Fundamenta Informaticae XX (2010) 1–18
IOS Press

A representation theorem for primitive recursive algorithms

Philippe Andary, Bruno Patrou
University of Rouen, Computer Scicence Dept.
LITIS, Technopole Madrillet
Philippe.Andary, Bruno.Patrou@univ-rouen.fr

Pierre Valarcher
Université Paris Est Créteil, LACL
IUT Fontainebleau/Senart
pierre.valarcher@u-pec.fr

Abstract. We formalize the algorithms computing primitive recursive (PR) functions as the abstract state machines (ASMs) whose running length is computable by a PR function. Then we show that there exists a programming language (implementing only PR functions) by which it is possible to implement any one of the previously defined algorithms for the PR functions in such a way that their complexity is preserved.

Keywords: primitive recursion, algorithms, abstract state machines, structural complexity.

1. Introduction

In [3, 18], Colson and Moschovakis have obtained interesting results on the limitation of the algorithmic expressivity of a large class of primitive recursive languages. They have shown that even the imperative or functional programming languages do not allow to implement some very simple algorithms. These results have motivated our attempt to construct a more expressive language and prove that it captures a very large class of algorithms that specify primitive recursive functions (shortly: PR functions).

Let F be a set of computable functions and A_F a set of algorithms for F. A programming language L_F is called *A_F–complete* if it permits to implement every algorithm $A \in A_F$ by a program $P \in L_F$ in such a way that the number T_P of commands executed by P and the number $c(A)$ of updates executed by A during their respective computations is related by $T_P = O(c(A))$.

Our aim is to find such an A_F–complete programming language L_F for the set F of the PR functions and the *arithmetical abstract state machines* (AASMs) taken as the set A_F of algorithms for F. (cf. [16, 12]). Moreover, the complexity $c(A)$ of every algorithm $A \in A_F$ (the number of updates executed by A during the computation) is required to be computable by a PR function.

The construction of the required programming language L_F that we are going to propose is based on a simple LOOP language whose variables and expressions have either integer or boolean type, and the statements are reduced to assignments and conditional or bounded loops. This language is known not to be expressive enough (cf. [7]). In order to enhance its expressive power, we add a simple *escape* statement (`exit`[1]). The resulting language LOOP$_{exit}$ is the required one. We are ready to state this as the main result of this paper.

Theorem 1.1. (Representation)
Let f be a primitive recursive function and A_f be an algorithm for f such that $c(A_f)$ is a primitive recursive function. Then one can construct a program P in LOOP$_{exit}$ such that $T_P = O(c(A_f))$.

Historical context One of the first relevant works has been done by L. Colson [3]. Using the denotational semantics (in which the used domain is that of the lazy natural integers [14]), he has proved that, though the function min which computes the minimum of two integers is obviously a primitive recursive function (see [20] 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 for min, i.e. the one that decreases alternatively both arguments. The main reason is an ultimate obstinacy theorem showing that every PR-combinator must choose one (and only one) of its arguments and thus the alternation between the two arguments is impossible. A constructive proof of this property can be found in [5].

This work has been continued by R. David (see [10]) who developed a new semantics (the trace of computation) allowing him to prove a new property (the so called backtracking property) for any primitive recursive program using any kind of data types.

The third author of this paper has shown some new results on intensional behavior (now called structural complexity) of some other primitive recursive schemes (in [22]).

In the same framework, L. Colson and D. Fredholm (see [15] and [4]) have shown that even the call-by-value strategy (with primitive recursion over lists of integers and with primitive recursion in higher types, called the system T of Goedel) does not allow to implement the good algorithm for the min function.

Similar questions have been studied by S. Brookes and D. Dancanet, in [2] and [8], with the non-determinism and the CDS languages.

Recently, in [18], Y. Moschovakis has established a linear lower bound for the complexity of a non-trivial primitive recursive program from given piecewise linear functions. His main result is that the logtime programs for the greatest common divisor from such given functions (such as Stein's) cannot be matched in efficiency by the 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 [13]).

In [7], the min problem has been studied in an imperative framework of a LOOP language which computes only primitive recursive functions ([17]). Even there no good program for the min function can be produced. This result has been extended in the framework of a language with higher-order procedural variables ([6]).

