Algorithmic Completeness for PTIME Languages

Yoann Marquer, Pierre Valarcher

March 22, 2016

Contents

Abstract

The abstract state machines (of Y. Gurevich) capture all sequential algorithms, so we define the set of polytime algorithms as the set of all programs (from ASM) computed in polynomial time . Then we construct an imperative programming language (P) using bounded loop with escape and running with general data structures, that compute all and only polytime algorithms in lock step (one step in the ASM is simulated by k steps in P).

Context

For a long time there are many tentative to capture the set of functions computable in polynomial time ([BC92, Nig05] and [Bon06]). Of course the main motivation is to find "pleasant" syntactical (and semantical) languages to capture this set. Sometime, the main motivation is to capture more algorithms than the previous models or to give simpler method to decide a program is computable in PTIME.

Nevertheless, as far as we know, there is no definition of the ultimate set of algorithms that we could reach by some programming languages (limited to PTIME). The study of weakness or *supposed* weakness of programming languages is a whole field of research by studying the algorithmic power of some programming languages (essentially total programming languages because they match well known classes of total functions). All approaches (except in [APV11]) consider the *lacuna* of different languages see [Col89, Col96, CF98] for usual primitive recursion and [Mos03, vdD03] for more sophisticated recursion schemas): for instance, the *minimum* problem is such a well known *lacuna* (it refers to the fact that the usual algorithm that decrements alternatively its two inputs and stops when it achieves to read one of its inputs cannot be simulated by some programming languages; the number of steps to find the minimum is proportional to the smaller inputs but, in many programming languages this *algorithm* is not representable with the same complexity).

In the other side, from now thirty years, there is an interest in defining formally the notion of algorithms [Mos01, Gur00a] and some of these definitions allow to specify some classes of algorithms (in [Gur93], [Gur00a] and [DG08]); an axiomatic definition is mapped to the notion of *abstract state machine* with a *strict lock-step* simulation (see [DDG97] for a definition of simulation and strict lock-step[1]). The abstract state machines are a kind of super Turing machine that works not on simple tape (with finitary alphabets) but on multi–sorted algebras (see the point of view in [GV10]). A program is a finite set of rules that updates terms. It is shown in [Mar14], that the power of expression is essentially in the fact that data structures are modeled within a first order structure rather than in the structure of controls.

In [APV11, MV09] a class of primitive recursive algorithms APRA is defined for the basic data structure of (unary) integer coming from the *abstract state*

[1]↑ The algorithms allow to update simultaneously a finite amount of data.

machine theory. From there, two imperative programming languages and one functional programming languages are proved to be **complete** for this set of algorithms; that is, all algorithms defined in APRA can be written in those languages without lost of time complexity (the simulation is in $O(1)$).

But the class of primitive recursive algorithms is too vast and most of the algorithms is intractable. So there is a challenge to capture the set of algorithms that have time of computation in PTIME. In [Nig05], a LOOP programming language is presented and properties are found to capture PTIME on data structure such as stack, trees or graphs.

Following [APV11], we prove that one can define a programming language that capture exactly the set of of polynomial time algorithms ($\mathtt{Algo}_{\mathcal{P}ol}$). This language allows all first-order structures as data structures and limits only the use of iteration bounds to be inputs.

The paper is organized as follows: the notion of *fairly simulation* and a polytime language ($\mathtt{LoopC}_{\mathtt{stat}}$) are described in the first section. The second section is devoted to show that the familly of languages with bounded loop can be simulated by ASM (in strictly lock-step), so $\mathtt{LoopC}_{\mathtt{stat}}$ is simulated by ASM in polytime. Reciprocally, in the third section, we prove that all polytime ASM are captured by $\mathtt{LoopC}_{\mathtt{stat}}$. We conclude that $\mathtt{LoopC}_{\mathtt{stat}}$ is algorithmically complete for polytime algorithms.

1 Preliminaries

Usually in computability theory, the aim is to prove if some functions can be simulated in a computation model, but this is only an input-output result. In this paper we are interested in a simulation of the whole execution. Our aim is to find a programming language which can simulate every execution of the polytime algorithms.

We consider well known the definition of algorithms considered by the three postulates of Y. Gurevich, as well as the algorithmic model of *Abstract State Machines* (ASM). But reader can have a look at appendice p.**??**. We introduce in this section, the notion of simulation that is necessary to show the completeness of our programming language $\mathtt{LoopC}_{\mathtt{stat}}$ for the set of algorithms that are computable in Ptime. The language is defined in subsection 2.2 as its operational semantics.

1.1 First Order Structures

Definition 1.1. A (first-order) **structure** X is defined by:

1. an infinite[1] set $\mathcal{U}(X)$ called the **universe** of X

[1]↑ Usually the universe is only required to be non-empty, but we will define unary integers p.5 so we need the universe to be at least countable.

2. a finite[1] set of function symbols $\mathcal{L}(X)$ called the **language** of X
3. for each 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 our paper we follow the presentation of Yuri Gurevich[2]. In particular, the constant symbols are considered as function symbols with arity 0. The interpretation of a function is total by default, so we use a special symbol *undef* to define partial functions:

1. $Dom(f, X) =_{def} \{(a_1, \ldots, a_\alpha) \in \mathcal{U}(X)^\alpha \mid \overline{f}^X(a_1, \ldots, a_\alpha) \neq \overline{undef}^X\}$
2. $Im(f, X) =_{def} \{\overline{f}^X(a_1, \ldots, a_\alpha) \in \mathcal{U}(X) \mid (a_1, \ldots, a_\alpha) \in Dom(f, X)\}$

The **booleans** $\mathbb{B}$ are defined as usual. The constructors are *true* and *false* interpreted by $\overline{true}^X \neq \overline{false}^X$, and the operations are $\neg$ and $\wedge$ defined only on booleans by:

1. $\overline{\neg}^X(\overline{true}^X) = \overline{false}^X$
   $\overline{\neg}^X(\overline{false}^X) = \overline{true}^X$
2. $\overline{\wedge}^X(\overline{true}^X, \overline{true}^X) = \overline{true}^X$
   $\overline{\wedge}^X(\overline{true}^X, \overline{false}^X) = \overline{false}^X$
   $\overline{\wedge}^X(\overline{false}^X, \overline{true}^X) = \overline{false}^X$
   $\overline{\wedge}^X(\overline{false}^X, \overline{false}^X) = \overline{false}^X$

As usual, the other connectives can be defined with them, but in Gurevich's proof only $\neg$ and $\wedge$ are required:

1. $(A \vee B) =_{def} \neg(\neg A \wedge \neg B)$
2. $(A \Rightarrow B) =_{def} (\neg A \vee B)$
3. $(A \Leftrightarrow B) =_{def} ((A \Rightarrow B) \wedge (B \Rightarrow A))$

Usually the interpretation of a **relation** symbol R is a subset of the universe $\overline{R}^X \subseteq \mathcal{U}(X)^\alpha$. But in the following we will consider relations as function symbols because they will be interprated instead by their characteristic function χ_R:

$$\overline{\chi_R}^X(\vec{a}) =_{def} \begin{cases} \overline{true}^X & \text{if } \vec{a} \in \overline{R}^X \\ \overline{false}^X & \text{else} \end{cases}$$

[1]↑ In our paper we will consider only finite languages because we will use them to write programs.

[2]↑ See [CL03] for a more classical approach.

In particular, Gurevich's proof requires the equality symbol =, interpreted by the diagonal of the universe $\{(a,a) \mid a \in \mathcal{U}(X)\}$, so we will assume it in the following.

Definition 1.2. A **term** of a language is defined by induction:

1. if c has arity 0 then c is a term
2. if f has an arity $\alpha > 0$ and $t_1, \ldots, t_\alpha$ are terms, then $ft_1 \ldots t_\alpha$ is a term

In particular, a **formula** F will be only a boolean term:

$$F =_{def} \mathit{true} \mid \mathit{false} \mid Rt_1 \ldots t_\alpha \mid \neg F \mid (F_1 \wedge F_2)$$

where R is a relation symbol with arity α and $t_1, \ldots, t_\alpha$ are terms. Indeed, we will not use quantifiers in formulas, so we don't need the notion of logical variables. Instead, we will use the word "variables" for dynamical function symbols with arity 0.

Definition 1.3. The interpretation $\overline{t}^X$ of a term t is defined by induction on t:

1. If $t = c$ has arity 0 then $\overline{t}^X =_{def} \overline{c}^X$
2. If $t = ft_1 \ldots t_\alpha$ with $\alpha > 0$ then $\overline{t}^X =_{def} \overline{f}^X(\overline{t_1}^X, \ldots, \overline{t_\alpha}^X)$

The (unary) **integers** $\mathbb{N}$ are defined as usual. The constructors are $\emptyset$, interpreted by $\overline{\emptyset}^X = 0$, and S, defined only on unary integers by $\overline{S}^X : x \mapsto x + 1$. We can define the usual operations like $+$, $\times$ and so on... but we will not use them in the following.

