Studies in Weak Arithmetics

**Patrick Cégielski, Ali Enayat,
Roman Kossak**

April 5, 2016

**CENTER FOR THE STUDY
OF LANGUAGE
AND INFORMATION**

Contents

Foreword vii
Luc Hittinger

Introduction ix
Patrick Cégielski, Charalampos Cornaros, Costas Dimitracopoulos

1 An Imperative Language Characterizing PTIME Algorithms 1
Yoann Marquer and Pierre Valarcher

April 5, 2016

Foreword

Luc Hittinger[1]

I am very proud that the publishing collaboration between Stanford University, more precisely CSLI Publications, and the University Paris 12 (*Pôle de Recherche et d'Enseignement Supérieur Paris-Est*), continues. After the first Lecture Notes (number 196 in 2010) devoted to Studies in Weak Arithmetics and two more volumes (numbers 190 and 194 in 2011) of translation in French of select papers by Donald Knuth, the famous computer scientist who has inspired much work in our laboratory LACL (*Laboratoire d'Algorithmique, Complexité et Logique*), a second Lecture Notes devoted to Studies in Weak Arithmetics is being published. This time the volume contains papers based on lectures given during JAF31—the thirty first meeting of the *Journées sur les Arithmétiques Faibles* conferences that began in Lyon in 1990.

Although one of the successors of the original University of Paris, created in 1100, we like to consider our university as born in 1970 with the Faculty of Medicine and the Institute of Technology. At that time the aim was towards professional students, which continues to be our first objective. Fundamental science was essentially nonexistent in the initial plan—no masters in Mathematics or Theoretical Computer Science was offered, unlike what happened at many other universities in France and around the world. However, we had to adapt to the needs of our 32,000 students and our Laboratories of Mathematics (LAMA) and Theoretical Computer Science (LACL) were created; LAMA and LACL developed quickly and are now widely recognized by the international community of researchers.

[1]President of the University Paris-Est Créteil Val-de-Marne

Studies in Weak Arithmetics.
Patrick Cégielski, Ali Enayat,
Roman Kossak.
Copyright © 2016, CSLI Publications.

April 5, 2016

Introduction

Patrick Cégielski,[1] Charalampos Cornaros,[2]
Costas Dimitracopoulos[3]

The conference series, *Journées sur les Arithmétiques Faibles* (*Weak Arithmetics Days*, JAF), began in June 1990 when the first meeting was held at École Normale Supérieure in Lyon, France. That meeting has been followed by thirty more meetings, during which researchers from countries all over the world presented work and discussed research ideas concerning *Weak Arithmetics*, that is, the research area concerning, roughly speaking, the application of logical methods to Number Theory and related topics.

The latest meeting of the series, JAF31, took place from May 30 until June 1, 2012, on Samos, the island of Pythagoras, one of the most renowned mathematicians of all times. JAF31 was dedicated to the memory of Alan R. Woods, a colleague who died on December 11, 2011, having made significant contributions to the area of Weak Arithmetics and having participated is several meetings of the series.

The Steering Committee of the JAF series decided that a volume be published, containing papers based on lectures delivered during the meeting on Samos, as well as Alan's Ph.D. thesis, parts of which remained unpublished despite having motivated a tremendous amount of work by many distinguished researchers. While preparing the volume, which we undertook with enthusiasm, we invited more contributions by

[1]LACL, EA 4219, Université Paris-Est Créteil, France cegielski@u-pec.fr
[2]University of Aegean, Karlovassi, Greece kornaros@aegean.gr
[3]University of Athens, Greece cdimitr@phs.uoa.gr

Studies in Weak Arithmetics.
Patrick Cégielski, Ali Enayat,
Roman Kossak.
Copyright © 2016, CSLI Publications.

colleagues who had not been able to participate in JAF31 but wished to honor the memory of Alan R. Woods. After a proposal of Roman Kossak, we decided to also include in the volume another unpublished Ph.D. thesis that had contributed significantly to progress in the area of Weak Arithmetics, namely the thesis of Hamid Lesan, who died in 2006.

All papers published in the present volume were refereed by (at least) two referees each. We are grateful to the following colleagues, for having kindly helped us with the refereeing of the papers: Andrés Cordon-Franco, David Fernandez Duque, Alex Esbelin, Benedetto Intrigila, Francisco Félix Lara Martín, Vassilis Paschalis and Albert Visser. We are also grateful to Vassilis Paschalis, for having undertaken the typesetting of A. Woods' thesis and for having offered us valuable technical support.

We would like to express our gratitude to Alan's son, Robert, and daughter, Sarah, for their kind permission to include his thesis in this volume and for providing us with information concerning his life and career; we are also grateful to Hamid's wife, Lynda, and daughters, Sara and Leila, for granting us their kind permission to include his thesis and to George Wilmers, Hamid's thesis supervisor, for having prepared a note on Hamid's life and work.

We are grateful to Ch. Cornaros, E. Tachtsis and their team in Karlovassi, for organizing a memorable conference, and to all the sponsors of the meeting—University of Aegean, University of Athens, Regional Government of North Aegean (Mr. A. Yiakalis, Governor, and Mr. Th. Papatheofanous, Deputy Governor), Municipality of Samos (Mr. S. Thanos, Mayor, and Mr. P. Parianos, Deputy Mayor), Municipal Department of Karlovassi (Mr. S. Makris, Chairman), DOPONAS, ALPHA BANK, Energiaki Samou SA (Mr. Nikos Elenis), Mr. Evangelos Mytilinaios, Samaina Hotels, AB Shop & Go, DMGRAPHICS, EKTYPON and TO SXEDIO—for having provided generous financial and other support, without which the meeting would not have been so successful. Finally, we would like to express our deep gratitude to the Université Paris Est-Créteil, for having generously covered the cost of production of the present volume, and to Marie-Annick Le Traon (Université Paris Est Créteil–IUT de Sénart-Fontainebleau) for the design of the cover.

Internet site:

`http://lacl.fr/jaf/`

contains further information on "Journées sur les Arithmétiques Faibles".

1

An Imperative Language Characterizing PTIME Algorithms

Yoann Marquer[1] and Pierre Valarcher[2]

Abstract: Abstract State Machines of Y. Gurevich capture sequential algorithms, so we define the set of PTIME algorithms as the set of ASM programs computed in polynomial time (using timer-step principle). Then, we construct an imperative programming language PLoopC using bounded loop with break, and running with any data structure. Finally, we prove that PLoopC computes exactly PTIME algorithms in lock step (one step of the ASM is simulated by using a constant number of steps).

Context

The paper is devoted to answer the question:

> Is it possible to construct a programming language complete for the set of polynomial time algorithms, and no more?

The common definition of PTIME is the set of functions computable by a Turing Machine in polynomial time. This definition uses a step-timer principle (the polynomial bounding the running time), which can be criticized in two ways:

1. According to this definition, one-tape Turing Machines and two-tape Turing Machines should be equivalent. But the palindrome

[1]Laboratoire d'Informatique Algorithmique: Fondements et Applications, Universit Paris Diderot - Paris 7 yoann.apeiron.marquer@gmail.com

[2]Laboratoire Algorithmique, Complexit et Logique, IUT Snart-Fontainebleau, Universit Paris Est Crteil, pierre.valarcher@u-pec.fr

Studies in Weak Arithmetics.
Patrick Cégielski, Ali Enayat,
Roman Kossak.
Copyright © 2016, CSLI Publications.

recognition can be done in $O(n)$ steps with a two-tape Turing machine while requiring (see [24]) at least $O(n^2/log(n))$ steps with a one-tape Turing machine. So, the concrete implementation and its associated complexity (more specifically, the degree of the time complexity) depend on the considered model. Therefore, we need a more precise definition of polynomial time algorithms.

2. The second issue comes from the step-timer principle itself. It is not convenient for programmers to use an external function attached to the algorithm. The answer comes from *implicit complexity computation* frameworks. Since a long time, there are many attempts (like [2, 22] and [4]) to capture polynomial time algorithms. The main motivations are usually to find "pleasant" syntactical (and semantical) languages, to capture more algorithms than the previous models, or to give simpler method to decide if a program is computable in polynomial time.

We answer both criticisms by defining a convincing set of algorithms running in polynomial time (using the step-timer principle), and by constructing a programming language computing those algorithms:

Algorithms with Step-Timer Principle

From now half a century, there is a growing interest in defining formally the notion of sequential algorithms [20, 10], and some of these definitions allow to specify classes of algorithms[3] (in [11, 10] and [8]).

An axiomatic definition of the sequential algorithms is mapped to the notion of Abstract State Machine with a *strict lock-step* simulation[4]. The model of ASM is a kind of super Turing Machine that works not on simple tapes (with finite alphabets) but on multi–sorted algebras (see the point of view in [13]). A program is a finite set of rules that updates terms. It is shown in [17], that the expressive power of ASM lies not in control structures but in data structures, which are modeled within a first order structure. According to Gurevich's Thesis (see [12]), every algorithm can be computed step-by-step by an ASM. So, we will define p.6 the set $\texttt{ASM}_{\mathcal{P}}$ as the set of every ASM with polynomial running time.

The PLoopC language

Many attempts tries to restrict `Loop` imperative language (see [18]) in order to obtain relevant classes of algorithms.

[3] Even if there are parallel, distributed, real-time, bio-inspired or quantum algorithms, this paper focuses only on sequential algorithms.

[4] See [9] for a definition of simulation and strict lock-step

In [1, 23] a class APRA of primitive recursive algorithms is defined for the basic data structure of (unary) integer coming from the *abstract state machine* theory. From there, two imperative programming languages and one functional programming languages are proved to be **complete** for this set of algorithms. In other words, every algorithm defined in APRA can be written in those languages without lost of time complexity (the simulation is in $O(1)$).

For polynomial time, a `Loop` programming language is presented in [22], where some properties are found to capture PTIME on data structure such as stack, trees or graphs, and Neergaard introduced in [21] his polynomial time `PLoop` programming language `PLoop`, which uses stacks.

Following [1], we will define p.19 an imperative programming language `PLoopC`, and prove that `PLoopC` captures exactly the set $\texttt{ASM}_{\mathcal{P}}$ of polynomial time algorithms. We do that by following the Bellantoni and Cook's approach (see [2]) separating safe and normal variables. By restricting the use of iteration bounds to be inputs, we prove that `PLoopC` has polynomial time for every data structure.

The paper is organized as follows.

In the first section we briefly introduce Gurevich's framework of the sequential algorithms, and define p.11 the notion of *fair simulation*. In the second section we introduce the Gurevich's ASM model of computation, and our programming language `LoopC`. Moreover, we will prove p.20 that the sublanguage `PLoopC` has polynomial time.

The third section is devoted to prove p.35 that `ASM` fairly simulates `LoopC`, by using the notion of *graph of execution* and a translation of the ASM program into an imperative program. Therefore, because `PLoopC` is a sublanguage of `LoopC` with polynomial time, $\texttt{ASM}_{\mathcal{P}}$ fairly simulates `PLoopC`. Reciprocally, in the fourth section, we will prove p.25 that `PLoopC` fairly simulates $\texttt{ASM}_{\mathcal{P}}$, by using an imperative program translating one step of the ASM, and repeat it a sufficient number of times by using a translation of the complexity and a formula detecting the end of the execution.

Therefore, we will prove that `PLoopC` characterizes polynomial time algorithms.

Keywords. ASM, computability, imperative language, implicit complexity, polynomial time, sequential algorithms, simulation.

1 Sequential Algorithms

In [10] Gurevich introduced an axiomatic presentation of the sequential algorithms, by giving the three postulates of Sequential Time, Abstract States and Bounded Exploration. In our paper the set of "objects" satisfying these postulates is denoted by `Algo`.

We will introduce them briefly in the first subsection, as well as other notions from Gurevich's framework such as execution, time, structure and update. In the second subsection we will introduce our definition of a fair simulation between computation models.

