About implementation of primitive recursive algorithms

Philippe ANDARY, Bruno PATROU and Pierre VALARCHER

LIFAR, University of Rouen
Place Emile Blondel, 76820 Mont Saint-Aignan Cedex
email: `{Philippe.Andary, Bruno.Patrou, Pierre.Valarcher}@univ-rouen.fr`

Abstract. We consider the set of *primitive recursive functions* (PR) and we define a set of *algorithms* computing elements of PR as the set of *abstract state machines* (ASMs) which have their running lengths computable by PR functions. Then we show that there exists a programming language (only computing PR functions) which simulates these ASMs preserving their complexity.

Keywords: Primitive recursive functions, algorithms, abstract state machines, programming.

1 Introduction

Related works and motivations. From now more than a decade, there is a growing interest in structural complexity of programming languages. First work was done by L. Colson (see [3]) using denotational semantics (the used domain is the lazy natural integers [11]) to prove that, although the function *min*, which computes the minimum of two integers, is obviously a primitive recursive function (see [16] for a formal definition), there is no way to represent, in the model of primitive recursive programs (which are called PR-combinators), the *good algorithm*, *i.e.* the one which decreases alternatively both arguments. He proved an *ultimate obstination theorem* and showed that every PR-combinator must *choose* one (and only one) of its arguments and thus the alternation between arguments is impossible. A constructive proof of this property can be found in [5].

This work has been followed by R. David (see [8]) who developped a new semantics (the *trace* of computation) allowing him to prove a new property (the *backtracking property*) for any primitive recursive program using any kind of data types.

The third author has shown some new results on intensional behaviour (now called structural complexity) of other primitive recursive schemes (in [18]).

In the same framework, L. Colson and D. Fredholm (see [12] and [4]) show that call-by-value strategy (with primitive recursion over lists of integers and with primitive recursion in higher types, called system T of Gödel) does not allow to compute the *good algorithm* of the *min* function.

Similar questions have been studied by S. Brookes and D. Dancanet in [2] and [7] with non-determinism and CDS languages.

Recently, in [15], Y. Moschovakis has established a linear lower bound for the complexity of non-trivial primitive recursive program from *piecewise linear* given functions. His main result is that logtime programs for the greatest common divisor from such givens (such as *Stein*'s) cannot matched in efficiency by primitive recursive programs from the same given functions. He ended by an open problem relative to the classical *Euclidean algorithm* (L. Van Den Dries gives a partial answer in [10]).

In [6], the *min* problem is studied in an imperative framework: LOOP language which computes only primitive recursive functions ([14]). But there is no good program for the *min* function neither.

From now on, most results are negative answers to particular problems: the *min* problem, the *gcd* problem, ...

On the other part, Y. Gurevich has developped a framework, called *abstract state machine*, that captures the notion of sequential algorithm. This formal definition of algorithms allows us to ask a more general question:

> *let Alg be a set of algorithms, is there a programming language which implements the elements in this set (and no more)?*

In our purpose, we restrict the set *Alg* to primitive recursive algorithms (abstract state machine with primitive recursive run lengths).

As LOOP is not enough powerful we extend this language with an `exit`; command allowing to halt a computation anywhere in a program (see [17] for pedagogical considerations).

For the sake of simplicity, we constrain the vocabulary of ASMs to match with our programming language.

Organization of the paper. Section 2 is devoted to the LOOP_{exit} programming language. Sections 3 and 4 describe the notion of primitive recursive algorithm from the ASM framework. The two next sections are related to the main theorem, proving that LOOP_{exit} programming language implements primitive recursive algorithms.

2 The imperative LOOP language

2.1 Syntax

Let us define the LOOP language, as it is first described in [14], but with a more recent syntax. We add boolean expressions, conditional statements, the predecessor function, and a special expression which denotes undefined value:

$$
\begin{array}{lll}
expr_{int} & ::= & var \mid \texttt{pred}(expr_{int}) \mid \texttt{succ}(expr_{int}) \mid natural \mid undef \\
expr_{bool} & ::= & var = expr_{int} \mid \neg expr_{bool} \mid expr_{bool} \wedge expr_{bool} \\
 & \mid & expr_{bool} \vee expr_{bool} \mid true \mid false \\
com & ::= & var := expr_{int}; \\
 & \mid & \texttt{if}\ expr_{bool}\ \texttt{then}\ com_{list}\ \texttt{else}\ com_{list}\ \texttt{endif}; \\
 & \mid & \texttt{loop}\ expr_{int}\ \texttt{do}\ com_{list}\ \texttt{endloop}; \\
com_{list} & ::= & com\ com_{list} \\
 & \mid & \epsilon \\
prog & ::= & com_{list}
\end{array}
$$