Organization of the paper. Section 2 presents the LOOP$_{exit}$ programming language. Sections 3 and 4 describe the notion of primitive recursive algorithm within the ASM framework. The two remaining sections are devoted to the proof of the main theorem.

[1]The *escape* command is very simple; it stops the computation (like in C). It can be replaced by a more sophisticated behavior, as exception or escape from the named block, for instance.

2. Imperative LOOP languages

2.1. Syntax

Let us start with the version of LOOP language that has first been described in [17], and slightly modify it by a more recent syntax. This modification is done by adjoining to it the boolean expressions, conditional statements, the predecessor function, and a special expression for undefined values:

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

Remark 2.1. We include in this LOOP language also all the natural numbers though our language allows to get all of them from 0.

All variables are of the natural integer type `int` and identified by strings. For a program P, we denote by $Var(P)$, $Var_{in}(P)$, and $Var_{out}(P)$ the variables, the input variables, and the output variables of P, respectively. Those in $Var_{in}(P)$ must be explicitly initialized before the execution of P.

Remark 2.2. Our imperative LOOP language has more basic functions and relations than that in [3, 7]. But as Moschovakis has shown in [18], the lack of expressivity is due to the recursion schema rather than to the chosen set of basic functions.

For the sake of simplicity, we will write $e + 1$ (resp. $e - 1$) instead of `succ`(e) (resp. `pred`(e)) and `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 the `elsif` keyword when the 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 as the first one and only once.

2.1.1. The *min* example

All along the paper we will illustrate our purpose with the *min* problem (see the Introduction for motivations). It is the usual one, with $pred(0) = 0$. Consider the following LOOP program (let us call it `inf`) which computes the minimum of two natural integers n and m.

An `inf` LOOP program :

```
$res := x_1$;
loop $x_1$ do
   $x'_2 := x'_2 - 1$;
   if $x'_2 = 0$ then
      $res := x_2$;
   endif;
endloop;
```

where x_1 and x_2 are the input variables initialized by n and m, respectively, and res is the output variable.

Lemma 2.1. The `inf` LOOP program computes the $\min$ function with the time complexity $O(x_1)$ (see further for the formal definition of time complexity).

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

2.2. A conservative extension LOOP$_{exit}$

We extend the language with the `exit` command which stops the program execution in the moment it arrives at it:

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

For a real programming language we might have choosen a more subtle behavior (see Section 4 for a discussion on this subject).

2.3. Operational semantics

The operational semantics of LOOP$_{exit}$ 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 (such a rule is called a one step reduction or one step rewriting). The $\oplus$ operation on environments resolves clashes by choosing the value given by its second argument. Let com_1, com_2 and com_s be lists of commands:

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

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

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

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

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

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

Here, 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 adequate to a program P if all variables of P appear in the domain of env.

Let y be a variable and env 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 whether we have $y = x$ or not.

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

Remark 2.3. In our operational semantics, the cost of the equality test and of the evaluation of a conditional ($valueOf$) is for free, regardless of the complexity of the boolean formula.

Let us call $((\alpha, env), P)$ a *state*. A state is *terminal* if no (reduction) rule is applicable to it (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 $\Rightarrow$ which leads from a state $((\alpha, env), P)$ to a terminal state $((\alpha', env'), P')$. We denote by $\Rightarrow^n$ the nth iteration (power) of $\Rightarrow$ and $\Rightarrow^*$ its reflexive and transitive closure.

Remark 2.4. The above reduction rules are deterministic.

2.4. Some theoretical results

Theorem 2.1. A function f is primitive recursive if and only if it is computable by a LOOP program.

Proof:
See [11] for a proof. □

Theorem 2.2. No LOOP program computes $\min(x, y)$ in $O(\min(x, y))$ steps.

Proof:
See [7] for a proof. □

Lemma 2.2. 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 by 0 and substitute $v_{exit} := 1;$ for all the `exit` commands in P_e. Finally, in order to get P, replace each command com in P_e by `if` $v_{exit} = 0$ `then` com `endif`;. □

Corollary 2.1. A function f is primitive recursive if and only if it is computable by a LOOP$_{exit}$ program.