1.1 Three Postulates

Postulate 1 (Sequential Time)**.** A sequential algorithm A is given by:

1. A set of states $S(A)$
2. A set of initial states $I(A) \subseteq S(A)$
3. A transition function $\tau_A : S(A) \to S(A)$

Remarque 1. *According to this postulate, two sequential algorithms A and B are the same (see [3]) if they have the same set of states $S(A) = S(B)$, the same set of initial states $I(A) = I(B)$, and the same transition function $\tau_A = \tau_B$.*

An **execution** of A is a sequence of states $\vec{X} = X_0, X_1, X_2, \dots$ such that:

1. X_0 is an initial state
2. For every $t \in \mathbb{N}$, $X_{t+1} = \tau_A(X_t)$

A state X_t of an execution is final if $\tau_A(X_t) = X_t$. An execution is **terminal** if it contains a final state. The duration of an execution is defined by the number of steps[5] done before reaching a final state:

$$time(A, X_0) =_{def} min\{t \in \mathbb{N} \mid \tau_A^t(X_0) = \tau_A^{t+1}(X_0)\}$$

Notice that if the execution $\vec{X}$ is not terminal then $time(A, X_0) = \infty$.

Remarque 2. *Two algorithms A and B have the same set of executions if they have the same set of initial states $I(A) = I(B)$ and the same transition function $\tau_A = \tau_B$. In that case, they can only be different on the states which cannot be reached by an execution.*

To state the second postulate, we need to introduce the notion of structure. Gurevich formalized the states of a sequential algorithm with first-order structures. A (first-order) **structure** X is given by:

[5] In the definition of *time*, f^i is the iteration of f defined by $f^0 = id$ and $f^{i+1} = f(f^i)$.

1. An infinite[6] set $\mathcal{U}(X)$ called the **universe** (or domain) of X
2. A finite set of function symbols $\mathcal{L}(X)$ called the **signature** (or language) of X
3. For every symbol $s \in \mathcal{L}(X)$ an **interpretation** $\overline{s}^X$ such that:
   (a) If c has arity 0 then $\overline{c}^X$ is an element of $\mathcal{U}(X)$
   (b) If f has an arity $\alpha > 0$ then $\overline{f}^X$ is an application: $\mathcal{U}(X)^\alpha \to \mathcal{U}(X)$

In order to have a uniform presentation, Gurevich considered constant symbols of the signature as 0-ary function symbols, and relation symbols R as their indicator function χ_R. Therefore, every symbol in $\mathcal{L}(X)$ is a function. Moreover, partial functions can be implemented with a special value *undef*.

This formalization can be seen as a representation of a computer data storage. For example, the interpretation $\overline{s}^X$ of the symbol s in the structure X represents the value in the register s for the state X.

The second postulate can be seen as a claim assuming that every data structure can be formalized as a first-order structure[7]. Moreover, since the representation of states should be independent from their concrete implementation (for example the name of objects), isomorphic states will be considered as equivalent:

Postulate 2 (Abstract States)**.** For every algorithm A,

1. The states of A are (first-order) structures with the same signature $\mathcal{L}(A)$
2. $S(A)$ and $I(A)$ are closed by isomorphism
3. The transition function τ_A preserves the universes and commutes with the isomorphisms

The symbols of $\mathcal{L}(A)$ are distinguished between:

1. $Dyn(A)$ the set of **dynamic symbols**, whose interpretation can change during an execution (as an example, a variable x)
2. $Stat(A)$ the set of **static symbols**, which have a fixed interpretation during an execution. They are also distinguished between:
   (a) $Init(A)$, the set of **parameters**, whose interpretation depends only on the initial state (as an example, two given integers m and n).

[6] Usually the universe is only required to be non-empty, we need the universe to be at least countable in order to define unary integers .

[7] We tried to characterize common data types (such as integers, words, lists, arrays, and graphs) in [16]. But we will not go into details, because this is not the point of this article.

The symbols depending on the initial state are the dynamic symbols and the parameters, so we call them the **inputs**.

The other symbols have a uniform interpretation in every state (up to isomorphism), and they are also distinguished between:

(b) $Cons(A)$ the set of **constructors** (*true* and *false* for the booleans, 0 and S for the unary integers, ...)

(c) $Oper(A)$ the set of **operations** ($\neg$ and $\wedge$ for the booleans, $+$ and $\times$ for the unary integers, ...)

The **size** of an element of the universe is the length of its representation (see [16] for more details), in other words the number of constructors necessary to write it. As an example, $|\overline{\neg\neg true}^X| = |\overline{true}^X| = 1$, $|\overline{1+2}^X| = |\overline{S(S(S(0)))}^X| = 4$ in the unary numeral system[8], and $|\overline{1+2}^X| = |\overline{11}^X| = 2$ in the binary numeral system.

The size of a state is the maximum (or the sum, which is equivalent because the signature is finite) of the size of its inputs:

$$|X| =_{def} \max_{f \in Dyn(A) \sqcup Init(A)} \{|f|_X\}, \text{ where } |f|_X =_{def} \sup_{a_i \in \mathcal{U}(A)} |\overline{f}^X(\vec{a})|$$

Definition 1 (Time Complexity)**.**

The algorithm A is $\mathcal{C}$-time if there exists $\varphi_A \in \mathcal{C}$ such that for all $X \in I(A)$:

$$time(A, X) \leq \varphi_A(|X|)$$

Let $\texttt{Algo}_{\mathcal{C}}$ be the set of $\mathcal{C}$-time algorithms. In particular, we denote by $\texttt{Algo}_{\mathcal{P}}$ the set of polynomial-time algorithms.

The logical variables are not used in this paper: every term and every formula is closed, and every formula is without quantifier. In this framework the **variables** are the 0-ary dynamic function symbols.

For a sequential algorithm A, let X be a state of A, $f \in \mathcal{L}(A)$ be a dynamic α-ary function symbol, and $a_1, \ldots, a_\alpha, b \in \mathcal{U}(X)$. $(f, a_1, \ldots, a_\alpha)$ denotes a location of X and $(f, a_1, \ldots, a_\alpha, b)$ denotes an **update** on X at the location $(f, a_1, \ldots, a_\alpha)$.

If u is an update then[9] $X \oplus u$ is a new structure of signature $\mathcal{L}(A)$ and universe $\mathcal{U}(X)$ such that the interpretation of a function symbol

[8] More generally, in the unary numeral system $|\overline{n}^X| = n + 1$.

[9] The update is denoted $\oplus$, not $+$ like in [12] or [17], because the associativity has no meaning here, and because we don't have the commutativity:

$$(X \oplus (x, 0)) \oplus (x, 1) \neq (X \oplus (x, 1)) \oplus (x, 0)$$

$f \in \mathcal{L}(A)$ is:

$$\overline{f}^{X \oplus u}(\vec{a}) =_{def} \begin{cases} b & \text{if } u = (f, \vec{a}, b) \\ \overline{f}^{X}(\vec{a}) & \text{else} \end{cases}$$

If $\overline{f}^{X}(\vec{a}) = b$ then the update $(f, \vec{a}, b)$ is trivial in X, because nothing has changed. Indeed, if $(f, \vec{a}, b)$ is trivial in X then $X \oplus (f, \vec{a}, b) = X$.

If Δ is a set of updates then Δ is **consistent** on X if it does not contain two distinct updates with the same location. If Δ is inconsistent, there exists $(f, \vec{a}, b), (f, \vec{a}, b') \in \Delta$ with $b \neq b'$, so the entire set of updates clashes:

$$\overline{f}^{X \oplus \Delta}(\vec{a}) =_{def} \begin{cases} b & \text{if } (f, \vec{a}, b) \in \Delta \text{ and } \Delta \text{ is consistent on } X \\ \overline{f}^{X}(\vec{a}) & \text{else} \end{cases}$$

If X and Y are two states of the same algorithm A then there exists a unique consistent set $\Delta = \{(f, \vec{a}, b) \mid \overline{f}^{Y}(\vec{a}) = b \text{ and } \overline{f}^{X}(\vec{a}) \neq b\}$ of non trivial updates such that $Y = X \oplus \Delta$. This Δ is the **difference** between the two sets and is denoted by $Y \ominus X$.

Let $\Delta(A, X) = \tau_A(X) \ominus X$ be the set of updates done by a sequential algorithm A on the state X.

During an execution, if an increasing number of updates are done (as is the case in example 1 with the parallel lambda-calculus) then the algorithm will be said massively parallel, not sequential. The two first postulates cannot ensure that only local and bounded explorations/changes are done at every step. The third postulate states that only a bounded number of terms must be read or updated during a step of the execution:

Postulate 3 (Bounded Exploration)**.**

For every algorithm A there exists a finite set T of terms (closed by subterms) such that for every state X and Y, if the elements of T have the same interpretations on X and Y then $\Delta(A, X) = \Delta(A, Y)$.

This T is called the **exploration witness** of A.

Gurevich proved in [12] that if $(f, a_1, \ldots, a_\alpha, b) \in \Delta(A, X)$ then $a_1, \ldots, a_\alpha, b$ are interpretations in X of terms in T. So, since T is finite there exists a bounded number of $a_1, \ldots, a_\alpha, b$ such that the update $(f, a_1, \ldots, a_\alpha, b)$ belongs to $\Delta(A, X)$. Moreover, since $\mathcal{L}(A)$ is finite there exists a bounded number of dynamic symbols f. Therefore, $\Delta(A, X)$ has a bounded number of elements, and for every step of the algorithm only a bounded amount of work is done.

1.2 Fair Simulation

A **model of computation** can be defined as a set of programs given with their operational semantics. In our paper we only study sequential algorithms, which have a step-by-step execution determined by their transition function. So, this operational semantics can be defined by a set of transition rules, as is the case in the following example:

Example 1 (The Lambda-Calculus).

The lambda calculus is defined by of a set of lambda terms (which are the "programs"), and a set of transformation rules (which are the "operational semantics"):

$$\begin{array}{rc} \text{Syntax of Programs:} & t =_{def} x \mid \lambda x.t \mid (t_1)t_2 \\ \beta\text{-reduction:} & (\lambda x.t_1)t_2 \to_\beta t_1[t_2/x] \end{array}$$

In order to be deterministic the strategy of the transition system must be specified. An example is the call-by-name strategy defined by context:

$$\begin{array}{rc} \text{Call-by-Name Context:} & C_n\{.\} =_{def} . \mid C_n\{.\}t \\ \text{Transition Rule:} & C_n\{(\lambda x.t_1)t_2\} \to_n C_n\{t_1[t_2/x]\} \end{array}$$

This rule can be implemented in a machine:

$$\begin{array}{rcc} \text{Operational semantics:} & t_1t_2 \star \pi \succ_0 & t_1 \star t_2, \pi \\ & \lambda x.t_1 \star t_2, \pi \succ_1 t_1[t_2/x] \star \pi & \end{array}$$

These notations and this machine are directly taken from Krivine's [15]. In this machine π is a stack of terms. The symbol $\star$ is a separator between the current program and the current state of the memory. $\succ$ represents one step of computation, where only substitutions have a cost, not explorations inside a term, as is the case in the contextual transition rule. Programs in the machine are closed terms, so final states have the form $\lambda x.t \star \varnothing$.

We will follow the notations $\star$ and $\succ$ in the definition of the operational semantics of imperative programs 7.

Notice that if the substitution is given as an elementary operation this model satisfies the third postulate, because only one term is pushed or popped per step. This is not the case with the lambda-calculus with parallel reductions. For example, with the term $t = \lambda x.(x)x(x)x$ applied to itself:

$$(t)t \to_p (t)t(t)t \to_p (t)t(t)t(t)t(t)t \to_p \dots$$

Indeed, at the step t exactly 2^{t-1} β-reductions are done, which is unbounded.

Sometimes, not only the simulation between two models of computation can be proven, but also their identity. As an example, Serge

Grigorieff and Pierre Valarcher proved in [13] that Evolving MultiAlgebras (a variant of the Gurevich's ASMs) can unify common sequential models of computation. For instance, a family of EMAs can not only simulate step-by-step the Turing Machines, it can also be literally identified to them. The same applies for Random Access Machines, or other common models.