All variables are denoted by strings. We denote $Var(P)$ the set of variables of a program P and $Var_{in}(P)$ (resp. $Var_{out}(P)$) the subset of variables which contain the inputs (resp. outputs). Their type is natural integer and those in $Var_{in}(P)$ must be explicitly initialized before the execution of P. For the sake of simplicity, we will write "`if` e `then` c `endif`;" when the `else` part of the `if` command is an empty list of commands. Furthermore, we will allow the use of `elsif` keyword when conditional commands are nested, as in:

`if` e `then`
  com
`elsif` e `then`
  com
`endif`;

The semantics is the usual one, we just want to point out the fact that the integer expression in the loop command is evaluated first and only once.

In the sequel we will extend the LOOP language in such a way that we can implement new algorithms (still computing only primitive recursive functions).

2.2 Conservative extension : LOOP$_{exit}$

We extend the language with the `exit` command which allows us to stop the program execution:

$$com\ ::=\ \texttt{exit};$$

2.3 Operational semantics

The operational semantics is given by a simple abstract machine which is described by a set of rewriting rules. A rule has the form $((\alpha, env), com) \Rightarrow ((\alpha', env'), com')$, where α and α' lie in $\{0,1\}$, env and env' denote environments, and com and com' are programs (a rule is called a *one step reduction* or *one step rewriting*). Let com_1, com_2 and com_s be commands lists:

$$((0, env), x := e;\ com_s\ ) \Rightarrow ((0, env \oplus \{(x, valueOf(e))\}), com_s\ )$$

$$((0, env), \texttt{if}\ e\ \texttt{then}\ com_1\ \texttt{else}\ com_2\ \texttt{endif};\ com_s\ )$$

$$\Rightarrow ((0, env), com_1 \; com_s \;) \text{ if } valueOf(e) = true$$

$$((0, env), \texttt{if } e \texttt{ then } com_1 \texttt{ else } com_2 \texttt{ endif; } com_s \;)$$
$$\Rightarrow ((0, env), com_2 \; com_s \;) \text{ if } valueOf(e) = false$$

$$((0, env), \texttt{loop } expr_{int} \texttt{ do } com \texttt{ endloop; } com_s \;)$$
$$\Rightarrow ((0, env), com_s \;) \text{ if } valueOf(expr_{int}) = 0$$

$$((0, env), \texttt{loop } expr_{int} \texttt{ do } com \texttt{ endloop; } com_s \;)$$
$$\Rightarrow ((0, env), com \texttt{ loop } n \texttt{ do } com \texttt{ endloop; } com_s \;)$$
$$\text{if } valueOf(expr_{int}) = n + 1$$

$$((0, env), \texttt{exit; } com_s \;) \Rightarrow ((1, env), com_s \;)$$

where, as usual, an environment is a mapping of variables onto values (natural numbers) which can be seen as a set of couples. We say that an environment env is for a program P if all variables of P appear in env. Let y be a variable and env be an environment, if $env = env' \oplus \{(x, v)\}$ for some variable x and value v then $env(y) = v$ or $env(y) = env'(y)$ according to the fact that $y = x$ or not.

If e is an expression (boolean or natural), then $valueOf(e)$ is defined as usual. We just want to lay stress on the fact that $valueOf(e) = 0$ whenever $e = \texttt{pred}(e')$ with $valueOf(e') = 0$.

Let us denote $((\alpha, env), P)$ a *state*. Then a state is terminal if no (reduction) rule is applicable (there is no rule from $((1, env), P)$ neither from the empty sequence). A *run* of a program P is a sequence of one step reductions which leads from a state $((\alpha, env), P)$ to a terminal state $((\alpha', env'), P')$. We denote by $\Rightarrow^n$ the nth iteration of $\Rightarrow$ and $\Rightarrow^*$ its reflexive and transitive closure.

Remark 1. Those reduction rules are deterministic.