Notice that the terms obtained only with constructors has the form $S^n\emptyset$ and are interpreted by $\overline{S^n\emptyset}^X = n$. For each $n \in \mathbb{N}$ we say that $S^n\emptyset$ is the representation of n. We can define the **size** $|n|$ of n by the number of symbols used in its representation. For example, the size of $\overline{+S^4\emptyset S^3\emptyset}^X$ is not 10 but 8 because $\overline{+S^4\emptyset S^3\emptyset}^X = \overline{S^7\emptyset}^X$.

In [**?**] we defined the usual types[1] and we obtained their size in the same way. But we don't need such definition of the size for our main theorem, so in the following we will only assume that we can construct a function $|.| : X \to \mathbb{N}$ giving the size of every element of the universe. This will allow us to define the time complexity of a sequential algorithm.

1.2 Sequential Algorithms

Gurevich formalized in [Gur00b] the (sequential) **algorithms** as the objects verifying three postulates : sequential time[2], abstract states and bounded exploration. The first postulate formalizes the idea that a computation is made step-by-step:

[1]↑ Booleans, integers using any base, lists, arrays, trees and graphs.
[2]↑ Gurevich studied real-time algorithms too in [GGV01].

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

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

In particular two algorithms A and B will be the same (see [BDG09]) if they have the same states, the same initial states and the same transition function.

An **execution** of the algorithm A is a sequence $\vec{X} = X_0, X_1, X_2, \dots$ such that X_0 is an initial state and for all $i \in \mathbb{N}$ we have $X_{i+1} = \tau_A(X_i)$.

At p.11 we will define an algorithmic equivalence between models of computation using executions. Notice that two algorithms have the same set of executions only if they have the same initial states and the same transition function. So, we will not consider unreachable states in the following.

We didn't define a set of terminal states because we can assume that an execution $\vec{X}$ has stopped if there exists a state X_i such that $\tau_A(X_i) = X_i$. Indeed, after X_i no more changes happen. We will say that this state X_i is final, and that an execution is **terminal** if it contains a final state. So, we can define the time of an execution of A starting at the initial state X_0:

$$time(A, X_0) =_{def} \begin{cases} min\{i \in \mathbb{N} \mid \tau_A^i(X_0) = \tau_A^{i+1}(X_0)\} & \text{if } \vec{X} \text{ is terminal} \\ \infty & \text{else} \end{cases}$$

The second postulate states that the states of an execution can be represented, and that these representations are up to isomorphisms:

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

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

For the booleans and the integers we distinguished constructors like $\emptyset$ and S from operations like $+$ and $\times$. In fact, we will distinguish $\mathcal{L}(A)$ between:

1. $Dyn(A)$: the **dynamical** symbols, whose interpretation can be changed during an execution.
2. $Stat(A)$: the **static** symbols, which have a fixed interpretation during an execution.

   They will also be distinguished between:

   a. $Init(A)$: the **initial** symbols, whose interpretation depends of the initial state.

b. The other symbols have a uniform interpretation in every state, and they will also be distinguished between the **constructors** $Cons(A)$ and the **operations** $Oper(A)$.

Because the symbols of $Cons(A) \sqcup Oper(A)$ have an interpretation which does not depend of the considered structure, the size $|X|$ of a structure will depend only of the dynamical and initial symbols, which will be called the **inputs**[1]:

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

Definition 1.4 (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 $\mathtt{Algo}_{\mathcal{C}}$ be the set of the $\mathcal{C}$-time algorithms.

We assumed that the interpretation of a dynamical symbol f with arity α can change during an execution. The **update** of f in $(a_1, \ldots, a_\alpha) \in \mathcal{U}(A)^\alpha$ with the new value $b \in \mathcal{U}(A)$ will be denoted by $(f, a_1, \ldots, a_\alpha, b)$. If X is a state and u is an update, then $X \oplus u$[2] is a new structure with the same universe and language such that:

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

If $\overline{f}^X(\vec{a}) = b$ then the update $(f, \vec{a}, b)$ will be called trivial because $X \oplus (f, \vec{a}, b) = X$.

A set Δ of updates from the same universe and language will be said **consistent** if there is no updates $(f, \vec{a}, b), (f, \vec{a}, b') \in \Delta$ such that $b \neq b'$. If we try to execute an inconsistent Δ on X then the set of updates clashes:

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

Lemma 1.5 (Difference of structure)**.** *If X and Y are two structures with the same universe and language, then there exists a consistent set Δ of non-trivial updates such that $Y = X \oplus \Delta$.*

[1]↑ We could have defined $|X|$ by $\sum_{f \in Dyn(A) \sqcup Init(A)} |f|_X$, but the two are equivalent:

$$\max_{f \in Dyn(A) \sqcup Init(A)} \{|f|_X\} \leq \sum_{f \in Dyn(A) \sqcup Init(A)} |f|_X$$
$$\leq card(Dyn(A) \sqcup Init(A)) \times \max_{f \in Dyn(A) \sqcup Init(A)} \{|f|_X\}$$

[2]↑ We denote the update by $\oplus$ and not by $+$ as in [Gur00b] because the associativity has no meaning and because we don't have the commutativity: $(X \oplus (x, 0)) \oplus (x, 1) \neq (X \oplus (x, 1)) \oplus (x, 0)$.

Proof. $\Delta = \{(f, \vec{a}, b) \mid \overline{f}^Y(\vec{a}) = b \text{ and } \overline{f}^X(\vec{a}) \neq b\}$ □

This Δ will be denoted by $Y \ominus X$, so we have $X \oplus (Y \ominus X) = Y$. For every state $X \in S(A)$ we will denote by $\Delta(A, X) =_{def} \tau_A(X) \ominus X$ the set of the updates made by A on X.

We will say that two structures X and Y coincide over a set of terms T if for all $t \in T$ we have $\bar{t}^X = \bar{t}^Y$.

The third postulate states that only a finite number of terms determines the evolution of the execution:

Postulate 3 (Bounded Exploration)**.** There exists a finite set of terms $T(A)$ closed by subterms[1] such that if two states X and Y coincide over $T(A)$ then $\Delta(A, X) = \Delta(A, Y)$.

Gurevich proved in [Gur00b] that this axiomatic presentation of the set of (sequential) algorithms `Algo` is equivalent to his operational presentation using the Abstract State Machines (ASMs).

1.3 Abstract State Machines

Definition 1.6 (`ASM` Programs)**.**

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

Where f is a dynamical symbol of arity α, $t_0, t_1, \ldots, t_\alpha$ are terms, and F is a formula.

Notation. For $n = 0$ a command `par` $\Pi_1 \| \ldots \| \Pi_n$ `endpar` is an empty program. If the `else` part of a `if` is empty we will simply write `if` F `then` Π `endif`.

Definition 1.7 (Operational Semantics of `ASM`)**.**

An `ASM` program Π defines a transition function:

$$\tau_\Pi(X) =_{def} X \oplus \Delta(\Pi, X)$$

where the set of updates $\Delta(\Pi, X)$ is defined by induction on Π:

$$\begin{aligned}\Delta(ft_1 \ldots t_\alpha := t_0, X) &=_{def} \{(f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)\} \\ \Delta(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}, X) &=_{def} \Delta(\Pi_i, X) \\ &\quad \text{where } i = 1 \text{ if } F \text{ is true in } X \\ &\quad \text{and } i = 2 \text{ if } F \text{ is false in } X \\ \Delta(\texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}, X) &=_{def} \Delta(\Pi_1, X) \cup \cdots \cup \Delta(\Pi_n, X)\end{aligned}$$

[1] $\uparrow T$ is closed by subterms if for all $ft_1 \ldots t_\alpha \in T$ we have $t_1, \ldots, t_\alpha \in T$.

The semantics of the `par` is a set of simultaneous updates, contrary to the following imperative language where the updates are sequential.

We will use at p.34 the set of the terms read[1] by Π:

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

Lemma 1.8. *If X and Y coincide over $\mathit{Read}(\Pi)$ then $\Delta(\Pi, X) = \Delta(\Pi, Y)$.*

Proof. By induction on Π. □

Remark. Notice that $\Delta(\Pi, X)$ may be inconsistent or contain updates trivial on X. Inded, we have:

$$\Delta(\tau_\Pi, X) = \tau_\Pi(X) \ominus X = (X \oplus \Delta(\Pi, X)) \ominus X \subseteq \Delta(\Pi, X)$$

But it won't be a problem, because the ASM programs used to prove Gurevich's theorem are in **normal form**:

```
      if F_1 then Π_1
else  if F_2 then Π_2
⋮
else  if F_c then Π_c
      endif ... endif
```

Where for all state X one and only one F_i is true, and where every ASM program Π_i is formed only by parallel updates such that $\Delta(\Pi, X)$ is consistent and contains no update trivial on X. In particular, for every ASM program Π in normal form we have $\Delta(\Pi, X) = \Delta(\tau_\Pi, X)$.