But generally it is only possible to prove a simulation between two models of computation. In our framework, a computation model M_1 can simulate another computation model M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 producing in a "reasonable way" the "same" executions as those produced by P_2. The following two examples will detail what can be used in a "fair" simulation:

Example 2 (Temporary Variables).

In this example a programmer is trying to simulate a `loop` n $\{P\}$ command in an imperative programming language[10] containing `while` commands. The well-known solution is to use a temporary variable i in the new program:

$$i := 0; \texttt{while } i < n \ \{P; i := i + 1;\};$$

This simulation is very natural, but a fresh variable i is necessary.

Another example is to simulate the exchange $x \leftrightarrow y$ between two variables using a temporary variable:

$$v := x; x := y; y := v;$$

In any case, the signature $\mathcal{L}_1$ of the simulating program must be bigger than the signature $\mathcal{L}_2$ of the simulated program.

Notation 3. We follow the notation from [7], where $X|_{\mathcal{L}_2}$ denotes the **restriction** of the $\mathcal{L}_1$-structure X to the signature $\mathcal{L}_2$. The signature of $X|_{\mathcal{L}_2}$ is $\mathcal{L}_2$, its universe is the same than X, and every symbol $s \in \mathcal{L}_2$ has the same interpretation in $X|_{\mathcal{L}_2}$ than in X.

This notation is extended to a set of updates:

$$\Delta|_{\mathcal{L}} =_{def} \{(f, \vec{a}, b) \in \Delta \mid f \in \mathcal{L}\}$$

But fresh function symbols could be "too powerful", for example a dynamical unary symbol *env* alone would be able to store an unbounded amount of information. In order to obtain a fair simulation, we assume that the difference $\mathcal{L}_1 \setminus \mathcal{L}_2$ between both signatures is a set containing only a bounded number of variables (0-ary dynamical symbols).

The initial values of these **fresh variables** could be a problem if they depend on the inputs. For example, the empty program could

[10] We will define 6 the precise syntax of imperative programs.

compute any $f(\vec{n})$ if we assume that an output variable contains in the initial state the result of the function f on the inputs $\vec{n}$.

So, in this paper we use an initialization which depends[11] only on the constructors[12]. Because this initialization is independent (up to isomorphism) from the initial states, we call it a **uniform initialization**.

Example 4 (Temporal Dilation).

At every step of a Turing machine, depending on the current state and the symbol in the current cell:

- the state of the machine is updated
- the machine writes a new symbol in the cell
- the head of the machine can move left or right

Usually these actions are considered simultaneous, so only one step of computation is necessary to execute them. This is our classical model M_1 of Turing machines. But if we consider that every action requires one step of computation then we could imagine a model M_3 where three steps are necessary to simulate one step of M_1.

In other words, if we only observe an execution

$$\mathbf{X_0}, X_1, X_2, \mathbf{X_3}, X_4, X_5, \mathbf{X_6}, \dots$$

of M_3 every three steps (the observed states are bolded) then we will obtain an execution defined by $Y_t = X_{3\times t}$, which is an execution of M_1.

Imagine that M_1 and M_2 are implemented on real machines such that M_3 is three times faster than M_1. In that case if an external observer starts both machines simultaneously and checks their states at every step of M_1 then both machines cannot be distinguished.

In the following a (constant) **temporal dilation** d is allowed. We will say that the simulation is step-by-step, and strictly step-by-step if $d = 1$. Unfortunately, contrary to the previous example this constant may depend on the simulated program.

But this temporal dilation is not sufficient to ensure the termination of the simulation. For example, a simulated execution $Y_0, \dots, Y_t, Y_t, \dots$ could have finished, but the simulating execution

$$X_0, \dots, X_{d\times t}, X_{d\times t+1}, \dots, X_{d\times t+(d-1)}, X_{d\times t}, X_{d\times t+1}, \dots$$

may continue forever. So, an ending condition like $time(A, X) = d \times$

[11] Even the values of fresh variables in the initial states can be irrelevant. See the program P_{Π} 8 where the variables $\vec{v}$ are explicitly updated with the value of terms $\vec{t}$ before being read.

[12] See the program Π_P 15 where the boolean variable b_P is initialized with $true$ and the others with $false$.

$time(B, X) + e$ is necessary, and corresponds to the usual consideration for asymptotic time complexity.

Definition 2 (Fair Simulation)**.**

Let M_1, M_2 be two models of computation.

M_1 simulates M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 such that:

1. *$\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2)$, and $\mathcal{L}(P_1) \setminus \mathcal{L}(P_2)$ is a finite set of variables (with a uniform initialization)*

and there exists $d \in \mathbb{N} \setminus \{0\}$ and $e \in \mathbb{N}$ (depending only on P_2) such that, for every execution $\vec{Y}$ of P_2 there exists an execution $\vec{X}$ of P_1 satisfying:

2. *for every $t \in \mathbb{N}$, $X_{d\times t}|_{\mathcal{L}(P_2)} = Y_t$*

3. *$time(P_1, X_0) = d \times time(P_2, Y_0) + e$*

*If M_1 simulates M_2 and M_2 simulates M_1 then these models of computation are **algorithmically equivalent**, which is denoted by $M_1 \simeq M_2$.*

Remarque 3. *The second condition $X_{d\times t}|_{\mathcal{L}(P_2)} = Y_t$ implies for $t = 0$ that the initial states are the same, up to temporary variables.*

2 Models of Computation

In this section, Gurevich's Abstract State Machines are defined, and we use his theorem $\texttt{Algo} = \texttt{ASM}$ to get a constructive (from an operational point of view) occurrence of sequential algorithms. So, in the rest of the paper, the set of polynomial-time algorithms $\texttt{Algo}_{\mathcal{P}}$ will be the set $\texttt{ASM}_{\mathcal{P}}$ of ASMs with a polynomial time complexity.

We will define in the second part of the section the language `LoopC` from [1], and prove that a sublanguage `PLoopC` has polynomial time. We will prove in the next section that this sublanguage is complete for $\mathcal{P}$-time algorithms. In order to do so, we will prove a bisimulation between $\texttt{ASM}_{\mathcal{P}}$ and `PLoopC`, using the same data structures in these two models of computation.

2.1 Abstract State Machines

Without going into details, the Gurevich's Abstract State Machines (`ASM`) require only the equality $=$, the constants $true$ and $false$, the unary operation $\neg$ and the binary operations $\wedge$.

Definition 3 (ASM programs)**.**

$$\begin{aligned}\Pi =_{def} & f(t_1,\dots,t_\alpha) := t_0 \\ & \mid \texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif} \\ & \mid \texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar}\end{aligned}$$

where f is a dynamic α-ary function symbol, $t_0, t_1, \dots, t_\alpha$ are closed terms, and F is a formula.

Notation 5. For $n = 0$ a `par` command is an empty program, so let `skip` be the command `par endpar`. If the `else` part of an `if` is a `skip` we only write `if` F `then` Π `endif`.

The sets $Read(\Pi)$ of terms read by Π and $Write(\Pi)$ of terms written by Π can be used to define the exploration witness of Π. But we will also use them in the rest of the article, especially to define the μ-formula F_Π p.34.

$Read(\Pi)$ is defined by induction on Π:

$$\begin{aligned}Read(f(t_1,\dots,t_\alpha) := t_0) & \\ & =_{def} \{t_1,\dots,t_\alpha,t_0\} \\ Read(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}) & \\ & =_{def} \{F\} \cup Read(\Pi_1) \cup Read(\Pi_2) \\ Read(\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar}) & \\ & =_{def} Read(\Pi_1) \cup \dots \cup Read(\Pi_n)\end{aligned}$$

$Write(\Pi)$ is defined by induction on Π:

$$\begin{aligned}Write(f(t_1,\dots,t_\alpha) := t_0) & \\ & =_{def} \{f(t_1,\dots,t_\alpha)\} \\ Write(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}) & \\ & =_{def} Write(\Pi_1) \cup Write(\Pi_2) \\ Write(\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar}) & \\ & =_{def} Write(\Pi_1) \cup \dots \cup Write(\Pi_n)\end{aligned}$$

Remarque 4. *The exploration witness of Π is the closure by subterms of $Read(\Pi) \cup Write(\Pi)$ and not only $Read(\Pi)$ because the updates of a command could be trivial.*

As said p.8, defining the syntax of programs is not enough to obtain a model of computation, we still have to define their semantics. An ASM program Π determines a transition function $\tau_\Pi(X) =_{def} X \oplus \Delta(\Pi, X)$, where the set of updates $\Delta(\Pi, X)$ done by Π on X is defined by induction:

Definition 4 (Operational Semantics of ASMs)**.**

$$\begin{aligned}\Delta(f(t_1,\dots,t_\alpha) := t_0, X) &=_{def} \{(f, \overline{t_1}^X, \dots, \overline{t_\alpha}^X, \overline{t_0}^X)\} \\ \Delta(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}, X) &=_{def} \Delta(\Pi_i, X)\end{aligned}$$

$$\textit{where } i = \begin{cases} 1 \textit{ if } F \textit{ is true on } X \\ 2 \textit{ else} \end{cases}$$

$$\Delta(\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar}, X) =_{def} \Delta(\Pi_1, X) \cup \dots \cup \Delta(\Pi_n, X)$$

Notice that the semantics of the `par` is a set of updates done simultaneously, contrary to the imperative language defined in the next subsection, which is strictly sequential.

Remarque 5. *For every states X and Y, if the terms of Read(Π) have the same interpretation on X and Y then $\Delta(\Pi, X) = \Delta(\Pi, Y)$.*

We can now define the set `ASM` of Abstract States Machines:

Definition 5. *An Abstract State Machine M with signature $\mathcal{L}$ is given by:*

- *an* `ASM` *program Π on $\mathcal{L}$*
- *a set $S(M)$ of $\mathcal{L}$-structures closed by isomorphisms and τ_Π*
- *a subset $I(M) \subseteq S(M)$ closed by isomorphisms*
- *an application τ_M, which is the restriction of τ_Π to $S(M)$*

For every sequential algorithm A, the finiteness of the exploration witness in the third postulate allows us (see [12]) to write a finite `ASM` program Π_A, which has the same set of updates than A for every state. Every program Π_A obtained in this way has the same form, which we call the **normal form**:

$$\begin{array}{rl} \texttt{par} & \texttt{if } F_1 \texttt{ then } \Pi_1 \\ \| & \texttt{if } F_2 \texttt{ then } \Pi_2 \\ & \vdots \\ \| & \texttt{if } F_c \texttt{ then } \Pi_c \\ \texttt{endpar} & \end{array}$$

where F_i are "guards", which means that for every state X one and only one F_i is *true*, and the programs Π_i have the form

$$\texttt{par } u_1 \| \dots \| u_{m_i} \texttt{ endpar}$$

where $u_1, \dots, u_{m_i}$ are update commands.

Remarque 6. *$\Delta(\Pi_A, X) = \Delta(A, X) = \tau_A(X) \ominus X$, so $\Delta(\Pi_A, X)$ is consistent without trivial updates.*

The proof that the set of sequential algorithms is identical to the set of ASMs uses mainly the fact that every ASM has a finite exploration witness. Reciprocally, for every sequential algorithm we can define an ASM with the same transition function:

Theorem 6 (Gurevich, 2000)**.**

$$\texttt{Algo} = \texttt{ASM}$$

So, Gurevich proved that his axiomatic presentation for sequential algorithms defines the same objects than his operational presentation of ASMs.

Remarque 7. *According to this theorem, every ASM is a sequential algorithm and every sequential algorithm can be simulated by an ASM in normal form. So, for every ASM there exists an equivalent ASM in normal form.*

2.2 Imperative programming

We know that an imperative language such as Albert Meyer and Dennis Ritchie's `Loop` defined in [18] can compute any primitive recursive function, but cannot compute some "better" algorithms (see [6] for the *min* and [19] for the *gcd*). This `Loop` language has been extended in [with an `exit` command to obtain every Arithmetical Primitive Recursive Algorithm[13].

In [16] we generalized this result, by proving that `LoopC` characterized algorithms with primitive recursive time and data structures. We use this language because it is minimal. The programs are only sequences of updates, `if` or `loop` commands. Notice that the `loop` commands can be broken if an exception is reached, like in [1]. Moreover, in the following section we will prove that a sublanguage `PLoopC` has polynomial time.