2.4 Some theoritical results

Theorem 1. *All primitive recursive functions are computable by a* LOOP *program.*

Proof. See [9] for a proof. ■

Theorem 2. *No* LOOP *program computes the* min *function in time* $\mathcal{O}(\min)$.

Proof. See [6] for a proof. ■

Lemma 1. *If P_e is a LOOP_{exit} program and $((0, env_1), P_e) \Rightarrow^* ((\alpha, env_2), P'_e)$ ($\alpha \in \{0, 1\}$) then there exists a* LOOP *program P such that $((0, env_3), P) \Rightarrow^* ((0, env_4), P')$ with $env_1 \subset env_3$ and $env_2 \subset env_4$.*

Proof. Use a new variable v_{exit} initialized with 0. Then substitute $v_{exit} := 1$; for all `exit` commands in P_e. Finally, replace each command com in P_e with this one: `if` $v_{exit} = 0$ `then` com `endif`; to get P ■

Corollary 1. *The* $\textsc{Loop}_{exit}$ *language computes exactly all primitive recursive functions.*

Definition 1. *Let P be a* $\textsc{Loop}_{exit}$ *program. We denote by* $T_P : \mathbb{N}^{|Var_{in}(P)|} \mapsto \mathbb{N}$ *the mapping from P inputs to the number of one step rewriting rules executed during a computation of P.*

Proposition 1. *There exists a* $\textsc{Loop}_{exit}$ *program which computes the* min *function in time* $\mathcal{O}(\min)$.

Lemma 2. *Let f be a primitive recursive function from $\mathbb{N}^k$ into $\mathbb{N}$. Then, there exists a* $\textsc{Loop}$ *program Q such that $T_Q \geq f$.*

Proof. Let P be a $\textsc{Loop}$ program computing f and let us call r the variable whose value is $f(v_1, \ldots, v_k)$ ($(v_1, \ldots, v_k)$ are the input values for P) after the execution of P. Just add the loop

```
loop r do  endloop;
```

to the end of P to get Q. ■

3 Y. Gurevich algorithms definition

We first recall the main concepts defined in [13], then we add some new definitions for later use.

A **vocabulary** is a finite collection of function names, each with a fixed arity. **Terms** over a given vocabulary Υ (or Υ-terms) are defined by induction: a nullary function name is a term; if $f \in \Upsilon$ has arity k and $t_1, ..., t_k$ are terms then $f(t_1, ..., t_k)$ is a term. A **Υ-structure** X is a nonempty set S (the **base set** of X) together with an interpretation of the function names over S. Elements of S are also called elements of X. The **value** $val(t, X)$ of a term t in a structure X is also defined by induction: if t is a nullary function name $val(t, X)$ is the interpretation of t over the base set of X else $t = f(t_1, ..., t_n)$ (also denoted $f(\overline{t})$ when the arity does not matter) and $val(t, X)$ is the interpretation of f for the parameters given by $val(t_1, X), ..., val(t_n, X)$. Some function names can be marked as **relational**, their values always belonging to the boolean universe $\{true, false\}$. A term is boolean if the outermost function name is relational.

Let X be a Υ-structure. If f is a j-ary function name over Υ and $\overline{a}$ is a j-tuple of elements of X, then the pair $(f, \overline{a})$ is a **location** of X and we denote by $content_X(f, \overline{a})$ the element $f(\overline{a})$ of X. If $(f, \overline{a})$ is a location of X and b is an element of X, then $(f, \overline{a}, b)$ is an **update** of X, which is trivial if $b = content_X(f, \overline{a})$. To **execute** an update $(f, \overline{a}, b)$ replace the current content of location $(f, \overline{a})$ with b. Two updates **clash** if they refer to the same location but are distinct. A set of updates is **consistent** if it has no clashing updates. To execute a consistent set of updates, execute simultaneously all updates in the set. To execute an inconsistent set of updates do nothing. The result of executing an update set Δ over X will be denoted by $X + \Delta$ and it is proved in [13] that for any couple (X, Y) of Υ-structures with the same base set there is a unique consistent set Δ of nontrivial updates of X such that $Y = X + \Delta$.