2.4.1. The *min* example (continued)

The inf$'$ LOOP$_{exit}$ program :

```
$res := x_1$;
loop $x_1$ do
  $x'_2 := x'_2 - 1$;
  if $x'_2 = 0$ then
    $res := x_2$;
    exit;
  endif;
endloop;
```

Definition 2.1. The computation time of a LOOP$_{exit}$ program P is the map $T_P : \mathbb{N}^{|Var_{in}(P)|} \mapsto \mathbb{N}$ giving the number of one step rules executed during the computation of P for each concrete initialization.

Proposition 2.1. There exists a LOOP$_{exit}$ program which computes the $\min$ function in $O(\min)$ time.

From this result we notice that the LOOP$_{exit}$ programming language is more powerful than the LOOP language. Indeed, one can also prove that the *Stein*'s algorithm is programmable in the LOOP$_{exit}$ language with the good time complexity ($O(\log_2(x) + \log_2(y))$). See the program in Annex A. Moreover, functions with piecewise linear complexity from [18] are also programmable with the good time complexity in LOOP$_{exit}$.

Put otherwise, LOOP$_{exit}$ allows us to implement more efficient algorithms for primitive recursive functions than LOOP does. In order to formalize this comparison in the following sections, we shall have to define and use very carefully the notion of primitive recursive algorithm.

We end this section with the following simple fact about the LOOP programs.

Lemma 2.3. For every primitive recursive function $f : \mathbb{N}^k \to \mathbb{N}$, there exists a LOOP program Q such that $T_Q \geq f$.

Proof:
Let P be a LOOP program computing f and r a variable whose value is $f(v_1, \ldots, v_k)$ at the end of execution of P, where $(v_1, \ldots, v_k)$ are the values of the input variables of P. Just add the loop

```
loop $r$ do endloop;
```

to the end of P to get Q. □

3. Primitive recursive algorithms

We first recall some of the main concepts defined in [16], then we add some new definitions for later use.

3.1. Gurevich algorithms

A **vocabulary** Υ is a finite collection of function names, each one with a fixed arity. The **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 $\overline{u} = (u_1, ..., u_k)$ a k-tuple of terms, then $f(\overline{u}) = f(u_1, ..., u_k)$ 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. The elements of S are also considered as the elements of X. The **value** $val(u, X)$ of a term u in a structure X is also defined by induction: if u is a nulary function name, then $val(u, X)$ is the interpretation of u over the base set of X; if $u = f(u_1, ..., u_n)$, then $val(u, X)$ is the value of the interpretation of f on the n-tuple of arguments $(val(u_1, X), ..., val(u_n, X))$. 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 $(f, \overline{a}, b)$ means to replace the current content of the 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 means to execute simultaneously all the updates in the set; to execute an inconsistent one means to do nothing. The result of the execution of an update set Δ over X will be denoted by $X + \Delta$. It is proved in [16] 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 $Y = X + \Delta$.

The so called (ASM) programs by which we are going to manipulate the Υ-structures will be constructed as the rules over Υ according to the following definition of the three fundamental rule forms:

Definition 3.1. A **rule** over a vocabulary Υ has one of the three following forms:

1. $f(u_1, ..., u_n) := u_0$

2. **par**
```
    $R_1$
    ⋮
    $R_k$
endpar
```

3. **if** g **then**
```
    $R$
endif
```

where $f \in \Upsilon$ has arity n, $u_0, u_1, ..., u_n$ 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** over a structure X a rule of the first form, called an **update rule**, compute the values $a_i = val(u_i, X)$ for $i = 0, ..., n$ and then obtain a structure Y by executing the associated update $(f, (a_1, ..., a_n), a_0)$.

To fire over X a rule of the second form , called a **par rule**, fire simultaneously over X all the rules in the block. If the block is empty, then the rule is also called a **skip rule** and it has no effect when fired.

To fire over X a rule of the third form , called a **conditional rule**, first evaluate the guard g, and, if it is *true* then fire R, else do nothing.