The difference with common models of computation is that the data structures are not fixed. As is the case for the ASMs, the equality and the booleans are needed, and the unary integers are necessary for the `loop` commands, but the other data structures are seen as oracular. If they can be implemented in a sequential algorithm then they are implemented using the same language, universe and interpretation in this programming language. So, the fair simulation between $\texttt{ASM}_{\mathcal{P}}$ and `PLoopC` is proven for control structures, up to data structures.

Definition 6 (Syntax of `LoopC` programs)**.**

$$\begin{aligned} c =_{def}\ & f(t_1, \dots, t_\alpha) := t_0 \\ & \mid \texttt{if } F \texttt{ \{} P_1 \texttt{\} else \{} P_2 \texttt{\}} \\ & \mid \texttt{loop } n \texttt{ except } F \texttt{ \{} P \texttt{\}} \\ P =_{def}\ & \texttt{end} \\ & \mid c; P \end{aligned}$$

where f is a dynamic α-ary function symbol, $t_0, t_1, \dots, t_\alpha$ are closed terms, F is a formula, and n is a variable which is not updated in the

[13] APRA is defined as the set of the sequential algorithms with a primitive recursive time complexity, using only booleans and unary integers as data structures, and using only variables as dynamical symbols.

body of the loop.

Notation 7. As is the case for `ASM` programs, we write only `if` F $\{P\}$ for the command `if` F $\{P\}$ `else` $\{$`end`$\}$. Following Meyer and Ritchie's style [18], we write simply `loop` n $\{P\}$ a command

$$\texttt{loop}\ n\ \texttt{except}\ \mathit{false}\ \{P\}$$

For the sake of clarity, we will omit the `end` inside curly brackets in the rest of the paper.

The composition of commands $c; P$ can be generalized by induction to **composition of programs** $P_1; P_2$ by $\texttt{end}; P_2 =_{def} P_2$ and $(c; P_1); P_2 =_{def} c; (P_1; P_2)$. As seen in example 1 p.8 the operational semantics of this `LoopC` programming language is formalized by a state transition system. A state of the system is a pair $P \star X$ of a `LoopC` program and a structure. Its transitions are determined only by the head command and the current structure:

Definition 7 (Operational Semantics of `LoopC`)**.**

$$f(t_1, \ldots, t_\alpha) := t_0; P \star X \succ P \star X \oplus (f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)$$

$$\texttt{if}\ F\ \{P_1\}\ \texttt{else}\ \{P_2\}; P_3 \star X \succ P_j; P_3 \star X \quad \textit{where } j = \begin{cases} 1 & \textit{if } F \textit{ is true in } X \\ 2 & \textit{else} \end{cases}$$

$$\texttt{loop}\ n\ \texttt{except}\ F\ \{P_1\}; P_2 \star X \succ Q; P_2 \star X \oplus (i, a)$$

$$\textit{where } Q = \begin{cases} P_1; \texttt{loop}\ n\ \texttt{except}\ F\ \{P_1\} & \textit{if } i < n \textit{ and } \overline{F}^X = \textit{false} \\ \texttt{end} & \textit{else} \end{cases}$$

$$\textit{and } a = \begin{cases} \overline{i}^X + 1 & \textit{if } i < n \textit{ and } F \textit{ is false in } X \\ 0 & \textit{else} \end{cases}$$

i is a dynamical symbol initialized to 0 *in the initial states and which does not appear in the program. Each loop has a different counter i.*

The successors are unique, so this transition system is deterministic. We denote by $\succ_t$ a succession of t transition steps.

Only the states $\texttt{end} \star X$ have no successor, so they are the terminating states.

Notation 8. P **terminates** on X if there exists t and X' such that:

$$P \star X \succ_t \texttt{end} \star X'$$

Because the transition system is deterministic, t and X' are unique. So X' is denoted $P(X)$ and t is denoted $\mathit{time}(P, X)$. A program is terminal if it terminates for every initial state.

Example 9. This program computes the minimum of two integers m and n in $O(min(m,n))$ steps, and stores the result in the output variable r:

$$P_{min} =_{def} r := 0; \texttt{loop } n \texttt{ except } r = m \ \{r := r + 1;\}; \texttt{end}$$

The execution of this program for $m = 2$ and $n = 3$ on a structure X is:

$$\begin{aligned}
& r := 0; \texttt{loop } 3 \texttt{ except } r = 2 \ \{r := r + 1;\}; \star\, X \oplus (i,0) \\
\succ\ & \texttt{loop } 3 \texttt{ except } r = 2 \ \{r := r + 1;\}; \star\, X \oplus \{(i,0),(r,0)\} \\
\succ\ & r := r + 1; \texttt{loop } 3 \texttt{ except } r = 2 \ \{r := r + 1;\}; \star\, X \oplus \{(i,1),(r,0)\} \\
\succ\ & \texttt{loop } 3 \texttt{ except } r = 2 \ \{r := r + 1;\}; \star\, X \oplus \{(i,1),(r,1)\} \\
\succ\ & r := r + 1; \texttt{loop } 3 \texttt{ except } r = 2 \ \{r := r + 1;\}; \star\, X \oplus \{(i,2),(r,1)\} \\
\succ\ & \texttt{loop } 3 \texttt{ except } r = 2 \ \{r := r + 1;\}; \star\, X \oplus \{(i,2),(r,2)\} \\
\succ\ & \texttt{end} \star\, X \oplus \{(i,0),(r,2)\}
\end{aligned}$$

So $time(P_{min}, X) = 2 + 2 \times min(\overline{m}^X, \overline{n}^X) = O(min(\overline{m}^X, \overline{n}^X))$

The composition of programs behaves as intended. It is proved in [16] by using only the determinism and the transitivity of the transition system.

Proposition 1 (Composition of Programs)**.**

$P_1; P_2$ terminates on X if and only if P_1 terminates on X and P_2 terminates on $P_1(X)$, such that:

1. $P_1; P_2(X) = P_2(P_1(X))$
2. $time(P_1; P_2, X) = time(P_1, X) + time(P_2, P_1(X))$

As a consequence, we can prove by induction that every `LoopC` program is terminal.

Because the transition system is deterministic, there exists a unique P' and X' such that $P \star X \succ_t P' \star X'$. Let $\tau^t_X(P)$ be that P' and $\tau^t_P(X)$ be that X':

$$P \star X \succ_t \tau^t_X(P) \star \tau^t_P(X)$$

Remarque 8. *τ^t_P is not a transition function in the sense of the first postulate 1, because $\tau^t_P(X) \neq \tau_P \circ \cdots \circ \tau_P(X)$.*

Definition 8. *The set of updates made by P on X is:*

$$\Delta(P, X) =_{def} \bigcup_{0 \leq t < time(P,X)} \tau^{t+1}_P(X) \ominus \tau^t_P(X)$$

Remarque 9. *In our transition system, one update at most can be done per step, so $\tau^{t+1}_P(X) \ominus \tau^t_P(X)$ is empty or is a singleton. Therefore, the cardinal of $\Delta(P, X)$ is bounded by $time(P, X)$.*

In imperative programming languages, an **overwrite** occurs when a variable is updated to a value, then is updated to another value later

in the execution. In our framework, this means that there exists in $\Delta(P, X)$ two updates $(f, \vec{a}, b)$ and $(f, \vec{a}, b')$ with $b \neq b'$, which makes $\Delta(P, X)$ inconsistent. So, we say that P is without overwrite on X if $\Delta(P, X)$ is consistent.

Lemma 1 (Updates of a Non-Overwriting Program)**.**
If P is without overwrite on X then $\Delta(P, X) = P(X) \ominus X$.

Proof. The proof is admitted in this paper, but is detailed in [16]. ⊣

2.3 Polynomial Time

We proved in [16] that `LoopC` is algorithmically complete for $\mathcal{PR}$-time algorithms (restricted to $\mathcal{PR}$-space data structures). The purpose of this subsection is to syntactically restrict this language in order to obtain a $\mathcal{P}$-time language `PLoopC`.

FIGURE 1 A Program for the Exponential Function

```
r := 0
r := r + 1
loop n
    x := r
    loop x
        r := r + 1
```

The program P_{pow} in figure 1 is a `Loop` program (see [18]) because it uses only variables, zero, successor, and loops. The command $x := r; \texttt{loop}\ x\ \{r := r + 1;\}$ computes $r := 2r$, so $\overline{r}^{P_{pow}(X)} = 2^{\overline{n}^X}$. Variables of a `Loop` program can only be increased by substitution or successor. So, according to the definition 1.1:

$$|P_{pow}(X)| \leq |X| + time(P_{pow}, X)$$

Therefore, P_{pow} is (at least) exponential in time.

In order to obtain a $\mathcal{P}$-time language, Neergaard used in [21] the Bellantoni and Cook's approach (see [2]) separating **safe and normal variables**. Safe variables can be updated, and normal variables are the bounds n in the `loop` n commands:

Definition 9 (Bounds of Loops)**.**

$$\begin{aligned} Bound(\texttt{end}) &=_{def} \{\} \\ Bound(c; P) &=_{def} Bound(c) \cup Bound(P) \end{aligned}$$

$$\begin{aligned} Bound(f(t_1, \dots, t_\alpha) := t_0) &=_{def} \{\} \\ Bound(\texttt{if } F \ \{P_1\} \ \texttt{else} \ \{P_2\}) &=_{def} Bound(P_1) \cup Bound(P_2) \\ Bound(\texttt{loop } n \ \texttt{except } F \ \{P\}) &=_{def} \{n\} \cup Bound(P) \end{aligned}$$

The programming language PLoop of Neergaard contains only programs such that for every loop n $\{P\}$ command occurring in the program, $\{n\} \cup Bound(P) \subseteq Stat(P)$. In other words, normal variables and safe variables are distinct sets in loops.

As suggested by the name, Neergaard proved that programs in PLoop are in polynomial time (and space[14]). In order to do so, he used lists as data structures, such that $|(d, e)| = max(|d|, |e|) + 1$. This result can be generalized to "translation functions" (in a geometric sense), those satisfying that:

$$|f(\vec{x})| \leq max|\vec{x}| + c_f$$

Theorem 10 (A $\mathcal{P}$-time Language)**.**

If the operations are translations, then PLoop *is $\mathcal{P}$-time and $\mathcal{P}$-space.*

But the set of translation functions is very restrictive. As an example, the program in figure 2 satisfies Neergaard's condition, but is not $\mathcal{P}$-time because $x \mapsto 2x$ is not a translation.

FIGURE 2 A PLoop Program Using more than Translation Functions

```
r := 1
loop n
    r := 2r
loop r
```

As an example, for unary integers the successor is a translation but not the addition, and for binary integers the addition is a translation but not the multiplication. In order to obtain more algorithms, we need to have larger data structures, so we will have to restrict even more our programming language.

The program in figure 2 illustrates that an exponential space in r can be converted into an exponential time using loop r and the

[14] A program P is $\mathcal{C}$-space if there exists $\varphi_P \in \mathcal{C}$ such that, for every initial state X, $|P(X)| \leq \varphi_P(|X|)$.

composition. This is the reason why the space must be $\mathcal{P}$-time too in Neergaard's theorem 10.

In order to obtain a $\mathcal{P}$-time programming language using every possible data structure, we will simply remove the connection between space and time. So, the bounds will remain static in the whole program, and not only in loops.