Theorem (Gurevich, 2000)**.** `Algo` = `ASM`

Every ASM program is a sequential algorithm, which is an ASM program in normal form. So, in the following we will assume that every ASM program is in normal form.

1.4 Fairly Simulation

Example 1.9 (Temporary Variables)**.** The following program containing a `loop`:

$$\{\texttt{loop } n \ \{s\};\}$$

is generally simulated by `while` with:

$$\{i := 0; \texttt{while } i \neq n \ \{s;\ i := i + 1;\};\}$$

[1]↑ The constructors *true* and *false* must be added to $\mathit{Read}(\Pi)$, as explained in [Gur00b] and [**?**].

But this program uses a fresh variable i, so the language is not preserved, even if in a way the implement the same behavior.

Another example[1] is to simulate the exchange $x \leftrightarrow y$ between two variables using a **temporary variable**:

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

So, the language $\mathcal{L}_1$ of the simulating program may be an extension of the language $\mathcal{L}_2$ of the simulated program: $\mathcal{L}_1 \supseteq \mathcal{L}_2$. The language must remain finite, so $\mathcal{L}_1 \setminus \mathcal{L}_2$ must be finite, but this is not enough. Using a function symbol of arity > 0 will be unfair because we will be able to store an unbounded amount of information. So, in our definition of the simulation p.11 we will only assume that $\mathcal{L}_1 \setminus \mathcal{L}_2$ is a finite set of variables (function symbols with arity 0) with a uniform initialization[2].

Definition 1.10 (Restriction of Structure)**.**

Let X be a structure of language $\mathcal{L}_1$, and let $\mathcal{L}_2$ be a language such that $\mathcal{L}_1 \supseteq \mathcal{L}_2$. The **restriction** of X to the language $\mathcal{L}_2$ will be denoted $X|_{\mathcal{L}_2}$ and is defined as a structure of language $\mathcal{L}_2$ such that:

1. $\mathcal{U}(X|_{\mathcal{L}_2}) = \mathcal{U}(X)$
2. For all $f \in \mathcal{L}_2$ we have $\overline{f}^{X|_{\mathcal{L}_2}} = \overline{f}^{X}$

Example 1.11 (Temporal Dilatation)**.**

At each computation step of a Turing machine, several actions are done, depending of the current state and of the symbol read in the current cell:

1. the state of the machine is updated
2. a new symbol may be written in the current cell
3. the head may move left or right

The notion of elementary action is very arbitrary. For example we may consider that:

- either these actions are distinct, so three steps have been made
- or only multi-action has been done, so only one step of computation

Let M_3 the Turing machine requiring three steps to make these actions, and let M_1 be the "classical" machine requiring only one step.

An execution $\vec{X}$ of M_3 can correspond to an execution $\vec{Y}$ of M_1 if for every three steps of M_3 the state is the same than M_1:

[1]↑ We will use it again p.29.

[2]↑ A $\bar{t}^{X_0}$ where t is formed only of constructors and operations (including the size, see p.7) and X_0 is an initial state.

$$\begin{array}{cc} M_3 & M_1 \\ \hline X_0 = & Y_0 \\ X_1 & \\ X_2 & \\ X_3 = & Y_1 \\ X_4 & \\ X_5 & \\ X_6 = & Y_2 \\ X_7 & \\ \vdots & \vdots \end{array}$$

If M_3 is implemented in a machine than is three times faster than the machine implementing M_1, an external observer would not be able to notice a difference: the time unit itself is arbitrary.

In this paper we will identify two executions $\vec{X}$ and $\vec{Y}$ if there exists a uniform **temporal dilatation** $d > 0$ for every execution such that for all step $i \in \mathbb{N}$ we have $X_{d\times i} = Y_i$. That means that d steps of $\vec{X}$ are required to simulate one step of $\vec{Y}$, and we will say that the simulation is step-by-step. Unfortunately, contrary to the previous example where all the machines can be simulated with a temporal dilatation $d = 3$, in our paper the temporal dilatation will depend of the simulated program.

Remark. The constraint of the temporal dilatation is not enough to ensure that the terminating behavior is the same. Indeed, even if Y_i is a terminal state for the simulated machine M_1, the states between $X_{3\times i}$ and $X_{3\times i+(d-1)}$ may alternate, so M_3 may not be terminal:

$$\begin{array}{cc} M_3 & M_1 \\ \hline X_0 = & Y_0 \\ \vdots & \vdots \\ X_{3\times i+0} = & Y_i \\ X_{3\times i+1} & \\ X_{3\times i+2} & \\ X_{3\times i+0} = & Y_i \\ \vdots & \vdots \end{array}$$

So, an **ending time** $e \geq 0$ must be added to our definition of the fairly simulation, such that $time(P_1, X) = d \times time(P_2, X) + e$, where e only depends of the simulated program:

Definition 1.12 (Fairly Simulation)**.**

Let M_1 and M_2 be two models of computation[1]. We will say that M_1 simulates M_2 if for all program P_2 of M_2 there exists a program P_1 of M_1 such that:

[1]↑ Formally, we can define a model of computation as a set of programs associated with an operational semantics. See p.12 for `LoopC` and p.8 for the ASM.

1. $\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2)$, and $\mathcal{L}(P_1) \setminus \mathcal{L}(P_2)$ is a bounded set of variables

and if there exists $d > 0$ and $e \geq 0$ depending only of P_2 such that for every execution $\vec{Y}$ of P_2 there exists an execution $\vec{X}$ of P_1 such that:

2. for all $i \in \mathbb{N}$ we have $X_{d\times i}|_{\mathcal{L}(P_2)} = Y_i$
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 thses two models of computation will be said **algorithmically equivalent**, which will be denoted by $M_1 \simeq M_2$.

Remark. Applying the second condition with $i = 0$ requires that the initial states must be the same, up to temporary variables.

According to the third condition, if the simulated execution requires n steps, then the simulating will require $O(n)$ steps, so our fairly simulation preserves the class of the time complexity defined p.7.

1.5 Imperative Language

We recall the usual commands allowed in paradigm of imperative language. We then give a operational semantics within the form of Krivine machines ([Kri07]).

Definition 1.13 (`LoopC` Programs)**.**

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

where f has arity α, $t_0, t_1, \ldots, t_\alpha$ are terms, F is a formula, and n is a variable.

Notation. Similarly to the `ASM` programs, we will write only `if` F $\{P\}$ for the command `if` F $\{P\}$ `else` $\{$`end`$\}$. Following Meyer and Ritchie's style [MR67], we will note simply `loop` n $\{P\}$ a command `loop` n `except` *false* $\{P\}$. For readability reasons, we will omit the ; `end` inside curly brackets in the examples.

The composition of commands $c; P$ can be extended to **composition of programs** $P_1; P_2$ by $\texttt{end}; P_2 =_{def} P_2$ and $(c; P_1); P_2 =_{def} c; (P_1; P_2)$. This allows us to define the operational semantics of our imperative programs. The transistion relation is denoted by $\succ$ and X denotes an environment (a mapping from variables to values). The $\oplus$ symbol is defined by $(E \oplus (u, e))(x) =$ if $x = u$ then e else $E(x)$.

Definition 1.14 (Operational Semantics of `LoopC`)**.**

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

$$\texttt{if } F \ \{P_1\} \ \texttt{else} \ \{P_2\}; P_3 \star X \succ P_1; P_3 \star X \quad \text{if } F \text{ is true in } X$$

$$\texttt{if } F \ \{P_1\} \ \texttt{else} \ \{P_2\}; P_3 \star X \succ P_2; P_3 \star X \quad \text{if } F \text{ is false in } X$$

$$\texttt{loop } n \texttt{ except } F \ \{P_1\}; P_2 \star X \succ P_1; \texttt{loop } n \texttt{ except } F \ \{P_1\}; P_2 \star X \oplus (i, \overline{i}^X + 1) \quad \text{if } i < n \text{ and } F \text{ is false in } X$$

$$\texttt{loop } n \texttt{ except } F \ \{P_1\}; P_2 \star X \succ P_2 \star X \oplus (i, 0) \quad \text{if } i = n \text{ or } F \text{ is true in } X$$

where i is a dynamical symbol initialized to 0 in the initial states and which does not appear in the program[1]. Each loop has a different counter i.

Remark. The transition system is **deterministic** and we denote $\succ_*$ the transitive closure of $\succ$. t transition steps will be denoted by $\succ_t$.

$\texttt{end} \star X$ are the only states with no following. We say that P **terminates** on X (denoted by $P \downarrow X$) if there exists t and X' such that $P \star X \succ_t \texttt{end} \star X'$. A program will be said terminal if it terminates on all its states. t and X' are unique if P terminates on X (respectively denoted by $time(P, X)$ and $P(X)$).

If $t \leq time(P, X)$[2] then there exists a unique couple P' and X' such that: $P \star X \succ_t P' \star X'$. They are denoted by $\tau_X^t(P)$ and $\tau_P^t(X)$.

In particular, if $P \downarrow X$ and $t = time(P, X)$ then $\tau_X^t(P) = \texttt{end}$ and $\tau_P^t(X) = P(X)$.