To link up structures one another we use three fundamental rules:

Definition 2. *A **rule** over a vocabulary Υ has one of the three following forms:*

1. $f(t_1, ..., t_n) := t_0$
2. ***par***
   R_1
   $\vdots$
   R_k
   endpar
3. ***if*** g ***then***
   R'
   endif

*where $f \in \Upsilon$ has arity n, $t_0, t_1, ..., t_n$ are Υ-terms, $R', R_1, ..., R_k$ are rules and g (the **guard** of the third rule) is a boolean Υ-term.*

To **fire** a rule R over a structure X is equivalent to execute the set of updates $\Delta(R, X)$ where:

1. $\Delta(R, X) = \{(f, (a_1, ..., a_n), a_0)\}$ when R corresponds to the first form rule, also called an **update rule** ;
2. $\Delta(R, X) = \Delta(R_1, X) \cup ... \cup \Delta(R_k, X)$ when R corresponds to the second form rule, called a **par rule** or a **skip rule** if $k = 0$.
3. when R corresponds to the third form rule, called **conditionnal rule**, $\Delta(R, X) = \Delta(R', X)$ if g evaluates to *true* and $\Delta(R, X) = \emptyset$ else.

An (ASM) **program** Π is a rule and the mapping τ_Π is defined for any Υ-structure X by: $\tau_\Pi(X) = X + \Delta(\Pi, X)$.

Definition 3. *An **Abstract State Machine** A (say ASM for short) of vocabulary Υ_A is given by:*

- *a program Π_A of vocabulary Υ_A*
- *a set $\mathcal{S}(A)$ of Υ_A-structures (also called **states**) closed under isomorphisms and under the map τ_{Π_A}*
- *a subset $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ closed under isomorphisms.*

The restriction of τ_{Π_A} to $\mathcal{S}(A)$ is usually denoted by τ_A. Let $X_0 \in \mathcal{I}(A)$ the sequence $X_0, \tau_A(X_0), \tau_A^2(X_0), ...$ is called a **run** of A. Two ASMs A and A' are **equivalent** if $\mathcal{S}(A) = \mathcal{S}(A')$, $\mathcal{I}(A) = \mathcal{I}(A')$, and if they have the same runs.

It is convenient to specify the functions of a vocabulary as **dynamic** or **static**, depending wether they can be modified by the ASM or not. Another distinction is made over dynamic functions: some are called **in** functions or **out** functions to precise where to find the input data and the output results in the ASM. In the sequel we will essentially consider ASMs relatively to the function they compute. In such cases it is essential to specify the sequences of in and out functions used by an ASM and which will be denoted by $in(A)$ and $out(A)$. Y. Gurevich says also that the vocabulary of an ASM always contains the nullary function names $true$, $false$, $undef$, the names of usual boolean operations, the unary function name *boole*, and the binary function name denoted by the equal sign. This set of function names is called the **logic part** of the vocabulary.

We extend in a natural way the *val* notation to sequences of terms

$$val(T, X) = (val(t_1, X), ..., val(t_n, X)) \text{ if } T = (t_1, ..., t_n)$$

and sets of states

$$val(T, \{X_1, ..., X_k\}) = \{val(T, X_1), \ldots, val(T, X_k)\}.$$

Considering a state X of an ASM A, any other state of A is reachable or not from X by τ_A. We measure this reachability with the following definition:

Definition 4. *Let A be an ASM and X be a state in $\mathcal{S}(A)$. We define the partial map $\delta_X : \mathcal{S}(A) \rightarrow \mathbb{N}$ by*

$$\begin{aligned} \delta_X(Y) &= 0 \quad \textit{if } Y = X \\ &= k \quad \textit{if } \exists Y' \in \mathcal{S}(A) \textit{ such that } Y = \tau_A(Y') \textit{ and } \delta_X(Y') = k - 1 \end{aligned}$$

Notice that δ_X is undefined on the set of states unreachable from X, and is well-defined elsewhere since τ_A is a map.

Y. Gurevich stipulates that an ASM A stops when a state X verifies $\tau_A(X) = X$. We will explicitly qualify such a state as **terminal**.

Since τ_A is a map, any state of an ASM A can be associated to at most one terminal state (depending if the runs it belongs to are finite or not).

We observe, from [13], that there exists for any ASM A an equivalence relation with finite index over $\mathcal{S}(A)$ and, if we denote by $C_1, \ldots, C_n$ the classes of this relation, there exists boolean expressions g_i characterizing each C_i. That is:

$$\forall X \in \mathcal{S}(A), \exists i \in \{1, n\} \text{ such that } (val(g_j, X) \text{ holds } \Rightarrow j = i)$$

Y. Gurevich has proved that there always exists an equivalent ASM A' whose program $\Pi_{A'}$ is:

par
  if g_1 **then**
    R_1
  endif
  $\vdots$
  if g_n **then**
    R_n
  endif
endpar

where each R_i is a par rule containing only update rules, and such that for any $X \in C_i$, $\Delta(R_i, X)$ is the unique consistent set of non-trivial updates verifying $X + \Delta(R_i, X) = \tau_A(X)$.