Definition 10 ($\mathcal{P}$-time Programming Language)**.**
Let `PLoopC` *be the set of* `LoopC` *programs* P *satisfying that:*

$$Bound(P) \subseteq Stat(P)$$

Remarque 10. *This language is not closed by composition. As an example, the programs* $n := pow(n)$; `end` *and* `loop` n {}; `end` *are in* `PLoopC`*, not* $n := pow(n)$; `loop` n {}; `end`*.*

But this language will be useful anyway. We prove at proposition 2 that programs in `PLoopC` are $\mathcal{P}$-time, where the degree of the complexity is the depth of the program:

Definition 11 (Depth of a Program)**.**

$$\begin{aligned} depth(\texttt{end}) &=_{def} 0 \\ depth(c; P) &=_{def} max(depth(c), depth(P)) \end{aligned}$$

$$\begin{aligned} depth(f(t_1, \ldots, t_\alpha) := t_0) &=_{def} 0 \\ depth(\texttt{if } F \ \{P_1\} \ \texttt{else } \{P_2\}) &=_{def} max(depth(P_1), depth(P_2)) \\ depth(\texttt{loop } n \texttt{ except } F \ \{P_1\}) &=_{def} \delta + depth(P_1) \end{aligned}$$

$$where\ \delta = \begin{cases} 1 \ if\ n \in Dyn(P) \sqcup Init(P) \\ 0 \ if\ n \in Cons(P) \sqcup Oper(P) \end{cases}$$

Both cases in the previous definition may be surprising, because the depth is distinguished from the nesting. Indeed, if the bound of a loop is an uniform symbol, we assume that the loop is not a "true" loop, but only a syntactical convention to avoid code duplication, as illustrated at figure 3.

FIGURE 3 Different nesting, same depth.

$r := 0$	$r := 0$
`loop` 3	$r := r + 1$
$\quad r := r + 1$	$r := r + 1$
	$r := r + 1$

Remind that for every program P in `PLoopC`, $Bound(P) \subseteq Stat(P)$. So, we use only the symbols in $Init(P)$ for the depth. Moreover, if P_1

is a subprogram of P then $Init(P_1) \subseteq Init(P)$. So, in the following proposition and its proof, for the sake of simplicity we use the notation $|X|_{Init}$ for the size in X of the initial symbols of the program and its subprograms:

$$|X|_{Init} =_{def} \max_{f \in Init(P)} \{|f|_X\}$$

Proposition 2 (Polynomial Time)**.**

For every program P in `PLoopC` *there exists $\varphi_P \in \mathcal{P}$ such that for every X:*

1. $time(P, X) \leq \varphi_P(|X|_{Init})$
2. $deg(\varphi_P) = depth(P)$

Remarque 11. *Initial symbols are static, so their interpretations in $P(X)$ are the same as in X. As a consequence, for every program P, $|P(X)|_{Init} = |X|_{Init}$.*

Proof. The proof is made by induction on P, by using:

- $\varphi_{\texttt{end}} = 0$
- $\varphi_{P_1;P_2} = \varphi_{P_1} + \varphi_{P_2}$ (according to proposition 1 p.16)

It remains to prove the proposition for commands alone, using the induction hypothesis. Both cases for updates and conditionals are straightforward:

- $\varphi_{f(t_1,\dots,t_\alpha):=t_0} = 1$
- $\varphi_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}} = 1 + \varphi_{P_1} + \varphi_{P_2}$

So, we focus only on the non-trivial case of loops.

Because a `for` command (where the counter i is read) can be simulated by using a fresh variable, we can assume for the sake of simplicity that counters are not read in the program. So, we can write the execution of the program $P = \texttt{loop } n \texttt{ except } F \ \{P_1\}$ in this way:

$$\begin{array}{lr}
 & \texttt{loop } n \texttt{ except } F \ \{P_1\}; \texttt{end} \star \quad X \oplus (i,0) \\
\succ & P_1; \texttt{loop } n \texttt{ except } F \ \{P_1\}; \texttt{end} \star \quad X \oplus (i,1) \\
\succ_{time(P_1,X)} & \texttt{loop } n \texttt{ except } F \ \{P_1\}; \texttt{end} \star \ P_1(X) \oplus (i,1) \\
\succ & P_1; \texttt{loop } n \texttt{ except } F \ \{P_1\}; \texttt{end} \star \ P_1(X) \oplus (i,2) \\
\vdots & \\
\succ_{time(P_1,P_1^{a-1}(X))} & \texttt{loop } n \texttt{ except } F \ \{P_1\}; \texttt{end} \star \ P_1^a(X) \oplus (i,a) \\
\succ & \texttt{end} \star \ P_1^a(X) \oplus (i,0)
\end{array}$$

where a is the first $0 \leq t \leq \overline{n}^X$ such that F is true in $P_1^t(X)$, or $a = \overline{n}^X$ if F is false in $P_1^t(X)$ for every $0 \leq t \leq \overline{n}^X$.

1. Time of the execution:

$$\begin{aligned} time(P, X) &= \sum_{0 \leq t \leq a-1} (1 + time(P_1, P_1^t(X))) + 1 \\ &\leq 1 + \overline{n}^X + \sum_{0 \leq t \leq \overline{n}^X - 1} time(P_1, P_1^t(X)) \end{aligned}$$

By induction hypothesis, for every X:

$$time(P_1, X) \leq \varphi_{P_1}(|X|_{Init})$$

So, $time(P_1, P_1^t(X)) \leq \varphi_{P_1}(|P_1^t(X)|_{Init}) = \varphi_{P_1}(|X|_{Init})$. Therefore:

$$\begin{aligned} time(P, X) &\leq 1 + \overline{n}^X + \sum_{0 \leq t \leq \overline{n}^X - 1} \varphi_{P_1}(|X|_{Init}) \\ &= 1 + \overline{n}^X \times (1 + \varphi_{P_1}(|X|_{Init})) \quad (1) \\ &\leq 1 + (\overline{n}^X + 1) \times (1 + \varphi_{P_1}(|X|_{Init})) \quad (2) \end{aligned}$$

We use both inequalities for the following cases:

- If $n \in Cons(P) \sqcup Oper(P)$, then there exists an integer c_n such that $\overline{n}^X = c_n$.
  So, $\varphi_P = 1 + c_n \times (1 + \varphi_{P_1})$ (1).
- If $n \in Init(P)$, then $\overline{n}^X + 1 = |\overline{n}^X| \leq |X|_{Init}$.
  So, $\varphi_P(x) = 1 + x \times (1 + \varphi_{P_1}(x))$ (2).

2. The degree of the complexity depends on the case:
   - If $n \in Cons(P) \sqcup Oper(P)$, then:

$$\begin{aligned} deg(\varphi_P) &= deg(1 + c_n \times (1 + \varphi_{P_1})) \\ &= deg(\varphi_{P_1}) \\ &= depth(P_1) \\ &= depth(P) \end{aligned}$$

   - If $n \in Init(P)$, then:

$$\begin{aligned} deg(\varphi_P) &= deg(1 + id \times (1 + \varphi_{P_1})) \\ &= 1 + deg(\varphi_{P_1}) \\ &= 1 + depth(P_1) \\ &= depth(P) \end{aligned}$$

$\dashv$

The coefficients of φ_P are positive, so if $0 \leq m \leq n$ then $\varphi_P(m) \leq \varphi_P(n)$. More specifically, since $|X|_{Init} \leq |X|$, we have $\varphi_P(|X|_{Init}) \leq \varphi_P(|X|)$. As a consequence, we have $time(P, X) \leq \varphi_P(|X|)$. Therefore, the time complexity of P is at most a polynomial of degree $depth(P)$.

3 ASM Simulates LoopC

3.1 Graphs of Execution

The intuitive idea for translating `LoopC` programs into `ASM` programs is to translate separately every command, and to add a variable (for example, the number of the line in the program) to keep track of the current command[15].

Example 11. The imperative program P_{min} of the example 9 p.16:

$$\begin{array}{ll} 0: & r := 0 \\ 1: & \texttt{loop } n \texttt{ except } r = m \\ 2: & \quad r := r + 1 \end{array}$$

could be translated into the following ASM program:

FIGURE 4 Translation of P_{min}

```
par  if line = 0 then
        par r := 0 ‖ line := 1 endpar
     endif
   ‖ if line = 1 then
        if (i ≠ n ∧ r ≠ m) then
           par i := i + 1 ‖ line := 2 endpar
        else
           par i := 0 ‖ line := 3 endpar
        endif
     endif
   ‖ if line = 2 then
        par r := r + 1 ‖ line := 1 endpar
     endif
endpar
```

Remarque 12. *The number of a line is between* 0 *and* $length(P)$*. So, a finite number of booleans* $b_0, b_1, \ldots, b_{length(P)}$ *can be used*[16] *instead of an integer line.*

This approach has been suggested in [14], and is fitted for a line-based programming language (for example with `goto` instructions) but not the structured language `LoopC`. Indeed, the positions in the program can distinguish two commands even if they are identical for the operational semantics of `LoopC`:

[15] Programs of this form are called control state ASMs (see [5]).

[16] Remember that booleans must be in the data structure, but integers may not.

Example 12. (Labelled `LoopC`)

To make an easy example, let's compare the two updates $x := x + 1$ in the program of figure 5.

$$\texttt{loop}\ m\ \{x := x + 1;\ \texttt{loop}\ n\ \{x := x + 1;\};\};\texttt{end}$$

Because their positions are not the same in the program they have different numbers of line. So, we label them with $x := x + 1$ ⓐ and $x := x + 1$ ⓑ to distinguish each one from the other.

FIGURE 5 Loops with Labelling (1)

P_1	$\succ P_2$	$\succ P_3$
`loop` m	$x := x + 1$ ⓐ	`loop` n
$x := x + 1$ ⓐ	`loop` n	$x := x + 1$ ⓑ
`loop` n	$x := x + 1$ ⓑ	`loop` m
$x := x + 1$ ⓑ	`loop` m	$x := x + 1$ ⓐ
	$x := x + 1$ ⓐ	`loop` n
	`loop` n	$x := x + 1$ ⓑ
	$x := x + 1$ ⓑ	

FIGURE 6 Loops with Labelling (2)

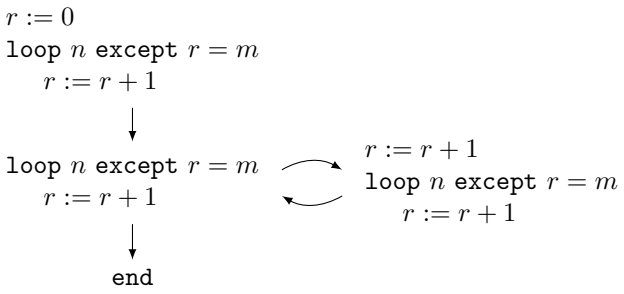
P_3	$\succ P_4$	$\succ \ldots$
`loop` n	$x := x + 1$ ⓑ	
$x := x + 1$ ⓑ	`loop` n	
`loop` m	$x := x + 1$ ⓑ	
$x := x + 1$ ⓐ	`loop` m	
`loop` n	$x := x + 1$ ⓐ	
$x := x + 1$ ⓑ	`loop` n	
	$x := x + 1$ ⓑ	

We can replace the program P_2 of figure 5 by the program P_4 of figure 6 without changing anything except the labels. The commands $x := x + 1$ ⓐ and $x := x + 1$ ⓑ are the same for the operational semantics, so we should find another way to keep track of the current command.

We will not use booleans $b_0, b_1, \ldots, b_{length(P)}$ indexed by the lines of

the program, but booleans indexed by the possible states of the program during the execution. The possible executions of a program will be represented by a graph where the edges are the possible transitions, and the vertices are the possible programs:

Example 13. (Graph of Execution of P_{min})



In the following only the vertices of the graph are needed, so the graph of execution of P_{min} will be denoted by the set of possible programs:

$$\begin{aligned}\mathcal{G}(P_{min}) = \{ \quad & r := 0; \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r + 1;\}; \texttt{end},\\ & \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r + 1;\}; \texttt{end},\\ & r := r + 1; \texttt{loop}\ n\ \texttt{except}\ r = m\ \{r := r + 1;\}; \texttt{end},\\ & \texttt{end}\\ \}\end{aligned}$$

Notation 14. In order to define graphs of execution we need to introduce the notation:

$$\mathcal{G}; P =_{def} \{P_j; P \mid P_j \in \mathcal{G}\}$$

where $\mathcal{G}$ is a set of imperative programs and P is an imperative program.