Remark. τ_P^t is not a transition function in the sense of the first postulate p.6 because $\tau_P^t(X) \neq \tau_P^1 \circ \dots \circ \tau_P^1(X)$.

The sequence of the updates made by P on X is $\tau_P^1(X) \ominus \tau_P^0(X), \tau_P^2(X) \ominus \tau_P^1(X), \dots$ where every $\tau_P^{t+1}(X) \ominus \tau_P^t(X)$ is empty or is a singleton. We can define the set of the updates made by P on X, $\Delta(P, X)$:

Definition 1.15 (Updates of an Imperative Program)**.**

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

Remark. If $P \downarrow X$ then $\Delta(P, X)$ is finite.

In the following we will say that P is without **overwrite** on X if $\Delta(P, X)$ is consistent.

Indeed, for all imperative program there is an overwrite if a variable is updated more than twice in the sequence of steps. In our framework, that means

[1]↑ The `for` loop where i is read can be simulated by $j := 0; \texttt{loop } n \texttt{ except } F \ \{P_1[j]; j := j + 1\}; P_2$.

[2]↑ If P does not terminate on X we can assume that $time(P, X) = \infty$.

that there exists in $\Delta(P, X)$ two updates $(f, \vec{a}, b)$ and $(f, \vec{a}, b')$ with $b \neq b'$, which is the definition p.**??** that $\Delta(P, X)$ is inconsistent.

Lemma 1.16 (Updates without overwrite)**.**
If $P \downarrow X$ without overwrite then $\Delta(P, X) = P(X) \ominus X$.

Proof. See [Mar14] or [**?**]. □

Moreover we can prove that the composition of programs behaves as intended for the output and the execution time:

Proposition 1.17 (Composition of Programs)**.**
$P_1; P_2 \downarrow X$ if and only if $P_1 \downarrow X$ and $P_2 \downarrow P_1(X)$, such that:

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

Proof. The proof is straightforward and is based on the determinism and transitivity of the transition rules. □

Notation. For sake of readability of imperative programs we will use the offside rule (no more semicolons, nor curly brackets, ...).

```
loop n
    n := n + 1
```

Figure 1: A non-terminal program

According to the operational semantics 1.14 p.12, the program in the figure 1 p.14 is not terminal. This program is not forbidden by the syntactical definition 1.13 p.12. As we are interested in programs running in polynomal time, we need to restrict the language, at least, to eliminate non terminating programs. Let $\texttt{LoopC}_{bound}$ be the set of the `LoopC` programs verifying this condition :

$$(bound) : \text{for all } \texttt{loop}\ n\ \{Q\} \in P, n \in Stat(Q)$$

This restriction is strongest than the restriction "loop variables can not be updated inside their scope".

Corollary 1.18 (Termination)**.**
Every program of $\texttt{LoopC}_{bound}$ is terminal.

Proof. By induction, using the proposition 1.17 p.14. □

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

Figure 2: A program for the exponential

1.6 Polynomial Time

The program P_{pow} in the figure 2 p.15 is in $\texttt{LoopC}_{bound}$ (see the condition p.14). It uses only variables, zero and successor (no more conditionals, nor conditional loop exit), so it is a usual `Loop` program [MR67] with $time(P_{pow}, X) \geq max\{\overline{n}^{P_{pow}(X)} \mid n \in Dyn(P_{pow})\}$.

The block $x := r; \texttt{loop}\ x\ \{r := r + 1\}$ computes $r := 2r$, so $\overline{r}^{P_{pow}(X)} = 2^{\overline{n}^X}$, and the program is (at least) exponential in time (and space).

To forbid this kind of programs, Neergaard [Nee03] used the Bellantoni and Cook's approach [BC92] separating **safe and normal variables**. The safe variables can be updated, and the normal variables are the bounds n in the `loop` n commands:

Definition 1.19 (Bounds of Loops)**.**

$$\begin{aligned} Bound(ft_1 \dots t_k := t_0) &=_{def} \{\} \\ Bound(\texttt{if}\ F\ \{P_1\}\ \texttt{else}\ \{P_2\}) &=_{def} Bound(P_1) \cup Bound(P_2) \\ Bound(\texttt{loop}\ n\ \texttt{except}\ F\ \{P\}) &=_{def} \{n\} \cup Bound(P) \end{aligned}$$

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

Remark. $Bound(P_1; P_2) = Bound(P_1) \cup Bound(P_2)$

Neergaard added the following condition to obtain the language `Ploop`, which will be denoted in this paper by $\texttt{LoopC}_{neer}$:

$$(neer) : \text{for all } \texttt{loop}\ n\ \{Q\} \in P, (\{n\} \cup Bound(Q)) \subseteq Stat(Q)$$

Of course $neer \Rightarrow bound$ and so $\texttt{LoopC}_{neer} \subseteq \texttt{LoopC}_{bound}$. In particular, according to the corollary 1.18 p.14, the programs are terminal. Moreover, as suggested by the name `Ploop`, Neergaard proved that computation are in polynomial time (and space) even if they use lists as data structures, since $|(d, e)| = max(|d|, |e|) + 1$.

This result can be extended to **translation** functions (in the geometric sense), those verifying that:

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

Theorem 1.20 ($\texttt{LoopC}_{neer}$ is polytime)**.**

If the static symbols are translations, then $\texttt{LoopC}_{neer}$ *is* $\mathcal{Pol}$*-time and* $\mathcal{Pol}$*-space*[1]*.*

Proof. Generalization of the proof in [Nee03]. □

As our goal is to find imperative polytime programming languages that compute all algorithms, we need to enlarge the set of data structures to be closed to "natural" programming (higher is the level of abstraction useful it is). The set of translation functions is very restrictive, for example the program 3 p.16 respects Neergaard's condition but is not polytime because $x \mapsto 2x$ is not a translation.

$r := 1$
`loop` n
$r := 2 \times r$
`loop` r

Figure 3: A non-polytime program verifying Neergaard's condition

In particular, according to our framework [**?**] for unary integers the successor is a translation but not the addition, and for the binary integers the addition is a translation but not the multiplication.

The program in the figure 3 p.16 illustrates that an exponential space in r can be converted into an exponential time using `loop` r and the composition. That's why the space must be polytime too in the theorem 1.20 p.16.

The idea to get a polytime language without constraint on the data structures is to disconnect the link between space and time. The bounds must not be updated in the whole program:

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

Then $stat \Rightarrow neer$ and so $\texttt{LoopC}_{stat} \subseteq \texttt{LoopC}_{neer}$. In particular, according to the corollary 1.18 p.14, programs are terminal.

Remark. Sadly, this language is not closed by composition. For example $n := pow(n)$ and `loop` n are in $\texttt{LoopC}_{stat}$, but not $n := pow(n); \texttt{loop}\ n$.

But $\texttt{LoopC}_{stat}$ will be useful anyway. We will prove 1.22 p.17) that programs in $\texttt{LoopC}_{stat}$ are polytime, where the degree of the complexity is the depth of the program (proposition 1.22 p.17):

[1]↑ A program P is $\mathcal{C}$-space if there exists $\varphi_P \in \mathcal{C}$ such that for all initial state X we have $|P(X)| \leq \varphi_P(|X|)$.

Definition 1.21 (Depth of a Program)**.**

$$
\begin{aligned}
depth(f t_1 \dots t_k := t_0) &=_{def} 0 \\
depth(\texttt{if } F \ \{P_1\} \ \texttt{else} \ \{P_2\}) &=_{def} max(depth(P_1), depth(P_2)) \\
depth(\texttt{loop } n \ \texttt{except } F \ \{P_1\}) &=_{def} \begin{cases} 1 + depth(P_1) & \text{if } n \in Dyn(P) \sqcup Init(P) \\ depth(P_1) & \text{if } n \in Cons(P) \sqcup Oper(P) \end{cases}
\end{aligned}
$$

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

Remark. $depth(P_1; P_2) = max(depth(P_1), depth(P_2))$

The two 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 fig. 4 p.17.

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

Figure 4: Different nesting, same depth.

Remind that for all programs P in $\texttt{LoopC}_{stat}$, $Bound(P) \subseteq Stat(P)$. So we will use for the depth only the symbols in $Init(P)$. 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 simplicity we will use the notation $Init$ for the set of the initial symbols of the program and its subprograms, and $|X|_{Init}$ for the size of these symbols in X.

Proposition 1.22 (Polynomial Time)**.**
For all $P \in \texttt{LoopC}_{\texttt{stat}}$ there exists $\varphi_P \in \mathcal{Pol}$ such that for all X:

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

Remark. The initial symbols are statics, so their interpretation in $P(X)$ is the same as in X. In particular for all program P: $|P(X)|_{Init} = |X|_{Init}$

Proof. By induction on P.

***P* = `end`**
$\varphi_P = 0$ is suitable:

1. $time(\texttt{end}, X) = 0$ so $time(P, X) \leq 0$
2. $deg(0) = 0 = depth(\texttt{end})$

$P = P_1; P_2$

By induction we assume that the proposition is true for P_1 and P_2.