The result of firing a rule R over a structure X is a structure denoted by $\Delta(R, X)$. An (ASM) **program** Π is a rule and the mapping τ_Π is defined for any Υ-structure X by: $\tau_\Pi(X) = \Delta(\Pi, X)$.

Definition 3.2. 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 $S(A)$ of Υ_A-structures (also called **states**) closed under isomorphisms and under the map τ_{Π_A}
- a subset $I(A) \subseteq S(A)$ closed under isomorphisms.

The restriction of τ_{Π_A} to $S(A)$ is usually denoted by τ_A. For every $X_0 \in I(A)$ 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 $S(A) = S(A')$, $I(A) = I(A')$, and if they have the same runs.

It is convenient to specify the functions of the vocabulary as **dynamic** or **static**, depending wether they can be modified by the ASM or not. Another distinction is made over dynamic functions: some of them are called **in** functions or **out** functions, in order 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, denoted by $in(A)$ and $out(A)$. Y. Gurevich says also that the vocabulary of an ASM always contains the nulary function names $true$, $false$, $undef$, the names of usual boolean operations, the unary function name $boole$, and the binary function name denoted by the equality sign =. This set of function names is called the **logic part** of the vocabulary.

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

$$val(U, X) = (val(u_1, X), ..., val(u_n, X)) \text{ if } T = (u_1, ..., u_n),$$

and to the sets of states

$$val(U, \{X_1, ..., X_k\}) = \{val(U, X_1), \dots, val(U, X_k)\}.$$

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 using the following definition:

Definition 3.3. Let A be an ASM and X be a state in $S(A)$. We define the partial map $\#step_X : S(A) \to \mathbb{N}$ by

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

Notice that $\#step_X$ is undefined on the set of states unreachable from X, while 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 on whether the runs it belongs to are finite or not).

Another important fact from [16] that we need to recall here is the existence, for any ASM A, of an equivalence relation of finite index dividing $S(A)$ into the classes $C_1, \dots, C_n$ recognizable by suitable corresponding boolean expressions $g_1, \dots, g_n$ as follows:

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

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

```
par
  if $g_1$ then
    $R_1$
  endif
  ⋮
  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 $X + \Delta(R_i, X) = \tau_A(X)$.

Remark 3.1. One can always extend the previous ASM inserting at the end a rule with a guard complementary of the previous guards, and a skip rule. This is feasible for normal form ASM at least.

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

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

Definition 3.4. Let A be an ASM and T a finite sequence of terms defined over Υ_A. We say that T is **characteristic** for $I(A)$ if

$$\forall X_0, X_0' \in I(A), val(U, X_0) = val(U, X_0') \Rightarrow X_0 = X_0'$$

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

3.2. Primitive recursive ASMs

The aim of this section is to give a formal definition of a class of algorithms we call *primitive recursive algorithms*. The following presentation is similar to that of the arithmetical ASMs ([12]).

According to Y. Gurevich's thesis (ASMs are algorithms, [16]) it is sufficient to define the concept of *primitive recursive* ASM.

The first thing to define is the vocabulary for the *primitive recursive* ASMs. Since we want them to treat the *primitive recursive* functions, and these take their values in $\mathbb{N}$, we will restrict the interpretations of the vocabulary for the ASMs so as to render them $\mathbb{N}$-typed.

Remark 3.2. We restrict the set of static functions to the constants, the successor and the predecessor functions. If we extend the vocabulary in the ASM framework, we need to associate the same functions (and same *extensional* semantics) in the framework of the LOOP language. We recall, as noticed by Y. Gurevich and Y. Moschovakis [19], that the notion of algorithm is not absolute but relative to a given set of functions.

Of course, one may construct algorithms that use some extra data types, like lists, to be efficient (an example is given by R. David in [9]). If the data type is used to store some partial results (it's then representable by a set of dynamics functions) we need to construct its counterpart in the LOOP language and to give an operational semantics for their evaluation; the operational semantics needs to be closed to the one of the ASM (where any evaluation is given *for free*).

In the sequel we will only focus on primitive arithmetical functions.

Definition 3.5. The $\mathbb{N}$-typed ASMs are the ASMs whose non-logic part of the vocabulary is interpretable only by functions taking their values in $\mathbb{N}$.

Definition 3.6. An $\mathbb{N}$-typed ASM A is called **basic** or **arithmetical** if the non-logic part of its vocabulary is reduced to:

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

where:

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

One can notice that (see [12]):

Proposition 3.1. The basic ASMs are Turing-complete.

In order to specify among the arithmetical ASM the *primitive recursive* ones, we need an adequate notion of complexity:

Definition 3.7. Let A be a **basic** ASM. The **complexity** c_A of A is a function defined over the set

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

by

$$c_A(val(in(A), X_0)) = \#step_{X_0}(Z_0),$$

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

Remark 3.3. A $\mathcal{PR}$ function is necessarily defined as a function from $\mathbb{N}^n$ to $\mathbb{N}$. So, to be $\mathcal{PR}$, c_A must be defined over $\mathbb{N}^n$, and, imposing A to be $\mathbb{N}$-typed ensures $val(in(A), \mathcal{I}(A))$ to be completely defined over $\mathbb{N}$.

Hence a *primitive recursive* ASM is an arithmetical ASM whose complexity is a *primitive recursive* function.

Definition 3.8. A **basic** or arithmetical ASM A is $\mathcal{PR}$ if c_A is $\mathcal{PR}$. We denote by $\mathcal{PR}$-ASM the set of *primitive recursive* ASMs.

3.3. The min problem

An `inf` ASM-PR :