Let P be an imperative program. $\mathcal{G}(P)$ is the set of every possible $\tau^t_X(P)$ programs, which does not depend on an initial state X:

Definition 12. *(Graph of Execution)*

$$\begin{aligned}\mathcal{G}(\texttt{end}) &=_{def} \{\texttt{end}\}\\ \mathcal{G}(c; P) &=_{def} \mathcal{G}(c); P \cup \mathcal{G}(P)\end{aligned}$$

$$\begin{aligned}\mathcal{G}(f(t_1, \ldots, t_\alpha) := t_0) &=_{def} \{f(t_1, \ldots, t_\alpha) := t_0; \texttt{end}\}\\ \mathcal{G}(\texttt{if}\ F\ \{P_1\}\ \texttt{else}\ \{P_2\}) &=_{def} \{\texttt{if}\ F\ \{P_1\}\ \texttt{else}\ \{P_2\}; \texttt{end}\\ &\qquad \cup \mathcal{G}(P_1) \cup \mathcal{G}(P_2)\\ \mathcal{G}(\texttt{loop}\ n\ \texttt{except}\ F\ \{P\}) &=_{def} \mathcal{G}(P); \texttt{loop}\ n\ \texttt{except}\ F\ \{P\}; \texttt{end}\end{aligned}$$

As intended, we can prove (see [16]) that $card(\mathcal{G}(P)) \leq length(P)+1$.

So, only a finite number of guards depending only on P are necessary. Notice that for some programs (like P_{min} in example 13 p.24) which do not follow example 12 p.23, $card(\mathcal{G}(P)) = length(P)+1$ can be reached, so the bound is optimal.

Again, to focus on the simulation, we admit in this paper the proof (see [16]) stating that a graph of execution is closed for the operational semantics of the imperative programs:

Proposition 3. *(Operational Closure of Graph of Execution)*

- *If* $f(t_1, \ldots, t_\alpha) := t_0; Q \in \mathcal{G}(P)$
  then $Q \in \mathcal{G}(P)$
- *If* `if` F $\{P_1\}$ `else` $\{P_2\}; Q \in \mathcal{G}(P)$
  then $P_1; Q$ *and* $P_2; Q \in \mathcal{G}(P)$
- *If* `loop` n `except` F $\{P_1\}; Q \in \mathcal{G}(P)$
  then P_1; `loop` n `except` F $\{P_1\}; Q$ *and* $Q \in \mathcal{G}(P)$

3.2 Translation of an Imperative Program

Notation 15. The fresh boolean variables will be denoted b_{P_j}, where $P_j \in \mathcal{G}(P)$. One and only one b_{P_j} will be true for each step of an execution, so in the following we will write $X[b_{P_j}]$ if b_{P_j} is true and the other booleans b_{P_k} are false, where X denotes a $\mathcal{L}(P)$-structure. Notice that $X[b_{P_j}]|_{\mathcal{L}(P)} = X$.

Proposition 3 ensures that the following translation is well-defined:

Definition 13. *(Translation of imperative programs into ASM)*

$$\Pi_P =_{def} \mathtt{par} \underset{P_j \in \mathcal{G}(P)}{\quad} \mathtt{if}\ b_{P_j}\ \mathtt{then}\ P_j^{tr}\ \mathtt{endpar}$$

where P_j^{tr} *is defined at the figure 7 p.26.*

Notice that for every $P_j \in \mathcal{G}(P)$, $\Delta(\Pi_P, X[b_{P_j}]) = \Delta(P_j^{tr}, X[b_{P_j}])$. We use this fact in [16] to prove by exhaustion on $\tau_X^t(P)$ that the translation of the imperative program P behaves as intended:

Proposition 4. *(Step-by-Step Simulation)*

For every $t < time(P, X), \tau_{\Pi_P}(\tau_P^t(X)[b_{\tau_X^t(P)}]) = \tau_P^{t+1}(X)[b_{\tau_X^{t+1}(P)}]$

Theorem 16. `ASM` *fairly simulates* `LoopC`.

Proof. We prove the three conditions of the fair simulation defined p.11:

1. $\mathcal{L}(\Pi_P) = \mathcal{L}(P) \cup \{b_{P_j} \mid P_j \in \mathcal{G}(P)\}$
   where $card(\{b_{P_j} \mid P_j \in \mathcal{G}(P)\}) \leq length(P) + 1$.
2. Using proposition 4, we can prove by induction on $t \leq time(P, X)$ that $\tau_{\Pi_P}^t(X[b_P]) = \tau_P^t(X)[b_{\tau_X^t(P)}]$.

FIGURE 7 Translation of an Imperative Program

$$(\texttt{end})^{tr} =_{def} \texttt{ par endpar}$$

$$\begin{aligned}
&(f(t_1,\dots,t_\alpha) := t_0; Q)^{tr}\\
&\qquad \texttt{par } b_{f(t_1,\dots,t_\alpha):=t_0;Q} := \mathit{false}\\
&\qquad\quad \| \ f(t_1,\dots,t_\alpha) := t_0\\
&=_{def} \qquad\quad \| \ b_Q := \mathit{true}\\
&\qquad \texttt{endpar}
\end{aligned}$$

$$\begin{aligned}
&(\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}; Q)^{tr}\\
&\qquad \texttt{par } b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\};Q} := \mathit{false}\\
&\qquad\quad \| \ \texttt{if } F \texttt{ then}\\
&\qquad\qquad\quad b_{P_1;Q} := \mathit{true}\\
&=_{def} \qquad\quad \texttt{else}\\
&\qquad\qquad\quad b_{P_2;Q} := \mathit{true}\\
&\qquad\quad \texttt{endif}\\
&\qquad \texttt{endpar}
\end{aligned}$$

$$\begin{aligned}
&(\texttt{loop } n \texttt{ except } F \ \{\texttt{end}\}; Q)^{tr}\\
&\qquad \texttt{if } (i \neq n \wedge \neg F) \texttt{ then}\\
&\qquad\quad i := i+1\\
&\qquad \texttt{else}\\
&\qquad\qquad \texttt{par } b_{\texttt{loop } n \texttt{ except } F \ \{\texttt{end}\};Q} := \mathit{false}\\
&=_{def} \qquad\quad \| \ i := 0\\
&\qquad\qquad\quad \| \ b_Q := \mathit{true}\\
&\qquad\quad \texttt{endpar}\\
&\qquad \texttt{endif}
\end{aligned}$$

$$\begin{aligned}
&(\texttt{loop } n \texttt{ except } F \ \{c; P_1\}; Q)^{tr}\\
&\qquad \texttt{par } b_{\texttt{loop } n \texttt{ except } F \ \{c;P_1\};Q} := \mathit{false}\\
&\qquad\quad \| \ \texttt{if } (i \neq n \wedge \neg F) \texttt{ then}\\
&\qquad\qquad\quad \texttt{par } i := i+1\\
&\qquad\qquad\qquad\quad \| \ b_{c;P_1;\texttt{loop } n \texttt{ except } F \ \{c;P_1\};Q} := \mathit{true}\\
&\qquad\qquad\quad \texttt{endpar}\\
&=_{def} \qquad\quad \texttt{else}\\
&\qquad\qquad\quad \texttt{par } i := 0\\
&\qquad\qquad\qquad\quad \| \ b_Q := \mathit{true}\\
&\qquad\qquad\quad \texttt{endpar}\\
&\qquad\quad \texttt{endif}\\
&\qquad \texttt{endpar}
\end{aligned}$$

So, $\tau^t_{\Pi_P}(X[b_P])|_{\mathcal{L}(P)} = \tau^t_P(X)$, and the temporal dilation is $\boxed{d = 1}$.

3. If $t = time(P, X)$ then $\tau^t_X(P) = \texttt{end}$.
   So, $\Delta(\Pi_P, \tau^t_P(X)[b_{\tau^t_X(P)}]) = \varnothing$, and $\tau^{t+1}_{\Pi_P}(X[b_P]) = \tau^t_{\Pi_P}(X[b_P])$.
   Therefore, $time(\Pi_P, X[b_P]) \leq time(P, X)$. (1)
   Let $t < time(P, X)$. According to the operational semantics 7:
   If $\tau^t_X(P) \star \tau^t_P(X) \succ \tau^{t+1}_X(P) \star \tau^{t+1}_P(X)$ then $\tau^t_X(P) \neq \tau^{t+1}_X(P)$ or $\tau^t_X(P) = \texttt{loop}\ n\ \texttt{except}\ F\ \{\}; Q$.
   In the first case, $b_{\tau^t_X(P)} \neq b_{\tau^{t+1}_X(P)}$, and in the second case $\tau^t_P(X) \neq \tau^{t+1}_P(X)$, since $\overline{i}^{\tau^{t+1}_P(X)} = \overline{i}^{\tau^t_P(X)} + 1$.
   In any case, $\tau^{t+1}_P(X)[b_{\tau^{t+1}_X(P)}] \neq \tau^t_P(X)[b_{\tau^t_X(P)}]$.
   So, $\tau^{t+1}_{\Pi_P}(X[b_P]) \neq \tau^t_{\Pi_P}(X[b_P])$.
   Therefore, $time(\Pi_P, X[b_P]) \geq time(P, X)$. (2)
   According to (1) and (2), $time(\Pi_P, X[b_P]) = time(P, X)$, so $\boxed{e = 0}$.

⊣

Therefore, as stated in the conclusion, ASMs in polynomial time can fairly simulate programs of `PLoopC`, since (according to proposition 2 p.20) they are in polynomial time, and `PLoopC` is a sublanguage of `LoopC`.

4 PLoopC Simulates Polynomial-Time ASM

Let Π be an ASM program with a polynomial-time complexity $\varphi_\Pi \in \mathcal{P}$. The purpose of this section is to find a `PLoopC` program P_{step} simulating the same executions as Π. We construct this program P_{step} in three steps:

1. Translate Π into an imperative program P_{step} simulating one step of the ASM.
2. Repeat P_{step} a sufficient number of times, depending on φ_Π, the complexity of Π.
3. Ensure that the final program stops at the same time as the ASM, up to temporal dilation.

4.1 Translation of one Step

Remember that Π contains only updates, `if` and `par` commands. The intuitive solution is to translate the commands directly, without paying attention to the parallelism:

Definition 14 (Syntactical Translation of the ASM programs)**.**

$$(f(t_1,\dots,t_\alpha) := t_0)^{tr} =_{def} f(t_1,\dots,t_\alpha) := t_0;\texttt{end}$$

$$(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})^{tr} =_{def} \texttt{if } F \texttt{ then } \{\Pi_1^{tr}\} \texttt{ else } \{\Pi_2^{tr}\};\texttt{end}$$

$$(\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar})^{tr} =_{def} \Pi_1^{tr};\dots;\Pi_n^{tr}$$

Updates and `if` commands are the same in these two models of computation, but the simultaneous commands of `ASM` must be sequentialized in `LoopC`, so this translation does not respect the semantics of the ASM programs:

Example 17. Let X be a structure such that $\overline{x}^X = 0$ and $\overline{y}^X = 1$, and Π be the program:

$$\Pi = \texttt{par } x := y \| y := x \texttt{ endpar}$$

Since both updates are done simultaneously, the semantics of Π is to exchange the value of x and y. In that case $\Delta(\Pi, X) = \{(x,1),(y,0)\}$, so $\tau_\Pi(X) = X \oplus \{(x,1),(y,0)\}$.

$$\Pi^{tr} = x := y; y := x; \texttt{end}$$

But the semantics of Π^{tr} is to replace the value of x by the value of y and leave y unchanged. In that case, we have the following execution:

$$\begin{aligned} & x := y; y := x; \texttt{end} \star X \\ \succ \quad & y := x; \texttt{end} \star X \oplus \{(x,1)\} \\ \succ \quad & \texttt{end} \star X \oplus \{(x,1),(y,1)\} \end{aligned}$$

So $\tau_\Pi(X) = X \oplus \{(x,1),(y,0)\} \neq X \oplus \{(x,1),(y,1)\} = \Pi^{tr}(X)$.