We prove it for $P = P_1; P_2$ with $\varphi_{P_1;P_2} = \varphi_{P_1} + \varphi_{P_2}$.

According to the proposition 1.17 p.14:

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

$$\begin{aligned} time(P_1; P_2, X) &\leq \varphi_{P_1}(|X|_{Init}) + \varphi_{P_2}(|P_1(X)|_{Init}) \\ &= \varphi_{P_1}(|X|_{Init}) + \varphi_{P_2}(|X|_{Init}) \end{aligned}$$

2.

$$\begin{aligned} deg(\varphi_{P_1;P_2}) &= deg(\varphi_{P_1} + \varphi_{P_2}) \\ &= max(deg(\varphi_{P_1}), deg(\varphi_{P_2})) \\ &= max(depth(P_1), depth(P_2)) \\ &= depth(P_1; P_2) \end{aligned}$$

It remains to prove the proposition for the commands alone, using the induction hypothesis:

$P = ft_1 \dots t_k := t_0$

$\varphi_P = 1$ is suitable:

1. $time(ft_1 \dots t_k := t_0, X) = 1$ so $time(P, X) \leq 1$
2. $deg(1) = 0 = depth(ft_1 \dots t_k := t_0)$

$P =$ `if` F `then {`P_1`} else {`P_2`}`

We prove that $\varphi_P = 1 + \varphi_{P_1} + \varphi_{P_2}$ is suitable:

$$P \star X \succ P_i \star X$$

where $i = 1$ if F is true in X and $i = 2$ if F is false in X.

1. $time(P, X) = 1 + time(P_i, X) \leq 1 + time(P_1, X) + time(P_2, X)$
   But $time(P_1, X) \leq \varphi_{P_1}(|X|_{Init})$ and $time(P_2, X) \leq \varphi_{P_2}(|X|_{Init})$
   So $time(P, X) \leq 1 + \varphi_{P_1}(|X|_{Init}) + \varphi_{P_2}(|X|_{Init})$
2.

$$\begin{aligned} deg(\varphi_P) &= deg(1 + \varphi_{P_1} + \varphi_{P_2}) \\ &= max(deg(\varphi_{P_1}), deg(\varphi_{P_2})) \\ &= max(depth(P_1), depth(P_2)) \\ &= depth(P) \end{aligned}$$

$P =$ `loop` n `except` F `{`P_1`}`

By simplicity, we assumed p.13 that the counters are not read in the

program. So we can write the execution like this:

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

where a is the first $i \leq \overline{n}^X$ such that F is true in $P_1^a(X)$, and else $a = \overline{n}^X$.

1. So we have:

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

   But by induction hypothesis for all X: $time(P_1, X) \leq \varphi_{P_1}(|X|_{Init})$
   So $time(P_1, P_1^t(X)) \leq \varphi_{P_1}(|P_1^t(X)|_{Init}) = \varphi_{P_1}(|X|_{Init})$, and we have:

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

   We use these two inequalities for the following cases:

   $\boldsymbol{n \in Cons(P) \sqcup Oper(P)}$
   In this case there exists a constant $c_n \in \mathbb{N}_1$ such that $\overline{n}^X = c_n$
   So $\varphi_P(\vec{x}) = 1 + c_n \times (1 + \varphi_{P_1}(\vec{x}))$ (1) is suitable.

   $\boldsymbol{n \in Init(P)}$
   Because $|\overline{n}^X| = \overline{n}^X + 1$ we have $\overline{n}^X + 1 \in |X|_{Init}$.
   We assume that this is the j-th of the k elements.
   So $\varphi_P(\vec{x}) = 1 + x_j \times (1 + \varphi_{P_1}(\vec{x}))$ (2) is suitable.

2. The degree depends of the case:

   $\boldsymbol{n \in Cons(P) \sqcup Oper(P)}$

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

   $\boldsymbol{n \in Init(P)}$

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

□

Remark. Because $time(P, X) \leq \varphi_P(|X|_{Init(P)})$, using the projections we have $time(P, X) \leq \tilde{\varphi}_P(|X|)$, where $\tilde{\varphi}_P$ is polynomial of degree $depth(P)$.

2 Simulation of LoopC with ASM

2.1 Graphs of Execution

The simplest idea to translate an imperative program into an ASM is to procreed command by command , adding a line counter [1] to move from one command to another while respecting the operational semantics p.12.

Example 2.1. The following program P_{min}:

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

can be translated , see the figure 5 p.20.

```
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
```

Figure 5: Translation of P_{min}

We develop an other approach related to graphs where nodes are programs and edges are one step execution.

[1] ↑ such ASM programs are called "Control State ASMs" in [Bör05].

Definition 2.2 (Length of an imperative program)**.**

$$\begin{aligned} \mathit{length}(\texttt{end}) &=_{def} 0 \\ \mathit{length}(c; P) &=_{def} \mathit{length}(c) + \mathit{length}(P) \end{aligned}$$

$$\begin{aligned} \mathit{length}(ft_1 \dots t_k := t_0) &=_{def} 1 \\ \mathit{length}(\texttt{if } F \ \{P_1\} \ \texttt{else} \ \{P_2\}) &=_{def} 1 + \mathit{length}(P_1) + \mathit{length}(P_2) \\ \mathit{length}(\texttt{loop } n \ \texttt{except } F \ \{P\}) &=_{def} 1 + \mathit{length}(P) \end{aligned}$$

Remark. The number of a line is the length of the program before the current command, so the line numbers are between 0 and $\mathit{length}(P)$. Because the number of values is bounded, instead of an instruction counter whose value will change a finite number of booleans $b_0, b_1, \dots b_{\mathit{length}(P)}$ can be used.

This approach was suggested in [GV12], and is suitable for a programming language based on line numbers (for example using `goto` commands) but not for a structured programming language such as `LoopC`. Indeed, the line number can distinguish commands which are identical for the operational semantics:

Example 2.3. We distinguish at the figure 6 p.21 the commands $\boxed{x := x + 1}$ and $\boxed{x := x + 1}$ by an external marking.

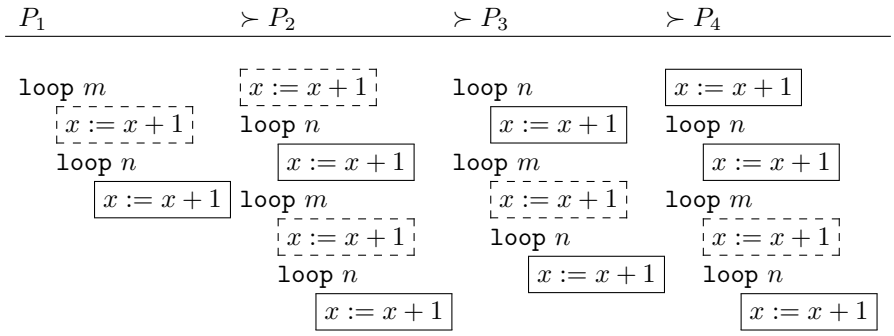


Figure 6: Loops with marking

From an operational point of view these commands are identical: we could have replaced P_2 by P_4 without changing the following of the execution. The same remark can be applied to conditionals, as showed at the figure 7 p.22.

Thus, a translation based on the line numbers could be defined, but to be distinguished the commands should be marked by their depth (for nested loops) and the path of the conditionals reached during the execution. In addition to being tedious, this approach would be useless because these commands would identical anyway, from an operational point of view.

So, we will not use the booleans $\{b_i \mid 0 \leq i \leq \mathit{length}(P)\}$ indexed by the line numbers, but instead we will index them by all the programs which can be reached during the execution:

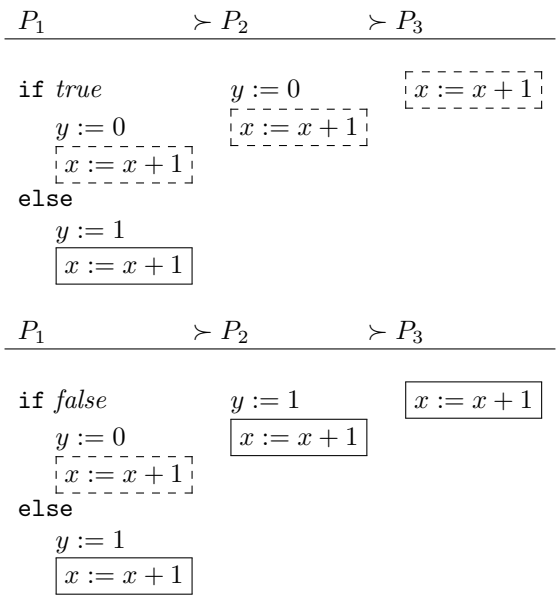
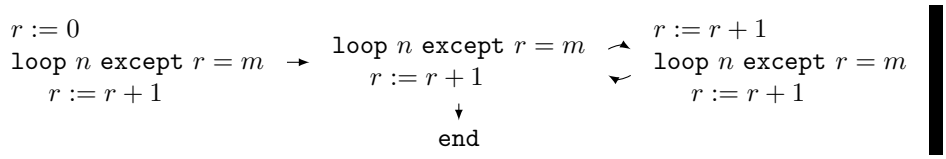


Figure 7: Conditionals with marking

Example 2.4. (Graph of execution for P_{min})[1]

The possible executions of a program may be represented by a graph where the edges are the possible transitions, and the vertices are possible programs:



In the following we will only need the set of vertices to index the booleans, so the graph of execution for P_{min} will be denoted by:

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

Notation. To define formally the graphs of execution, we need to introduce the notation $\mathcal{G}(P_1); P_2$. If $\mathcal{G}$ is a set of imperative programs and P is an imperative program, then:

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

[1]↑ Introduced before as example for the operational semantics ?