The ASM A' will be called a **partitionned ASM** since $\mathcal{S}(A')$ is partitionned by the guards of $\Pi_{A'}$. From these observations we can state, for any partitionned ASM A, that:

- for all $X \in \mathcal{S}(A)$, $\Delta(\Pi_A, X)$ contains no trivial update,
- for any state $X \in \mathcal{S}(A)$ the rule R_i, executed when Π_A is fired on X, is a skip rule iff X is a terminal state.

Definition 5. *Let A be an ASM and T be a finite sequence of terms defined over Υ_A. We say that T is* ***characteristic*** *for $\mathcal{I}(A)$ if:*

$$\forall X_0, X_0' \in \mathcal{I}(A), val(T, X_0) = val(T, X_0') \Rightarrow X_0 = X_0'$$

We constrain the sequence $in(A)$ of an ASM A to only contains nullary function names and to be characteristic for $\mathcal{I}(A)$. Indeed, if $in(A)$ is not characteristic the ASM is non-deterministic and we do not want to deal with such algorithms which will be impossible to implement with LOOP language. An ASM whose definition is accompanied by the specification of a sequence of nullary function names $in(A)$ characteristic for $\mathcal{I}(A)$ and of a sequence $\mathrm{out}(A)$ (also restricted to nullary function names) is called an **$\mathcal{F}$-ASM**.

4 Primitive recursive ASMs

The aim of this section is to give a useful notion for primitive recursive algorithms. According to Y. Gurevich's thesis it is sufficient to define the concept of primitive recursive (or PR) ASM. First we define a complexity notion for $\mathcal{F}$-ASMs:

Definition 6. *Let A be an $\mathcal{F}$-ASM. We call* ***complexity*** *of A and we denote by c_A, the function defined over the set*

$$\{val(in(A), X_0) \mid X_0 \in \mathcal{I}(A) \textit{ and its associated run is finite}\}$$

by:

$$c_A(val(in(A), X_0)) = \delta(X_0, Z_0)$$

where Z_0 is the terminal state of the finite run of A initiated with X_0.

Of course PR ASMs are related to PR functions. Such functions only take their values over $\mathbb{N}$, so we will restrict us to ASMs working over non-negative integers. In this sense we define:

Definition 7. *$\mathbb{N}$-typed ASMs are ASMs whose non-logic part of the vocabulary only contains functions taking their values on $\mathbb{N}$.*

PR ASMs definition follows:

Definition 8. *An $\mathbb{N}$-typed $\mathcal{F}$-ASM A is PR if c_A is PR.*

A PR function is necessarily defined from $\mathbb{N}^n$ to $\mathbb{N}$. So, to be PR, c_A must be defined over $\mathbb{N}^n$ and, imposing A to be $\mathbb{N}$-typed ensures $val(in(A), \mathcal{I}(A))$ completely defined over $\mathbb{N}$.

To compare algorithms from a complexity point of view we need to focuse on a certain level of abstraction. This can be made, for example, by minimizing the vocabulary of our ASMs as for Böhm-Jacopini ASMs (see [1]):

Definition 9. *An $\mathbb{N}$-typed ASM A is called* ***basic*** *if the non-logic part of its vocabulary is reduced to:*

$$\{n_1, \ldots, n_j, pred, succ, x_1, \ldots, x_k\}$$

where:

- $n_1, \ldots, n_j$, *pred and succ are static and represent respectively natural constant operations which belong to $\mathbb{N}$, the unary predecessor and successor functions on $\mathbb{N}$,*
- $x_1, \ldots, x_k$ *are nullary functions defined over $\mathbb{N}$ and the only dynamic functions of the ASM; as usual we can identify them with variables, further denoted by $\mathcal{V}ar(A)$.*

The last two sections refer to basic ASMs. This is justified by the fact that, as Böhm-Jacopini ASMs, basic ASMs allow us to compute any recursive function. Furthermore, we can notice that PR ASMs computations are not restricted to PR functions computations (for example, if the Ackermann function is in the vocabulary of a PR ASM, you can compute this function with this PR ASM). So when deciding wether an ASM is PR or not, we focuse on its behaviour, not on the function it is computing.