In order to capture the simultaneous behavior of the ASM program, we need to store the values of the variables read in the imperative program. As an example, if $v_x = x$ and $v_y = y$ in X then:

$$\begin{aligned} & x := v_y; y := v_x; \texttt{end} \star X \\ \succ \quad & y := v_x; \texttt{end} \star X \oplus \{(x,1)\} \\ \succ \quad & \texttt{end} \star X \oplus \{(x,1),(y,0)\} \end{aligned}$$

Indeed, even if x has been updated, its old value is still in v_x.

Definition 15 (Substitution of a Term by a Variable)**.**

$$(f(t_1,\dots,t_\alpha) := t_0)[v/t] =_{def} f(t_1[v/t],\dots,t_\alpha[v/t]) := t_0[v/t]$$

$$(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})[v/t] =_{def} \texttt{if } F[v/t] \texttt{ then } \Pi_1[v/t] \texttt{ else } \Pi_2[v/t] \texttt{ endif}$$

$$(\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar})[v/t] =_{def} \texttt{par } \Pi_1[v/t] \| \dots \| \Pi_n[v/t] \texttt{ endpar}$$

$$\text{where } t_1[v/t_2] =_{def} \begin{cases} v & \text{if } t_1 = t_2 \\ t_1 & \text{else} \end{cases}$$

Remarque 13. *Since the temporary variables are fresh, if t_1 and t_2 are distinct terms then $\Pi[v_{t_1}/t_1][v_{t_2}/t_2] = \Pi[v_{t_2}/t_2][v_{t_1}/t_1]$ As a consequence, since the substitutions can be made in any order, for the terms $t_1, \ldots, t_r$ read by Π (see the definition p.12), the notation $\Pi[\vec{v_t}/\vec{t}]$ is not ambiguous.*

But using $\Pi[\vec{v_t}/\vec{t}]^{tr}$ for P_{step} is not sufficient, because two issues remain:

1. The variables $\vec{v_t}$ must be initialized with the value of the terms $\vec{t}$. Because the fresh variables must have a uniform initialization (see p.10), we have to update the variables $\vec{v_t}$ explicitly at the beginning of the program by using a sequence of updates:
$$v_{t_1} := t_1; \ldots; v_{t_r} := t_r;$$
2. The execution time depends on the current initial state.
This is an issue because, according to our definition of the fair simulation p.11, every step of the ASM Π must be simulated by d steps, where d depends only on Π. In order to obtain a uniform temporal dilation, we will add `skip` commands[17] to the program:
$$\begin{array}{ll} \texttt{skip } 0 & =_{def} \texttt{ end} \\ \texttt{skip } n+1 & =_{def} \texttt{ if } \mathit{true} \ \{\}; \texttt{skip } n \end{array}$$

According to the Gurevich's Theorem, every ASM is equivalent to an ASM in normal form, so we can assume that Π is in normal form (see p.14). Therefore, its translation has the form:

$$\begin{array}{l} \texttt{if } F_1 \texttt{ then } \{\Pi_1^{tr}\}; \\ \texttt{if } F_2 \texttt{ then } \{\Pi_2^{tr}\}; \\ \vdots \\ \texttt{if } F_c \texttt{ then } \{\Pi_c^{tr}\}; \\ \texttt{end} \end{array}$$

Remind that every F is a guard, which means that one and only one F_i is true for the current state X. The block of updates $\Pi_i^{tr} = u_1; \ldots; u_{m_i}; \texttt{end}$ requires m_i steps to be computed by the imperative program, so we add `skip` $m - m_i$ at the end of the block, where m is

[17] It may seem strange in an algorithmic purpose to lose time, but these `skip` commands do not change the asymptotic behavior and are necessary for our strict definition of the fair simulation. It is possible to weaken the definition of the simulation to simulate one step with $\leq d$ steps and not $= d$ steps, but we wanted to prove the result for the strongest definition possible.

defined by:

$$m =_{def} max\{m_i \mid 1 \leq i \leq c\}$$

FIGURE 8 Translation P_{step} of one Step of Π

$P_{step} =_{def}$
$v_{t_1} := t_1;$
$v_{t_2} := t_2;$
$\vdots$
$v_{t_r} := t_r;$
`if` v_{F_1} `then` {
$f^1_1(\vec{v}_{t^1_1}) := v_{t^1_1};$
$f^1_2(\vec{v}_{t^1_2}) := v_{t^1_2};$
$\vdots$
$f^1_{m_1}(\vec{v}_{t^1_{m_1}}) := v_{t^1_{m_1}};$
`skip` $m - m_1;$
};
`if` v_{F_2} `then` {
$f^2_1(\vec{v}_{t^2_1}) := v_{t^2_1};$
$f^2_2(\vec{v}_{t^2_2}) := v_{t^2_2};$
$\vdots$
$f^2_{m_2}(\vec{v}_{t^2_{m_2}}) := v_{t^2_{m_2}};$
`skip` $m - m_2;$
};
$\vdots$
`if` v_{F_c} `then` {
$f^c_1(\vec{v}_{t^c_1}) := v_{t^c_1};$
$f^c_2(\vec{v}_{t^c_2}) := v_{t^c_2};$
$\vdots$
$f^c_{m_c}(\vec{v}_{t^c_{m_c}}) := v_{t^c_{m_c}};$
`skip` $m - m_c;$
};
`end`

We obtain at figure 8 p.30 the translation P_{step} of one step of the ASM program Π. Let X be a state of the ASM with program Π, extended with the variables $\vec{v_t}$. As intended, we prove that P_{step} simulates one step of Π in a constant time t_Π:

Proposition 5 (Semantical Translation of the ASM programs)**.**

There exists t_Π, depending only on Π, such that for every state X of P_{step}:

- $(P_{step}(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$
- $time(P_{step}, X) = t_\Pi$

Proof. The sequence of updates $v_{t_1} := t_1; \ldots; v_{t_r} := t_r;$ requires r steps. Because the variables $\vec{v_t}$ are fresh they don't appear in the terms $\vec{t}$. So, in the state Y after these updates, $\overline{v_{t_k}}^Y = \overline{t_k}^X$. Moreover, in the rest of the program the variables $\vec{v_t}$ are not updated, so for every following state Y, $\overline{v_{t_k}}^Y = \overline{t_k}^X$.

In particular, for every $1 \leq j \leq c$, $\overline{v_{F_j}}^Y = \overline{F_j}^X$. Since these conditionals are guards, one and only one is $true$ in X. Let F_i be this formula. Therefore, in every following state Y, $\overline{v_{F_i}}^Y = true$, and for every $j \neq i$, $\overline{v_{F_j}}^Y = false$.

$i-1$ steps are required to erase the conditionals before F_i, one step is required to enter the block of F_i, and after the commands in that block $c-i$ steps are required to erase the conditionals after F_i. So, $(i-1)+1+(c-i) = c$ steps are required for the conditionals.

Since for every following state Y, $\overline{v_{t_k}}^Y = \overline{t_k}^X$, the set of updates done in the block of F_i is $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$. These updates require m_i steps, then the `skip` command requires $m-m_i$ steps. So the commands in the block require $m_i + (m-m_i) = m$ steps, and the execution time depends only on Π:

$$time(P_{step}, X) = r + c + m = t_\Pi$$

The updates done by P_{step} are the initial updates and the updates done in the block of F_i:

$$\Delta(P_{step}, X) = \{(v_{t_1}, \overline{t_1}^X), \ldots, (v_{t_r}, \overline{t_r}^X)\} \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$$

The fresh variables are updated only once, and since Π is in normal form, $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$ is consistent. So, P_{step} is without overwrite on X, and according to proposition 1 p.17:

$$\Delta(P_{step}, X) = P_{step}(X) \ominus X$$

So $(P_{step}(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$. ⊣

More generally, we can use this result to prove by induction on t that:

Corollaire 14. $P^t_{step}(X)|_{\mathcal{L}(\Pi)} = \tau^t_\Pi(X|_{\mathcal{L}(\Pi)})$

4.2 Translation of the Complexity

P_{step} simulates in constant time one step of the ASM program Π, so we want to repeat it a sufficient number of times in order to simulate every execution of the ASM. In this paper we focus on the polynomial time algorithms, so we assume that there exists a polynomial function φ_Π such that for every initial state X:

$$time(\Pi, X) \leq \varphi_\Pi(|X|)$$

Since φ_Π is a polynomial function, there exists $a_0, \dots, a_{deg(\varphi_\Pi)} \in \mathbb{Z}$ such that:

$$\varphi_\Pi(|X|) = \sum_{0 \leq n \leq deg(\varphi_\Pi)} a_n |X|^n \leq \left(\sum_{0 \leq n \leq deg(\varphi_\Pi)} max(0, a_n) \right) |X|^{deg(\varphi_\Pi)}$$

Therefore, there exists $c \in \mathbb{N}$ depending only of φ_Π, such that:

$$time(\Pi, X) \leq c \times |X|^{deg(\varphi_\Pi)}$$

We assume that the program has access to the size of its inputs, so it has access to $|X|$, which is the maximum (or the sum) of these values. Therefore, the following program has an execution time greater than Π on X, where c and *size* are fresh variables initialized respectively with $\sum_{0 \leq n \leq deg(\varphi_\Pi)} max(0, a_n)$ and $|X|$:

```
loop c
  loop size
    ⋱  deg(φ_Π) times
      loop size
```

Notice that according to the definition 11 p.19, the depth of this program is $deg(\varphi_\Pi)$. The intuitive program repeating P_{step} is:

```
loop c
  loop size
    ⋱
      loop size
        P_step
```

In that case, between two executions of P_{step} the number of steps depends on the actual depth in the program, so the simulation will not have a constant temporal dilation. We want the program to execute one step of a loop then to execute P_{step} , then to execute another step of a loop and so on... So, we need to duplicate[18] P_{step} before each body of a

[18] Like in [1], with the difference that we choose to have every execution of P_{step}

loop (when the execution enters a loop) and after each `loop` command (when the execution erases a loop):

$$
\begin{array}{l}
\texttt{loop}\ c \\
\quad P_{step} \\
\quad \texttt{loop}\ size \\
\qquad P_{step} \\
\qquad \ddots \\
\qquad\quad \texttt{loop}\ size \\
\qquad\qquad P_{step} \\
\qquad\quad P_{step} \\
\qquad \ddots \\
\quad P_{step} \\
P_{step}
\end{array}
$$

As a consequence, our candidate is

$$\texttt{loop}\ c\ \{P_{step}; \texttt{loop}^{deg(\varphi_\Pi)}\ size\ \{P_{step}\}\}; P_{step}$$

where $\texttt{loop}^i\ n\ \{P\}$ is defined by induction:

$$
\begin{aligned}
\texttt{loop}^0\ n\ \{P\} &=_{def} \texttt{end} \\
\texttt{loop}^{i+1}\ n\ \{P\} &=_{def} \texttt{loop}\ n\ \{P; \texttt{loop}^i\ n\ \{P\}\}; P
\end{aligned}
$$

The temporal dilation is $d = t_\Pi + 1$, since the program alternates between `loop` commands and executions of P_{step}. But we can't ensure that the program stops at the same time as Π, so we need to detect the end of the execution.

For any initial state of the program P_{step}, the fresh variables $\vec{v_t}$ store the value of the interpretation of the terms $\vec{t}$, then the terms $\vec{t}$ are updated. This means that at the end of P_{step} the variables $\vec{v_t}$ have the old values of the terms. In particular, if the initial state is $P^t_{step}(X)$, after one execution of P_{step} we have:

$$\overline{v_{t_k}}^{P^{t+1}_{step}(X)} = \overline{t_k}^{P^t_{step}(X)}$$

Π terminates when no more updates are done. In that case, the old values of the terms read by Π are the same as the new values. Therefore, since the old values are stored in the variables $\vec{v_t}$, every v_{t_k} is equal to t_k in the terminating state:

$$F_\Pi =_{def} \bigwedge_{t \in Read(\Pi)} v_t = t$$

---

after every command of the program, not before. This will make sense when we will add one occurrence of P_{step} before the program in order to initialize the μ-formula F_Π.