Remark. According to the definition of $\mathcal{G};P$:

1. $card(\mathcal{G};P) = card(\mathcal{G})$
2. $(\mathcal{G}_1;P) \cup (\mathcal{G}_2;P) = (\mathcal{G}_1 \cup \mathcal{G}_2);P$
3. $\mathcal{G}_1 \subseteq \mathcal{G}_2 \Rightarrow \mathcal{G}_1;P \subseteq \mathcal{G}_2;P$

Definition 2.5 (Graphs of execution)**.**

$$\begin{array}{rl} \mathcal{G}(\texttt{end}) & =_{def} \{\texttt{end}\} \\ \mathcal{G}(c;P) & =_{def} \mathcal{G}(c);P \cup \mathcal{G}(P) \\ \\ \mathcal{G}(t_1 \dots t_k := t_0) & =_{def} \{t_1 \dots t_k := t_0;\texttt{end}\} \\ \mathcal{G}(\texttt{if } F \ \{P_1\} \ \texttt{else} \ \{P_2\}) & =_{def} \{\texttt{if } F \ \{P_1\} \ \texttt{else} \ \{P_2\};\texttt{end}\} \cup \mathcal{G}(P_1) \cup \mathcal{G}(P_2) \\ \mathcal{G}(\texttt{loop } n \ \texttt{except } F \ \{P\}) & =_{def} \mathcal{G}(P);\texttt{loop } n \ \texttt{except } F \ \{P\};\texttt{end} \end{array}$$

Remark. `end` and $P \in \mathcal{G}(P)$

As intended, the number of possible programs occurring in an execution is bounded:

Lemma 2.6 (Finiteness of a graph of execution)**.**

$$card(\mathcal{G}(P)) \leq length(P) + 1$$

Proof. The proof is made in [**?**] by induction. □

So we will need only a bounded number of booleans to save the current states of the program during the execution.

Remark. For programs like P_{min} in the example 18 p.22, $card(\mathcal{G}(P)) = length(P)+1$ can effectively be reached. But the marked programs p.21 show that the inequality sign cannot be replaced by an equality sign. So this bound is optimal.

So we will use the set of booleans $\{b_{P_j} \mid P_j \in \mathcal{G}(P)\}$ to memorize the states of the program at each step of the execution. But in our translation from P into Π_P we need to prove [prop. p.23]: if $P_j \in \mathcal{G}(P)$ then the next step program is in $\mathcal{G}(P)$. We use the two following technical lemmas (proof are by induction on P_1 (resp. P)):

Lemma 2.7 (Composition of graphs of execution)**.** $\mathcal{G}(P_1;P_2) = \mathcal{G}(P_1);P_2 \cup \mathcal{G}(P_2)$.

Lemma 2.8 (Subgraphs of execution)**.** *If* $Q \in \mathcal{G}(P)$ *then* $\mathcal{G}(Q) \subseteq \mathcal{G}(P)$.

We can now prove that if a program appears in $\mathcal{G}(P)$ then its potential successors for the operational semantics p.12 appear in $\mathcal{G}(P)$ too:

Proposition 2.9 (Operational closure of the graphs of execution)**.**

1. *If* $t_1 \dots t_k := t_0;Q \in \mathcal{G}(P)$ *then* $Q \in \mathcal{G}(P)$

2. *If* `if` F $\{P_1\}$ `else` $\{P_2\}; Q \in \mathcal{G}(P)$ *then* $P_1; Q$ *and* $P_2; Q \in \mathcal{G}(P)$

3. *If* `loop` n `except` F $\{P_1\}; Q \in \mathcal{G}(P)$ *then* P_1;`loop` n `except` F $\{P_1\}; Q$ *and* $Q \in \mathcal{G}(P)$

Proof. The proof is made by case:

$t_1 \dots t_k := t_0; Q \in \mathcal{G}(P)$
$\{t_1 \dots t_k := t_0; Q\} \cup \mathcal{G}(Q) = \mathcal{G}(t_1 \dots t_k := t_0; Q) \subseteq \mathcal{G}(P)$ according to the lemma 2.8. But $Q \in \mathcal{G}(Q)$ and so $Q \in \mathcal{G}(P)$.

`if` F $\{P_1\}$ `else` $\{P_2\}; Q \in \mathcal{G}(P)$
$\{$`if` F $\{P_1\}$ `else` $\{P_2\}; Q\} \cup \mathcal{G}(P_1); Q \cup \mathcal{G}(P_2); Q \cup \mathcal{G}(Q)$
$= \mathcal{G}($`if` F $\{P_1\}$ `else` $\{P_2\}; Q) \subseteq \mathcal{G}(P)$ according to the lemma 2.8. But $P_1; Q \in \mathcal{G}(P_1); Q$ and $P_2; Q \in \mathcal{G}(P_2); Q$ and so $P_1; Q$ and $P_2; Q \in \mathcal{G}(P)$.

`loop` n `except` F $\{P_1\}; Q \in \mathcal{G}(P)$
$\mathcal{G}(P_1)$;`loop` n `except` F $\{P_1\}; Q \cup \mathcal{G}(Q) = \mathcal{G}($`loop` n `except` F $\{P_1\}; Q)$ $\subseteq \mathcal{G}(P)$ according to the lemma 2.8. But P_1;`loop` n `except` F $\{P_1\}; Q \in$ $\mathcal{G}(P_1)$;`loop` n `except` F $\{P_1\}; Q$ and $Q \in \mathcal{G}(Q)$.
So P_1;`loop` n `except` F $\{P_1\}; Q$ and $Q \in \mathcal{G}(P)$

□

2.2 Translation of an Imperative Program

Notation. According to the previous subsection, we will use the set of fresh variables $\{b_{P_j} \mid P_j \in \mathcal{G}(P)\}$. Only one of the b_{P_j}'s will be true at each step of the run, so in the following we will note $X[b_{P_j}]$ the only b_{P_j} that is true where X is a $\mathcal{L}(P)$-structure. In particular we have $X[b_{P_j}]|_{\mathcal{L}(P)} = X$.

To change the current state of the execution from P_j to P_k we only have to use the set of updates $\{(b_{P_j}, \overline{false}), (b_{P_k}, \overline{true})\}$. This set is consistent only if there is no program P such that one successor of P can be P itself[1].

It's only the case when $P =$ `loop` n `except` F $\{$`end`$\}; Q$ if $i < n$ and F is false (see 1.14 p.12).

So, if the body ??(?) of the loop is empty, instead of doing inconsistant updates we will simply don't change the state. But because the state is not changed we must verify that the ASM program do not stop . Fortunately, the loop counter i will be updated, so the execution can continue.

The proposition 2.9 ensures that the following translation is well defined:

Definition 2.10 (Translation of an Imperative Program)**.**

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

where P_j^{tr} is defined at the figure 8 p.25.

[1] $\uparrow \tau_X^1(P) = P$... notation introduced before ? To define $\Delta(P, X)$? $P(X) \ominus X$ enough ? I think so, if some proofs are admitted...

