. The two notions of algorithms (the one of Y. Gurevich and the one of Y. Moschovakis) can be considered similar for the small class of primitive recursive algorithms. This is promising for the comparison of a large class of algorithms (essentially arithmetics).

References

[1] Ph. Andary, B. Patrou, and P. Valarcher. "A theorem of representation for primitive recursive algorithms." *Fundamenta Informaticae*, 2009.

[2] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993.

[3] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.

[4] N. Dershowitz and Y. Gurevich. A natural axiomatization of church's thesis. *Bulletin of symbolic logic*, 2008.

[5] P. Doyle S. Dexter and Y. Gurevich. Gurevich abstract state machine and schonhage storage modification machines. *J. Universal Computer Science*, 3(4):279–303, 1997.

[6] S. Grigorieff and P. Valarcher. Local evolving algebras unify all kinds of sequential computation models. Submitted (2009).

[7] Y. N. Moschovakis. On founding the theory of algorithms. In H. G. Dales and G. Oliveri, editors, *Truth in mathematics*, pages pp. 71 – 104. Clarendon Press, Oxford, 1998.

[8] Y. N. Moschovakis. What is an algorithm ? In Springer, editor, *Mathematics unlimited – 2001 and beyond*, pages 919–936. B. Engquist and W. Schmid, 2001.

[10] L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 372, 1989.

[11] L. Colson. A unary representation result for system t. *Annals of Mathematics and Artificial Intelligence*, 16:385-403, 1996.

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

[13] R. David. Decidability results for primitive recursive algorithms. *Theoretical Computer Science*, 300:477–504, 2003.

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

[15] R. David. On the asymptotic behaviour of primitive recursive algorithms (part 2). *Draft paper*, 1997.

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

[17] L. Van Den Dries. Generating the greatest common divisor, and limitations of primitive recursive algorithms. *Foundations of Computational Mathematics*, 3 (2003) 297-3224.

[18] T. Crolard, E. Polonowski, and P. Valarcher. Extending the loop language with higher-order procedural variables. *ACM TOCL*, page 38, 2009.

[19] P. Valarcher. A complete characterization of primitive recursive intensional behaviours. *Theoretical Informatics and Applications*, (42) 2008.

[20] T. Crolard, S. Lacas, and P. Valarcher. On the expressive power of loop language. *Nordic Journal of Computing*, 13:46–57, 2006.

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

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

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

[24] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[25] L. Colson. Représentation intensionnelle d'algorithmes dans les systèmes fonctionnels. *Thèse de doctorat, Université P 7*, 1991.

[26] C. Calude, M. Salomon, and T. Ionel. The first example of a recursive function which is not primitive recursive. *HistoriaMath.*, 6:380–384, 1979.

[27] Y. N. Moschovakis and V. Paschalis. Elementary algorithms and their implementations. In Benedikt Lowe S. B. Cooper and Andrea Sorbi, editors, *New Computational Paradigms*. Springer, 2008.