We call it the "μ-formula" because it is similar to the minimization operator μ from recursive functions (see [7]):

Lemme 15 (The μ-formula)**.**

$$time(\Pi, X|_{\mathcal{L}(\Pi)}) = min\{t \in \mathbb{N} \mid \overline{F_\Pi}^{P_{step}^{t+1}(X)} = true\}$$

Proof. $time(\Pi, X|_{\mathcal{L}(\Pi)}) = min\{t \in \mathbb{N} \mid \tau_\Pi^t(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{t+1}(X|_{\mathcal{L}(\Pi)})\}$, so all that is left to prove (see [16]) is that $\tau_\Pi^t(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{t+1}(X|_{\mathcal{L}(\Pi)})$ if and only if $\overline{F_\Pi}^{P_{step}^{t+1}(X)}$ is true, by using the remark p.13 on $Read(\Pi)$. ⊣

So, the current candidate to simulate the ASM program Π is the program:

```
loop c except F_Π
  if ¬F_Π {P_step}
  loop size except F_Π
    if ¬F_Π {P_step}
    ⋱
      loop size except F_Π
        if ¬F_Π {P_step}
      if ¬F_Π {P_step}
    ⋱
  if ¬F_Π {P_step}
if ¬F_Π {P_step}
```

The temporal dilation becomes $d = t_\Pi + 2$, since entering the conditionals costs one more step. But two issues remain:

1. The variables $\vec{v_t}$ must be properly initialized to obtain a correct value for the μ-formula F_Π. We do that simply by adding an occurrence of P_{step} at the beginning of the program. F_Π becomes true after $time(\Pi, X|_{\mathcal{L}(\Pi)}) + 1$ steps, so we execute the program P_{step} one more time after the end of Π. This is not an issue because the execution time of P_{step} is t_Π, as required by the third condition of the fair simulation.
2. The simulation is correct until F_Π becomes true, and after that the remaining steps consist to erase the last `loop` commands. But their number depends on the current depth, determined by the initial state. This number is bounded by $deg(\varphi_\Pi) + 1$, so the current ending time can be bounded too. In fact, for every remaining `loop` commands two steps are done: erase the loop then erase the following `if` $\neg F_\Pi \{P_{step}\}$. Therefore, the ending time is

bounded[19] by $max_{end} = 2 \times (deg(\varphi_\Pi) + 1)$.

By using a fresh variable i_{end} which counts the number of steps done after F_Π became true, we can add at the end of the program the program $\texttt{skip}\ i_{end} \to max_{end}$ defined by:

$\texttt{skip}\ i \to 0 \quad =_{def} \texttt{end}$
$\texttt{skip}\ i \to m+1 =_{def} \texttt{if}\ i = m+1\ \{\texttt{end}\}\ \texttt{else}\ \{\texttt{skip}\ i \to m\}; \texttt{end}$

For every state X, we can prove by induction on $0 \leq \overline{i_{end}}^X \leq \overline{max_{end}}^X$ that:

$$time(\texttt{skip}\ i_{end} \to max_{end}, X) = \overline{max_{end}}^X - \overline{i_{end}}^X + 1$$

It remains to set the correct value for i_{end}. This variable is initialized to 0 and for each remaining `loop` commands three steps are done: erase the `loop`, enter the `if` and update i_{end}. So, we replace in our candidate program each $\texttt{if}\ \neg F_\Pi\ \{P_{step}\}$ by $\texttt{if}\ \neg F_\Pi\ \{P_{step}\}\ \texttt{else}\ \{i_{end} := i_{end} + 3; \texttt{end}\}$, and max_{end} becomes $3 \times (deg(\varphi_\Pi) + 1)$.

4.3 The Simulation

For every ASM program Π we obtain at figure 9 its translation P_Π simulating the execution of Π:

Theorem 18. `PLoopC` *fairly simulates* $\texttt{ASM}_\mathcal{P}$.

Proof. We prove the three conditions of the fair simulation defined p.11:

1. $\mathcal{L}(P_\Pi) = \mathcal{L}(\Pi) \sqcup \{v_t \mid t \in Read(\Pi)\} \sqcup \{c, size, i_{end}\}$
   So, there is a finite number of fresh variables, depending only on Π.
2. Until F_Π becomes true the execution alternates between:
   (a) t_Π steps of P_{step}, which simulates one step of Π, according to proposition 5 p.31.
   (b) Then one step to enter the body of a loop, or erase a loop command.
   (c) Then one step to enter the conditional $\texttt{if}\ \neg F_\Pi\ \{P_{step}\}$, then repeat from the beginning.

   So, each step of Π is simulated by exactly $\boxed{d = t_\Pi + 2}$ steps of its translation P_Π.
   Moreover, the execution is sufficiently long. Indeed, if c and *size* are initialized respectively with $\sum_{0 \leq n \leq deg(\varphi_\Pi)} max(0, a_n)$ and $|X_0|$

[19] max_{end} does not depend of the initial state so we can use constructors instead of a variable, and define $\texttt{skip}\ i \to m$ by induction on m. We cannot do that for c because contrary to conditionals, loop commands can only be bounded by a variable, not a term.

FIGURE 9 Translation P_Π of the ASM program Π

P_{step}
`loop` c `except` F_Π
  `if` $\neg F_\Pi$ $\{P_{step}\}$ `else` $\{i_{end} := i_{end} + 3; \texttt{end}\}$
  `loop` *size* `except` F_Π
    `if` $\neg F_\Pi$ $\{P_{step}\}$ `else` $\{i_{end} := i_{end} + 3; \texttt{end}\}$
    $\ddots$ $deg(\varphi_\Pi)$ times
      `loop` *size* `except` F_Π
        `if` $\neg F_\Pi$ $\{P_{step}\}$ `else` $\{i_{end} := i_{end} + 3; \texttt{end}\}$
      `if` $\neg F_\Pi$ $\{P_{step}\}$ `else` $\{i_{end} := i_{end} + 3; \texttt{end}\}$
    $\ddots$ $deg(\varphi_\Pi)$ times
  `if` $\neg F_\Pi$ $\{P_{step}\}$ `else` $\{i_{end} := i_{end} + 3; \texttt{end}\}$
`if` $\neg F_\Pi$ $\{P_{step}\}$ `else` $\{i_{end} := i_{end} + 3; \texttt{end}\}$
`skip` $i_{end} \to max_{end}$

in an initial state X_0 then:

$$time(\Pi, X) \leq \overline{c}^{X_0} \times (\overline{size}^{X_0})^{deg(\varphi_\Pi)}$$

Notice that c and *size* are never updated, so this inequality holds for every state of the execution. Moreover, since c does not depend on the chosen initial state, according to definition 11 p.19:

$$depth(P_\Pi) = deg(\varphi_\Pi)$$

3. Therefore, $time(\Pi, X|_{\mathcal{L}(\Pi)})$ repetitions of these steps simulate the ASM program. Then, according to lemma 15 p.34, t_Π more steps for the last iteration of P_{step} make F_Π true[20].
   Then, until `skip` $i_{end} \to max_{end}$ is reached, the execution alternates between:
   (a) One step to erase a loop command.
   (b) Then one step to enter the else part of the conditional `if` $\neg F_\Pi$.
   (c) Then one step $i_{end} := i_{end} + 3$, then repeat from the beginning.

Because the variable i_{end} is initialized to 0, when `skip` $i_{end} \to max_{end}$ is reached at the state X_{final} the value $\overline{i_{end}}^{X_{final}}$ is the number of steps done since F_Π is true. Then $\overline{max_{end}}^{X_{final}} - \overline{i_{end}}^{X_{final}} + 1$ steps

[20]Even if $time(\Pi, X|_{\mathcal{L}(\Pi)}) = 0$, in which case the initial P_{step} executes all these steps.

are done by $\texttt{skip}\ i_{end} \to max_{end}$. Therefore, the ending time is:

$$e = t_\Pi + \overline{i_{end}}^{X_{final}} + \overline{max_{end}}^{X_{final}} - \overline{i_{end}}^{X_{final}} + 1 = t_\Pi + \overline{max_{end}}^{X_{final}} + 1$$

$$\text{So } \boxed{e = t_\Pi + 3 \times (deg(\varphi_\Pi) + 1) + 1}$$

$\dashv$

5 Conclusion and Discussion

We proved p.35 that ASM fairly simulates LoopC. So, because PLoopC is a sublanguage of LoopC with polynomial time (see p.20), $\texttt{ASM}_\mathcal{P}$ fairly simulates PLoopC. Reciprocally, we proved p.25 that PLoopC fairly simulates $\texttt{ASM}_\mathcal{P}$. Therefore, according to the definition p.11 of the algorithmic equivalence, PLoopC characterizes polynomial time algorithms:

Theorem 19. $\texttt{PLoopC} \simeq \texttt{Algo}_\mathcal{P}$.

This result can be seen as an end of the quest for an *algorithmically complete* language for the set of PTIME algorithms.

Moreover, this language does not require constraints on data structures to be PTIME, unlike PLoop (see [21]).

Nevertheless, PLoopC is not very practicable:

- It is not fully compositional. Indeed, in $P_1; P_2$, we must ensure that the inputs of P_2 are not outputs of P_1.
- And moreover, it is difficult to program in this language because the complexity must be anticipated before writing the program.

In order to obtain the better of both worlds, it would be pleasant to construct an intermediary language, between our PLoopC and Neergaard's PLoop.

We do not need every possible first order structures, only common data structures, which are stronger than Neergaard's translations. So, we are looking for a compromise, by using a restriction on data structures, in order to gain more flexibility from a programmer's perspective.

But, the fact remains that other PTIME languages can be compared to PLoopC for the algorithmic completeness.

References

[1] Andary P., Patrou B. and Valarcher P., A theorem of representation for primitive recursive algorithms, *Fundamenta Informaticae*, XX (2010), pp. 118.

[2] Bellantoni S., and Cook S., A new recursion-theoretic characterisation of the polytime functions, *Computational complexity*, vol. 2 (1992), pp. 97110.

[3] Blass A., Dershowitz N., and Gurevich Y., When are two algorithms the same?, *Bull. Symbolic Logic* Volume 15, Issue 2 (2009), 145-168.

[4] Bonfante G., Some programming languages for LOGSPACE and PTIME, *11th International Conference*, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006.

[5] Borger E., Abstract State Machines: A Unifying View of Models of Computation and of System Design Frameworks, *Annals of Pure and Applied Logic* (2005).

[6] Colson L., About primitive recursive algorithms, *Theoretical Computer Science*, 83 (1991) 5769.

[7] Cori R., Lascar D., and Pelletier D., Mathematical Logic: A Course With Exercises: Part I and II, Paris, *Oxford University Press* (2000, 2001).

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

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

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

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

[12] Gurevich Y., Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic* (2000).

[13] Grigorieff S and Valarcher P., Evolving Multialgebras unify all usual models for computation in sequential time, *Symposium on Theoretical Aspects of Computer Science* (2010).

[14] Grigorieff S and Valarcher P., Classes of Algorithms: Formalization and Comparison, *Bulletin of the EATCS* 107 (2012).

[15] Krivine J.-L., A call-by-name lambda-calculus machine, *Higher Order and Symbolic Computation* 20 (2007) 199-207.

[16] Marquer Y., Caractérisation impérative des algorithmes séquentiels en temps quelconque, primitif récursif ou polynomial, *dr-apeiron.net/doku.php/en:research:thesis-defense* (thesis defended in 2015).

[17] Marquer Y., Algorithmic Completeness of Imperative Programming Languages, *dr-apeiron.net/doku.php/en:research:fi-while* (in revision for Fundamenta Informaticae).

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

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

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

[21] Neergaard P. M., Ploop: A Language For Polynomial Time, 2003.

[22] Niggl K.-H., Control structures in programs and computational complexity, *Annals of Pure and Applied Logic*, Volume 133, Issues 13, May 2005, Pages 247–273.

[23] Michel D., and Valarcher P., A total functional programming language that computes APRA, *Studies in Weak Arithmetic*, Stanford, CSLI Lecture Notes, 2009

[24] Vinar T., Biedl T., Buss J., Demaine E. D., Demaine M. L., and Hajiaghayi M., Palindrome recognition using a multidimensional tape, *Theoretical Computer Science* 302 (2003).