5 From PR ASMs to LOOP_{exit} programs

The object of this section is to build from a basic PR ASM A computing a function f, a LOOP_{exit} program P for f which simulates A in some sense. In particular, P has to preserve the complexity of A.

Partitionned ASMs allow us to construct LOOP_{exit} programs (without loop) which will be called **heart**-programs. A heart-program is a piece of code that will be executed as many times as necessary later in the whole program.

Definition 10. *For any basic partitionned $\mathcal{F}$–ASM A (with Π_A as described in section 3) we define a LOOP_{exit} program P_A, with $Var_{in}(P_A) = in(A)$ and $Var_{out}(P_A) = out(A)$ by:*

`if` g_1 `then`
    P_{R_1}
`elsif` g_2 `then`

$\vdots$

```
else
    P_{R_n}
endif;
```

where,

if R_i *is:*	*then* P_{R_i} *is:*
par $x_1 := e_1$ $\vdots$ $x_k := e_k$ **endpar**	$x'_1 := x_1;$ $x_1 := e_1;$ $x'_2 := x_2[x'_1/x_1];$ $x_2 := e_2[x'_1/x_1];$ $\vdots$ $x'_k := x_k[x'_1/x_1][x'_2/x_2]\dots[x'_{k-1}/x_{k-1}];$ $x_k := e_k[x'_1/x_1][x'_2/x_2]\dots[x'_{k-1}/x_{k-1}];$

and if R_i *is a skip rule then* P_{R_i} *is the* `exit` *command.*

The auxiliary variables are needed to simulate simultaneous execution of updates and $x_k := b_k[x'_1/x_1][x'_2/x_2]\dots[x'_{k-1}/x_{k-1}]$; denotes sequential substitutions.

Remark 2. The conditionnal statements must be imbricated since we come from an environment with parallel executed updates to a program with sequential instructions. This approach is acceptable because the guards $g_1, ..., g_n$ are mutually exclusive in a partitionned ASM.

For any state $X \in \mathcal{S}(()A)$ we denote by $env(X)$ the set $\{(v, val(v, X)) \mid v \in \mathcal{V}ar(A)\}$. This allows us to associate environments of P_A from a state.

Lemma 3. *Consider a basic partionned ASM A and two distinct states $X, Y \in \mathcal{S}(()A)$ with $\tau_A(X) = Y$. Let R be the par rule (with size $i > 0$) fired by A on X and P_R be the sequence of assignments (it has size $2 \times i$) constructed from R by the previously described transformation. If env is an environment for P_R containing $env(X)$ then*

$$((0, env), P_R) \Rightarrow^{2\times i} ((0, env'), \epsilon) \text{ with } env(Y) \subseteq env'$$

Proof. Direct from the definitions of operational semantics for ASM programs and LOOP$_{exit}$ programs. ■

Lemma 4. *Let A be a basic n-partitionned ASM and $X, Y \in \mathcal{S}(A)$ with $\tau_A(X) = Y$. If env is an environment for P_A containing $env(X)$ then there exists an integer $l \leq 2 \times j + n$ (where j is the size of the biggest par rule in A) such that*

$$((0, env), P_A) \Rightarrow^{l} ((\alpha, env'), \epsilon) \text{ with } env(Y) \subseteq env' \quad (\alpha \in \{0, 1\})$$

Proof. If the rule R_i, fired in X by A, is not a skip rule then lemma 3 is applied to obtain the desired result with $\alpha = 0$ and $l = i + 2 \times |R_i|$ (i guards to evaluate). Else $Y = X$, the command executed is an `exit` command, and the result is obtained with $\alpha = 1$ and $l = i + 1$ ($1 \leq i < n$). ∎

Note that n and j are constants only depending on the ASM.