```
if $x = 0$ then
    $res := n$
if $\neg(x = 0) \wedge y = 0$ then
    $res := m$
if $\neg(x = 0) \wedge \neg(y = 0)$ then
    $x := x - 1$
    $y := y - 1$
```

Remark 3.4. We could have defined the $\mathcal{PR}$-ASMs as the set of couples (A, f), where A is a basic ASM, $f \in \mathcal{PR}$, and f is the greatest running time of A. For instance, we can define (A_{Ack}, min^2) to be an $\mathcal{PR}$-ASM ($A_{Ack}(n, m)$ is a basic ASM computing the Ackerman function on n, m and we run it with at most $min^2(n, m)$ steps).

3.4. LOOP$_{exit}$ programs compute Basic PR-ASM

The objective 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 and respects its complexity in some sense.

Since an ASM is being executed until it reaches a fixed point, we must choose a length of run that is consistent with our programming language (which is limited to primitive recursive functions).

The solution is, intuitively, to consider two processes in parallel, one of which implements the algorithm, while the other one *counts* the number of times the algorithm is executed.

ASM rules are sometimes far from usual commands in programming language. The most important difference is the simultaneous execution of rules. We also consider only the number of update rules fired as complexity; evaluation of expressions has no cost (nor in update rule nor in condition of guards). Recall that the number of updates is bounded (in ASM the cost of fire is only one).

We consider in the following two kinds of complexity: the one due to the simulation of the simultaneous execution of rules and the second due to usual complexity of algorithms T_A (the number of updates at each step).

We denote T_A^{seq} the cost of the process that allows to simulate the features of an ASM A (simultaneous execution and test for a fix point) into a sequential programming language.

In what follows, d denotes the number of dynamic variables and g the number of guards in the ASM in question.

3.4.1. The heart of a program

The basic ASM programs allow us to construct certain LOOP_{exit} programs without loops 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.

If A is an ASM and P is a LOOP_{exit} program, we denote by $env(X)$ the set $\{(v, val(v, X)) \mid v \in Var(A)\}$. In this way, env allows us to associate environments of P from a state.

Proposition 3.2. Let τ_A be a basic ASM program with d dynamic variables. Then there exists a LOOP_{exit} program P_A with $2d$ variables such that, for any two distinct states $X, Y \in S(A)$ with $\tau_A(X) = Y$ and m variables updated, it holds that

$$((0, env), P_A) \Rightarrow^{\leq 2d+g} ((\alpha, env'), \epsilon) \text{ with } env(Y) \subseteq env' \text{ and } \alpha \in \{0, 1\}$$

and g is the number of guards of τ_A. With $T_A^{seq} = d + g + d$ (with the first d for saving old values of variables and the second d for the test of fix point) and $T_A = d$ (the maximum number of updates), we have $T \leq T_A + T_A^{seq}$.

Proof:
For any basic 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:

1. For each rule: we have the program:

```
if $g_i$ then
    $x_1 := e_1$
    $\vdots$
    $x_k := e_k$
endif
```

```
if $g_i$ then
    $x'_1 := x_1;\ x'_2 := x_2;\ \dots x'_k := x_k;$
    $x_1 := e_1[\vec{x'}/\vec{x}];$
    $x_2 := e_2[\vec{x'}/\vec{x}];$
    $\vdots$
    $x_k := e_k[\vec{x'}/\vec{x}];$
endif
```

2. if the rule is **if** g **then skip endif** then P is `if` g `then exit endif`.

3. and finally the whole P program is encapsulated by

```
if $\neg(x_1 = x'_1 \wedge \cdots \wedge x_n = x'_n)$ then
    P;
else
    exit;
endif
```

for all $x_i \in Var(A)$.

Remark 3.5. Some auxiliary variables are needed to simulate a simultaneous execution of a set of updates. The expression $x_i := e_i[x'_1/x_1][x'_2/x_2]\ldots[x'_k/x_k]$; denotes sequential substitutions.

Remark 3.6. We may have several true guards fired in the ASM; in our case we can not have any clashes in a set of updates. Then, since there is n variables in Π_A, there are at most $2n$ assignments executed by the LOOP$_{exit}$ program.

Remark 3.7. The complexity is preserved due to our model of computation: the cost of the equality test is neglected.

□

3.4.2. The min example

Following the previous transformation we obtain the LOOP$_{exit}$ program (without `loop`)

The $P_{\texttt{inf}}$ heart-program :