$$
\begin{array}{rl}
(\texttt{end})^{tr} =_{def} & \texttt{par endpar} \\
\\
(ft_1 \ldots t_k := t_0; Q)^{tr} =_{def} & \texttt{par} \\
& \quad b_{ft_1 \ldots t_k := t_0; Q} := \mathit{false} \\
& \| \; ft_1 \ldots t_k := t_0 \\
& \| \; b_Q := \mathit{true} \\
& \texttt{endpar} \\
\\
(\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}; Q)^{tr} =_{def} & \texttt{par} \\
& \quad b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}; Q} := \mathit{false} \\
& \| \; \texttt{if } F \texttt{ then} \\
& \qquad b_{P_1;Q} := \mathit{true} \\
& \quad \texttt{else} \\
& \qquad b_{P_2;Q} := \mathit{true} \\
& \quad \texttt{endif} \\
& \texttt{endpar} \\
\\
(\texttt{loop } n \texttt{ except } F \; \{\texttt{end}\}; Q)^{tr} =_{def} & \texttt{if } (i \neq n \wedge \neg F) \texttt{ then} \\
& \quad i := i + 1 \\
& \texttt{else} \\
& \quad \texttt{par} \\
& \qquad b_{\texttt{loop } n \texttt{ except } F \; \{\texttt{end}\}; Q} := \mathit{false} \\
& \quad \| \; i := 0 \\
& \quad \| \; b_Q := \mathit{true} \\
& \quad \texttt{endpar} \\
& \texttt{endif} \\
\\
(\texttt{loop } n \texttt{ except } F \; \{c; P_1\}; Q)^{tr} =_{def} & \texttt{par} \\
& \quad b_{\texttt{loop } n \texttt{ except } F \; \{c;P_1\};Q} := \mathit{false} \\
& \| \; \texttt{if } (i \neq n \wedge \neg F) \texttt{ then} \\
& \qquad \texttt{par} \\
& \qquad\quad i := i + 1 \\
& \qquad\quad \| \; b_{c;P_1;\texttt{loop } n \texttt{ except } F \; \{c;P_1\};Q} := \mathit{true} \\
& \qquad \texttt{endpar} \\
& \quad \texttt{else} \\
& \qquad \texttt{par} \\
& \qquad\quad i := 0 \\
& \qquad\quad \| \; b_Q := \mathit{true} \\
& \qquad \texttt{endpar} \\
& \quad \texttt{endif} \\
& \texttt{endpar}
\end{array}
$$

Figure 8: Translation of an Imperative Program

Example 2.11. Remind that the graph of execution for P_{min} obtained p.22 is:

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

So, the translation $\Pi_{P_{min}}$ of this imperative program into an `ASM` program is obtained fig.9 p.27[1].

Proposition 2.12 (Step by Step Simulation)**.**
For all $0 \leq t < time(P, X)$*:* $\tau_{\Pi_P}(\tau_P^t(X)[b_{\tau_X^t(P)}]) = \tau_P^{t+1}(X)[b_{\tau_X^{t+1}(P)}]$[2]

Proof. As $t < time(P, X)$ so $\tau_X^t(P) \neq \texttt{end}$. The proof is made by case on $\tau_X^t(P)$ (see Appendice p.**??**. □

2.3 The Simulation

Theorem 2.13. `ASM` *simulates* `LoopC`*.*

Proof. Remind that our simulation p.11[3] requires three conditions:

1. $\mathcal{L}(\Pi_P) = \mathcal{L}(P) \cup \{b_{P_j} \mid P_j \in \mathcal{G}(P)\}$
   where $card(\{b_{P_j} \mid P_j \in \mathcal{G}(P)\}) \leq length(P) + 1$, according to the lemma 2.6.
   So~~, in the worst case,~~ we use at most $\boxed{length(P) + 1}$ temporary variables.

2. We prove by induction on $0 \leq t \leq time(P, X)$ that:
   $$\tau_{\Pi_P}^t(X[b_P]) = \tau_P^t(X)[b_{\tau_X^t(P)}]$$
   $\boldsymbol{i = 0}$
   $\tau_{\Pi_P}^0(X[b_P]) = X[b_P] = \tau_P^0(X)[b_{\tau_X^0(P)}]$
   $\boldsymbol{i \Rightarrow i + 1}$
   $\tau_{\Pi_P}^{t+1}(X[b_P])$
   $= \tau_{\Pi_P}(\tau_{\Pi_P}^t(X[b_P]))$
   $= \tau_{\Pi_P}(\tau_P^t(X)[b_{\tau_X^t(P)}])$ according to the induction hypothesis
   $= \tau_P^{t+1}(X)[b_{\tau_X^{t+1}(P)}]$ according to the proposition 2.12
   with $0 \leq t < t + 1 \leq time(P, X)$

   So we have for all $t \in \mathbb{N}$:
   $$\tau_{\Pi_P}^t(X[b_P])|_{\mathcal{L}(P)} = \tau_P^t(X)[b_{\tau_X^t(P)}]|_{\mathcal{L}(P)} = \tau_P^t(X)$$

   Thus, the temporal dilatation is $\boxed{d = 1}$

[1]↑ We could remove the last conditional, but we prefer a homogeneous presentation.
[2]↑ Introduction of $\tau_X^t(P)$ and $\tau_P^t(X)$!
[3]↑ Introduction at the section Framework...

```
par
   if b_{r:=0;loop n except r=m {r:=r+1;};end} then
      par
         b_{r:=0;loop n except r=m {r:=r+1;};end} := false
       ‖ r := 0
       ‖ b_{loop n except r=m {r:=r+1;};end} := true
      endpar
   endif
 ‖ if b_{loop n except r=m {r:=r+1;};end} then
      par
         b_{loop n except r=m {r:=r+1;};end} := false
       ‖ if (i ≠ n ∧ r ≠ m) then
            par
               i := i + 1
             ‖ b_{r:=r+1;loop n except r=m {r:=r+1;};end} := true
            endpar
         else
            par
               i := 0
             ‖ b_end := true
            endpar
         endif
      endpar
   endif
 ‖ if b_{r:=r+1;loop n except r=m {r:=r+1;};end} then
      par
         b_{r:=r+1;loop n except r=m {r:=r+1;};end} := false
       ‖ r := r + 1
       ‖ b_{loop n except r=m {r:=r+1;};end} := true
      endpar
   endif
 ‖ if b_end then
      par
      endpar
   endif
endpar
```

Figure 9: Translation of P_{min} into an ASM program

3. We prove by inequalities that $time(\Pi_P, X[b_P]) = time(P, X)$:

   ***time*$(\Pi_P, X[b_P]) \leq$ *time*(P, X)**

   If $t = time(P, X)$ then $\tau_X^t(P) = \texttt{end}$, so:

   $\Delta(\Pi_P, \tau_P^t(X)[b_{\tau_X^t(P)}])$
   $= \Delta(\texttt{end}^{tr}, \tau_P^t(X)[b_{\tau_X^t(P)}])$
   $= \Delta(\texttt{par endpar}, \tau_P^t(X)[b_{\tau_X^t(P)}])$
   $= \varnothing$

   So we have:

   $\tau_{\Pi_P}^{t+1}(X[b_P])$
   $= \tau_{\Pi_P}(\tau_{\Pi_P}^t(X[b_P]))$
   $= \tau_{\Pi_P}(\tau_P^t(X)[b_{\tau_X^t(P)}])$
   $= \tau_P^t(X)[b_{\tau_X^t(P)}] + \Delta(\Pi_P, \tau_P^t(X)[b_{\tau_X^t(P)}])$
   $= \tau_P^t(X)[b_{\tau_X^t(P)}]$
   $= \tau_{\Pi_P}^t(X[b_P])$

   Thus, we have $time(\Pi_P, X[b_P]) \leq time(P, X)$.

   ***time*$(\Pi_P, X[b_P]) \geq$ *time*(P, X)**

   The proof is made by case for all $0 \leq t < time(P, X)$:

   $\tau_X^{t+1}(P) \neq \tau_X^t(P)$

   In that case $\tau_P^{t+1}(X)[b_{\tau_X^{t+1}(P)}] \neq \tau_P^t(X)[b_{\tau_X^t(P)}]$.
   So $\tau_{\Pi_P}^{t+1}(X[b_P]) \neq \tau_{\Pi_P}^t(X[b_P])$.

   $\tau_X^{t+1}(P) = \tau_X^t(P)$

   According to the transition system 1.14 p.12 it is only possible in the case

   $$P = \texttt{loop}\ n\ \texttt{except}\ F\ \{\texttt{end}\}; Q$$

   if $i < n$ and F is false.
   But in that case $\tau_P^{t+1}(X) = \tau_P^t(X) \oplus (i, \overline{i}^{\tau_P^t(X)} + 1)$
   So $\tau_P^{t+1}(X)[b_{\tau_X^{t+1}(P)}] \neq \tau_P^t(X)[b_{\tau_X^t(P)}]$.
   Thus $\tau_{\Pi_P}^{t+1}(X[b_P]) \neq \tau_{\Pi_P}^t(X[b_P])$.

   In any case, we have $\tau_{\Pi_P}^{t+1}(X[b_P]) \neq \tau_{\Pi_P}^t(X[b_P])$.
   So $time(\Pi_P, X[b_P]) \geq time(P, X)$.

   So we have $time(\Pi_P, X[b_P]) = time(P, X)$, thus $\boxed{e = 0}$

□

3 Simulation of $\texttt{ASM}_{\mathcal{P}ol}$ with $\texttt{LoopC}_{\texttt{stat}}$

We prove the simulation in three steps:

1. Translate Π into an imperative program P_{step} simulating one step of the ASM.

2. Repeat P_{step} enough time, depending of c_Π the complexity of Π.

3. Ensure that the final program stops at the the same time than the ASM, up to temporal dilatation.

3.1 Translation of one Step

Because `ASM` and `Imp` have the same updates and conditionals, an intuitive translation could be:

$$\begin{aligned}(f t_1 \dots t_k := t_0)^{tr} &=_{def} f t_1 \dots t_k := t_0; \texttt{end} \\ (\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})^{tr} &=_{def} \texttt{if } F \texttt{ then } \{\Pi_1^{tr}\} \texttt{ else } \{\Pi_2^{tr}\}; \texttt{end} \\ (\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar})^{tr} &=_{def} \Pi_1^{tr}; \dots; \Pi_n^{tr}\end{aligned}$$

But remember that the updates are parallel and not sequential in the ASM.