Definition 11. *Let P be a* Loop *program and Q be a* Loop_{exit} *program. We define the* **insertion** *of Q in P (denoted by $P[Q]$) by induction on the length of P:*

- *if P is $x := e;\ com_s$ then*
  $P[Q]$ *is* $Q\ x := e;\ com_s[Q]$
- *if P is* `if` e `then` com_1 `else` com_2 `endif`; com_s *then*
  $P[Q]$ *is* Q `if` e `then` $com_1[Q]$ `else` $com_2[Q]$ `endif`; $com_s[Q]$
- *if P is* `loop` $expr_{int}$ `do` com `endloop`; com_s *then*
  $P[Q]$ *is* Q `loop` $expr_{int}$ `do` $com[Q]$ Q `endloop`; $com_s[Q]$

Lemma 5. *Let A be a basic n-partitionned ASM and X, Y be two distincts states of $S(A)$ with $\tau_A(X) = Y$. Let env_{P_A} be an environment for P_A containing $env(X)$. Let P be a* Loop *program and env_P be an environment for P disjoint from env_{P_A}. If $((0, env_P), P) \Rightarrow ((0, env'_P), P')$ then $((0, env_P \cup env_{P_A}), P[P_A]) \Rightarrow^{l+1} ((0, env'_P \cup env'_{P_A}), P'[P_A])$ with $env(Y) \subseteq env'_{P_A}$ and $l \leq 2 \times j + n$ (where j is the size of the biggest par rule in A)*

Proof. By case on P and with the use of lemma 4:

- if $((0, env_P), v := e;\ com_s)$
  $\Rightarrow ((0, env_P \oplus \{(v, valueOf(e))\}), com_s)$
  then $((0, env_P \cup env_{P_A}), P_A\ v := e;\ com_s[P_A])$
  $\Rightarrow^{l+1} ((0, env_P \oplus \{(v, valueOf(e))\} \cup env'_{P_A}), com_s[P_A])$
- with $valueOf(b) = true$,
  if $((0, env_P),$ `if` b `then` com_1 `else` com_2 `endif`; $com_s)$
  $\Rightarrow ((0, env_P), com_1\ com_s)$
  then $((0, env_P \cup env_{P_A}), P_A$ `if` b `then` $com_1[P_A]$ `else` $com_2[P_A]$ `endif`; $com_s[P_A])$
  $\Rightarrow^{l+1} ((0, env_P \cup env'_{P_A}), com_1[P_A]\ com_s[P_A])$
- with $valueOf(b) = false$,
  if $((0, env_P),$ `if` b `then` com_1 `else` com_2 `endif`; $com_s)$
  $\Rightarrow ((0, env_P), com_2\ com_s)$
  then $((0, env_P \cup env_{P_A}), P_A$ `if` b `then` $com_1[P_A]$ `else` $com_2[P_A]$ `endif`; $com_s[P_A])$
  $\Rightarrow^{l+1} ((0, env_P \cup env'_{P_A}), com_2[P_A]\ com_s[P_A])$
- with $valueOf(e) = 0$,
  if $((0, env_P),$ `loop` e `do` com `endloop`; $com_s)$
  $\Rightarrow ((0, env_P), com_s)$
  then $((0, env_P \cup env_{P_A}), P_A$ `loop` e `do` $com[P_A]$ P_A `endloop`; $com_s[P_A])$
  $\Rightarrow^{l+1} ((0, env_P \cup env'_{P_A}), com_s[P_A])$

– with $valueOf(e) = n + 1$,
if $((0, env_P), \texttt{loop}\ e\ \texttt{do}\ com\ \texttt{endloop};\ com_s)$
$\Rightarrow ((0, env_P), com\ \texttt{loop}\ n\ \texttt{do}\ com\ \texttt{endloop};\ com_s)$
then $((0, env_P \cup env_{P_A}), P_A\ \texttt{loop}\ e\ \texttt{do}\ com[P_A]\ P_A\ \texttt{endloop};\ com_s[P_A]) \Rightarrow^{l+1}$
$((0, env_P \cup env'_{P_A}), com[P_A]\ P_A\ \texttt{loop}\ n\ \texttt{do}\ com[P_A]\ P_A\ \texttt{endloop};\ com_s[P_A])$

■

This lemma points out the fact that, when inserting P_A in P, each rewriting rule of P is preceded by an execution of P_A.