```
if $\neg(x = x' \wedge y = y' \wedge res = res')$ then
  if $x = 0$ then
    $res' := res$;
    $res := n$;
  endif;
  if $\neg(x = 0) \wedge y = 0$ then
    $res' := res$;
    $res := m$;
  endif;
  if $\neg(x = 0) \wedge \neg(y = 0)$ then
    $x' := x$;
    $y' := y$;
    $x := x' - 1$;
    $y := y' - 1$;
  endif;
else
  exit;
endif
```

We have constructed the first one of the two intended processes, that which must be iterated sufficiently many times.

3.4.3. Insertion of LOOP in LOOP$_{exit}$

We are now ready to consider the second process, whose role is to execute the heart-program in a bounded time. In order to do this, we inject in a program P that heart-program.

Definition 3.9. 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 the induction on the length of P:

- if P is $x := e;\ com_s$ then

$$P[Q] \text{ is } Q\ x := e;\ com_s[Q]$$

- if P is $\texttt{if}\ e\ \texttt{then}\ com_1\ \texttt{else}\ com_2\ \texttt{endif};\ com_s$ then

$$P[Q] \text{ is } Q\ \texttt{if}\ e\ \texttt{then}\ com_1[Q]\ \texttt{else}\ com_2[Q]\ \texttt{endif};\ com_s[Q]$$

- if P is $\texttt{loop}\ expr_{int}\ \texttt{do}\ com\ \texttt{endloop};\ com_s$ then

$$P[Q] \text{ is } Q\ \texttt{loop}\ expr_{int}\ \texttt{do}\ com[Q]\ Q\ \texttt{endloop};\ com_s[Q]$$

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

Lemma 3.1. Let A be a PR-ASM and X, Y be two distinct 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 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 $l \leq 2d+g+1$ (where d is the number of variables of P_A and g is the number of guards).

Proof:
Consider the following five distinct cases for P, using Proposition 3.2:

- if $((0, env_P), v := e;\ com_s)$
  $\Rightarrow ((0, env_P \oplus \{(v, valueOf(e))\}), com_s)$
  then $((0, env_P \cup env_{P_A}), P_A\ v := e;\ com_s[P_A])$
  $\Rightarrow^l ((0, env_P \oplus \{(v, valueOf(e))\} \cup env'_{P_A}), com_s[P_A])$

- with $valueOf(b) = true$,
  if $((0, env_P), \texttt{if}\ b\ \texttt{then}\ com_1\ \texttt{else}\ com_2\ \texttt{endif};\ com_s)$
  $\Rightarrow ((0, env_P), com_1\ com_s)$
  then $((0, env_P \cup env_{P_A}), P_A\ \texttt{if}\ b\ \texttt{then}\ com_1[P_A]\ \texttt{else}\ com_2[P_A]\ \texttt{endif};\ com_s[P_A])$
  $\Rightarrow^l ((0, env_P \cup env'_{P_A}), com_1[P_A]\ com_s[P_A])$

- with $valueOf(b) = false$,
  if $((0, env_P), \texttt{if}\ b\ \texttt{then}\ com_1\ \texttt{else}\ com_2\ \texttt{endif};\ com_s)$
  $\Rightarrow ((0, env_P), com_2\ com_s)$
  then $((0, env_P \cup env_{P_A}), P_A\ \texttt{if}\ b\ \texttt{then}\ com_1[P_A]\ \texttt{else}\ com_2[P_A]\ \texttt{endif};\ com_s[P_A])$
  $\Rightarrow^{l+1} ((0, env_P \cup env'_{P_A}), com_2[P_A]\ com_s[P_A])$

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

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

□

Therefore, starting with an *algorithm* one can construct a program in LOOP$_{exit}$ that preserves the complexity.

Theorem 3.1. Let f be a function from $\mathbb{N}^k$ to $\mathbb{N}$ and A be a PR-ASM which computes f. Then there exists a LOOP$_{exit}$ program computing f whose complexity belongs to $O(c_A)$ ($\leq c_A \times (3d + g + 1)$).

Proof:
By Lemma 3.2, each execution of P_A simulates (in at most $2n + c + 1$ steps) one run step of A. From Lemma 3.1, we know that the 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 to say, $T_P \geq c_A$ (such an inequality induces $Var_{in}(P) = in(A)$). The existence of such a program is ensured by Lemma 2.3, since A being a PR ASM implies that c_A is a PR function. Now it is easy to conclude that the desired LOOP$_{exit}$ program $P_{A,f}$ is:
$Var_{in}(P) := Var_{in}(P_A);$
$P[P_A]$
with $Var_{in}(P_{A,f}) = Var_{in}(P_A)$ and $Var_{out}(P_{A,f}) = Var_{out}(P_A)$.
Clearly $T_{P_{A,f}} \leq (3d + g + 1) \times c_A \in O(c_A)$ since the `exit` command stops the program as soon as the run is simulated. □

4. Conclusion and comments

This article is devoted to the problem of implementing *good* algorithms by restricted programming languages.

Inspired by the idea of Y. Gurevich to identify algorithms as abstract state machines (ASMs), we have constructed a large class of algorithms that we call *primitive recursive algorithms*, identifying them with a special class of AMSs.

As a counterpart of this class, we have specified a programming language (LOOP$_{exit}$) that captures all algorithms previously defined. This leads us to a concept that may be called *algorithmic completeness*:

Definition 4.1. Given a class A of algorithms, a programming language P is *A-complete* if all algorithms in A may be adequately implemented in P with respect to their the complexity.

One can notice that our class of algorithms includes the good algorithm for the min function and the *Stein* algorithm for the gcd function. We recall that those two algorithms have no adequate implementation in certain programming languages (see [3, 4, 10, 15, 18, 13, 7] for some particular results).

This suggests that $\textsc{Loop}_{exit}$ is a good programming language (from a complexity point of view). As for our definition of primitive recursive algorithms, it still may be discussed. The most obvious problem left open by this paper is the following:

Open problem 1. Is there an algorithm for a primitive recursive function which can not be implemented in $\textsc{Loop}_{exit}$?

As mentioned in section 3.4, an algorithm can use more sophisticated data types than natural numbers and ASM model has no problem to represent them by n-ary dynamics functions. But they are very difficult to represent in usual programming language except for data type representable by array indexed by integer. Usually, those functions are represented by Map but, as far as we know, there is no way to initialized them but constants.

Moreover, this work was done in an imperative framework, but using [7, 6], it may be interesting to define a total functional language which matches in efficiency the $\textsc{Loop}_{exit}$ 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, Polonovski Emmanuel and Valarcher Pierre. Extending the LOOP language with Higher-Order Procedural variables *in ACM-Transactions on Computational Logic*, (2008)

[7] Crolard Tristan, Lacas Samuel and Valarcher Pierre. On the expressive power of the For-language *in Nordic Journal of Computing* **13**, (2006)

[8] Dancanet Denis and Brookes Stephen. Programming language expressiveness and circuit complexity. *In: International Conference on the Mathematical Foundations of Programming Semantics* (1996).

[9] David René. Un algorithme primitif récursif pour la fonction Inf. *Compte rendu á l'académie des Sciences* **317** (1993).

[10] David René. On the asymptotic behaviour of primitive recursive algorithms. *Theoretical Computer Science* **266**(1-2) (2001) 159–193.

[11] Davis Martin D., Sigal Ron and Weyuker Elaine J. *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science.* Academic Press (2nd edition) (1994).

[12] Dershowitz Nachum and Gurevich Yuri. A Natural Axiomatization of Church's Thesis *Microsoft Research Technical Report* MSR-TR-2007-85, July 2007.

[13] van den Dries Lou. Generating the greatest common divisor, and limitations of primitive recursive algorithms. *Found. Comput. Math.* 3 (2003) 297-3224.

[14] Escardo Martin H. On Lazy Natural Numbers with Applications to Computability Theory and Functional Programming. *SIGACT News* **24**(1) (1993) 60–67.

[15] Fredholm Daniel. Computing Minimum with Primitive Recursion over Lists. *Theoretical Computer Science* **163**(1-2) (1996) 269–276.

[16] Gurevich Yuri. Sequential Abstract State Machines capture sequential algorithms. *ACM Trans. Computational Logic* **1**(1) (2000) 77–111.

[17] Meyer Albert R. and Ritchie Dennis M. The complexity of loop programs. *Proceedings of the ACM 22nd National Conference*, ACM Press (1967) 465–469.

[18] Moschovakis Yannis N. On primitive recursive algorithms and the greatest common divisor function. *Theoretical Computer Science* **301**(1-3) (2003) 1–30.

[19] Moschovakis Yannis N. and Paschalis V. Elementary algorithms and their implementations. *New Computational Paradigms*, ed. S. B. Cooper, Benedikt Lowe and Andrea Sorbi, Springer, (2008), pp. 81 - 118.

[20] Peter Roza. *Recursive Functions.* Academic Press (1968).

[21] 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.

[22] 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.

A. Stein's program