5.1 Main result

Theorem 3. *Let f be a function from $\mathbb{N}^k$ to $\mathbb{N}$ and A be a basic n–partitionned PR $\mathcal{F}$-ASM which computes f. Then there exists a* LOOP$_{exit}$ *program computing f whose complexity belongs to $\mathcal{O}(c_A)$.*

Proof. From lemma 4, each execution of P_A simulates (in at most $2 \times j + n$ steps) one step in a run of A. From lemma 5, we know that inclusion of P_A in a LOOP program P induces an execution of P_A before each rewriting rule of P. So, we just have to find a LOOP program with reductions of size greater than the length of A runs, that is $T_P \geq c_A$. The existence of such a program is ensured by lemma 2 since, A being a PR ASM, c_A is a PR function. Now it is easy to conclude that the desired LOOP$_{exit}$ program $P_{A,f}$ is:
$Var_{in}(P) \longleftarrow Var_{in}(P_A)$
$P[P_A]$
with $Var_{in}(P_{A,f}) = Var_{in}(P_A)$ and $Var_{out}(P_{A,f}) = Var_{out}(P_A)$.
Clearly $T_{P_{A,f}} \leq k + (2 \times j + n + 1) \times c_A \in O(c_A)$ since the `exit` command stops the program as soon as the run is simulated. ■

6 From LOOP$_{exit}$ to PR ASM

We can prove, with the help of a classical (but long to describe) construction, the reverse of our main result theorem:

Theorem 4. *If f is a primitive recursive function computed by a* LOOP$_{exit}$ *program P then we can construct an $\mathcal{F}$-ASM A_P computing f such that:*

1. *A_P is a basic PR ASM*
2. *$c_{A_P} \in \mathcal{O}(T_P)$*

7 Conclusion

We have shown that if an ASM A (with restricted vocabulary) has its run lengths computable by a primitive recursive function, we can construct a LOOP$_{exit}$ program P that belongs to the same set of complexity than A.

This suggests that LOOP_{exit} is a good programming language to implement primitive recursive algorithms.

This work was done in an imperative framework, but in conjonction with [6], we can find a total functional language which matches in efficiency the LOOP_{exit} programming language.

References

1. Börger, Egon and Stärk, Robert. *Abstract State Machines.* Springer-Verlag (2003).
2. Brookes, Stephen and Dancanet, Denis. Sequential Algorithms, Deterministic Parallelism, and Intensional Expressiveness. POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1995) 13–24.
3. Colson, Loïc. About primitive recursive algorithms. Theoretical Computer Science **83** (1991) 57–69.
4. Colson, Loïc and Fredholm, Daniel. System T, call-by-value and the minimum problem. Theoretical Computer Science **206** (1998) 301–315.
5. Coquand, Thierry. Une preuve directe du Théorème d'ultime obstination. Comptes rendus de l'Académie des sciences **314**(série I) (1992).
6. Crolard, Tristan; Lacas, Samuel and Valarcher, Pierre. On the expressive power of the For-language (to submit).
7. Dancanet, Denis and Brookes, Stephen. Programming language expressiveness and circuit complexity. *In: International Conference on the Mathematical Foundations of Programming Semantics* (1996).
8. David, René. On the asymptotic behaviour of primitive recursive algorithms. Theoretical Computer Science **266**(1-2) (2001) 159–193.
9. Davis, Martin D.; Sigal, Ron and Weyuker, Elaine J. *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science.* Academic Press (2nd edition) (1994).
10. van den Dries, Lou. Generating the greatest common divisor, and limitations of primitive recursive algorithms. (to appear in Foundations of Computational Mathematics).
11. Escardo, Martin H. On Lazy Natural Numbers with Applications to Computability Theory and Functional Programming. SIGACT News **24**(1) (1993) 60–67.
12. Fredholm, Daniel. Computing Minimum with Primitive Recursion over Lists. Theoretical Computer Science **163**(1-2) (1996) 269–276.
13. Gurevich, Yuri. Sequential Abstract State Machines capture sequential algorithms. ACM Trans. Computational Logic **1**(1) (2000) 77–111.
14. Meyer, Albert R. and Ritchie, Dennis M. The complexity of loop programs. Proceedings of the ACM 22nd National Conference, ACM Press (1967) 465–469.
15. Moschovakis, Yannis N. On primitive recursive algorithms and the greatest common divisor function. Theoretical Computer Science **301**(1-3) (2003) 1–30.
16. Peter, Roza. *Recursive Functions.* Academic Press (1968).
17. Roberts, Eric S. Loop exits and structured programming: reopening the debate. SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education (1995) 268–272.
18. Valarcher, Pierre. Intensionality vs Extensionality and Primitive Recursion. Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security, LNCS **1179** (1996) 142–151.