```
$x' := x$;
$y' := y$;
loop $x'$ do
  if $x = y \vee x = 0 \vee y = 0$ then
    $res := res * x$;
    exit;
  endif;
  if $even(x) \wedge even(y)$ then
    $x := x/2$;   $y := y/2$;
    $res := res * 2$;
  endif;
  if $even(x) \wedge \neg even(y)$ then
    $x := x/2$;
  endif;
  if $\neg even(x) \wedge even(y)$ then
    $y := y/2$;
  endif;
  if $\neg even(x) \wedge \neg even(y) \wedge (x > y)$ then
    $x := x - y$;
  endif;
  if $\neg even(x) \wedge \neg even(y) \wedge (x < y)$ then
    $y := y - x$;
  endif;
endloop;
loop $y'$ do
  if $x = y \vee x = 0 \vee y = 0$ then
    $res := res * x$;
    exit;
  endif;
  if $even(x) \wedge even(y)$ then
    $x := x/2$;   $y := y/2$;
    $res := res * 2$;
  endif;
  if $even(x) \wedge \neg even(y)$ then
    $x := x/2$;
  endif;
  if $\neg even(x) \wedge even(y)$ then
    $y := y/2$;
  endif;
  if $\neg even(x) \wedge \neg even(y) \wedge (x > y)$ then
    $x := x - y$;
  endif;
  if $\neg even(x) \wedge \neg even(y) \wedge (x < y)$ then
    $y := y - x$;
  endif;
endloop;
```