Example 3.1. The ASM program Π:

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

exchanges the value of x and y, but its translation Π^{tr}:

$$x := y; y := x; \texttt{end}$$

replaces the value of x by the value of y.

To capture the simultaneous behavior of the ASM, we need to substitute the terms read in Π by fresh temporary variables:

Definition 3.2 (Substitution of a term by a variable)**.**

$$\begin{aligned}(f t_1 \dots t_k := t_0)[v/t] &=_{def} f t_1[v/t] \dots t_k[v/t] := t_0[v/t] \\ (\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})[v/t] &=_{def} \texttt{if } F[v/t] \texttt{ then } \Pi_1[v/t] \\ &\qquad \texttt{else } \Pi_2[v/t] \texttt{ endif} \\ (\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar})[v/t] &=_{def} \texttt{par } \Pi_1[v/t] \| \dots \| \Pi_n[v/t] \texttt{ endpar}\end{aligned}$$

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

Remark. Because the temporary variables are fresh, if t_1 and t_2 are distinct terms then $\Pi[v_{t_1}/t_1][v_{t_2}/t_2] = \Pi[v_{t_2}/t_2][v_{t_1}/t_1]$. In particular, for the set of terms $\vec{t}$ of $Read(\Pi)$ (see p.9), 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 it remains two problems:

1. The variables $v_{t_1}, \dots, v_{t_r}$ must be initialized with the values of the terms $\{t_1, \dots, t_r\} = Read(\Pi)$. So the program $\vec{v_t} := \vec{t}$ must be computed before $\Pi[\vec{v_t}/\vec{t}]^{tr}$.

2. According to the definition 1.12 p.11 the temporal dilatation is uniform, so P_{step} should compute the same number of steps for all possible initial states. To dot that we pad [FZG10] the program using `skip` n commands, defined by:

$$\begin{array}{ll} \texttt{skip}\ 0 & =_{def} \texttt{end} \\ \texttt{skip}\ n+1 & =_{def} \texttt{if}\ \mathit{true}\ \{\};\texttt{skip}\ n \end{array}$$

We can assume[1] that Π is in normal form, so its translation has the form:

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

Because Π is in normal form, for all X only one F_i is true. So exactly i steps are done to enter in the block $\{\Pi_i^{tr}\}$. Then m_i updates are are done in the block, and the program stops. The number of steps done is $i + m_i \leq c + m$, where $m = max\{m_i \mid 1 \leq i \leq c\}$. So, it is sufficient to add `skip` $(c + m) - (i + m_i)$ at the end of each block $\{\Pi_i^{tr}\}$ to compute them in $c + m$ steps for each possible state X.

We obtain at the figure 10 p.31 the program P_{step} which simulates one step of the ASM program Π in a constant number of steps:

Proposition 3.3 (Translation[2] of one step of an ASM)**.**

1. $(P_{step}(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$
2. $time(P_{step}, X) = r + c + m = t_\Pi$

where:

- r *is the number of terms read by* Π *(see p.??)*
- c *is the number of states recognized by* Π *(voir p.??)*
- m *is the parallelism value of* Π *(voir p.??)*

Proof. The initialization requires r steps.

For each variable v_t and for each state Y after the initialization we have:

$$\overline{v_t}^Y = \bar{t}^X$$

In particular for each conditional F_i: $\overline{v_{F_i}}^Y = \overline{F_i}^X$

Because Π is in normal form, exactly one F_i is true in X, let F_j be this formula.

[1]↑ ref!

[2]↑ Restriction of structures, r and c

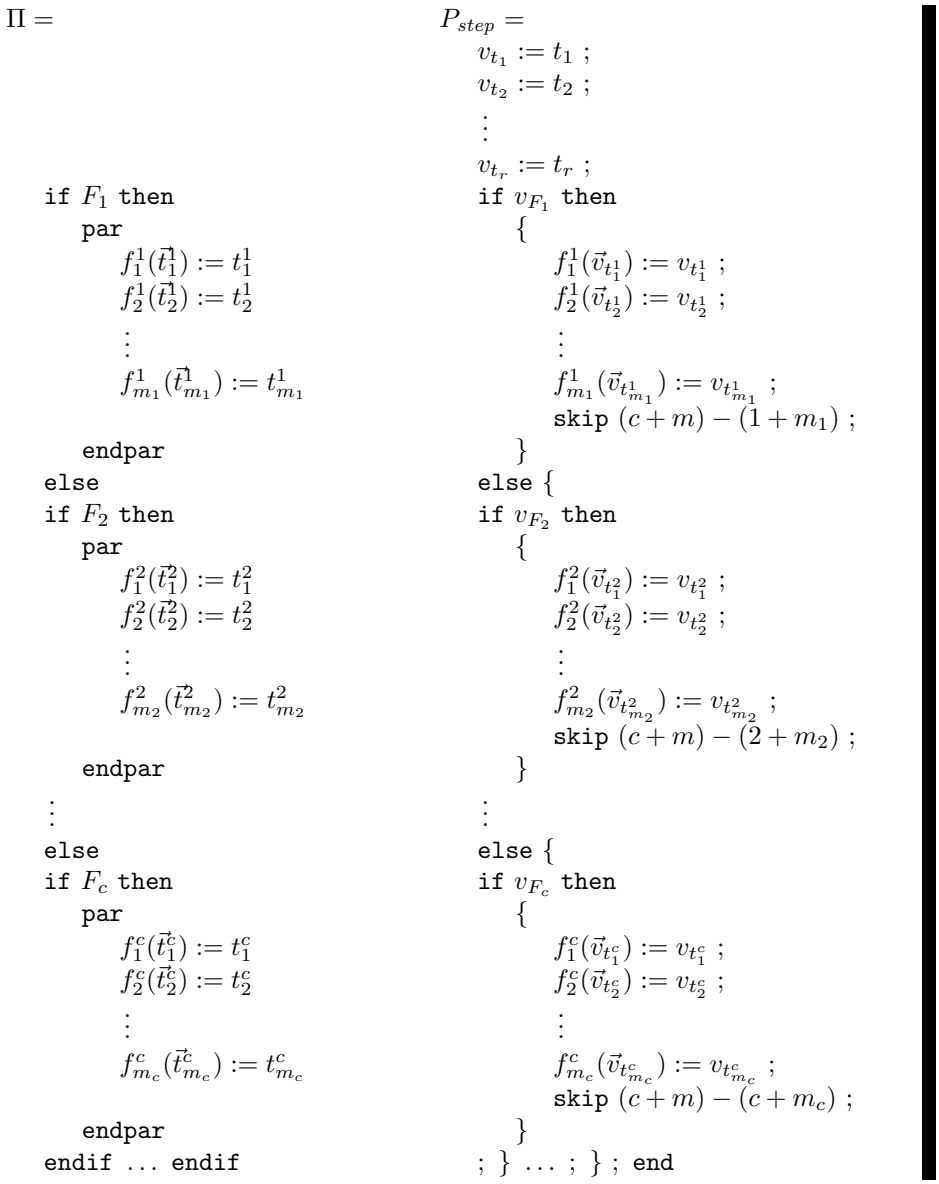


Figure 10: Translation P_{step} of the ASM program Π

The computation requires j steps to reach the block of F_j, then m_j steps for the updates. Then, because of the command `skip` $(c+m)-(j+m_j)$, we obtain an execution time depending only of Π:

$$time(P_{step}, X) = r + (j + m_j) + (c + m) - (j + m_j) = r + c + m$$

After the initialization of the variables, the updates done in the block are $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$:

$$\Delta(P_{step}, X) = \{(v_t, \bar{t}^X) \mid t \in Read(\Pi)\} \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$$

The temporary variables are updated only once during the initialization, then because the ASM is in norml form we have that $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$ is consistent.

So P_{step} is without overwrite, and because it is terminal according to the lemma 1.16 we have:

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

From these two equalities, we obtain that:

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

□

3.2 Translation of the Complexity

P_{step} simulates in constant time one step of the ASM program Π, so we want to repeat it "enough"' 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:

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

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

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

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

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

And 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 $|X|$[1]:

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

Notice that according to the definition 1.21 p.16, the depth of this program is $deg(\varphi_\Pi)$.

The intuitive program to repeat P_{step} is :

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

But between two executions of P_{step} the number of steps depends of the actual depth in the program, so the simulation will not have a constant temporal dilatation. 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[2] P_{step} before each body of a loop (case where the execution enter inside a loop) and after each `loop` command (case where the execution erase a loop):

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

In fact, 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:

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

Actually the temporal dilatation is $d = t_\Pi + 1$ because the program alternates between `loop` commands and execution of P_{step}. But we can't ensure that the

[1]↑ Préciser au début que nous supposons avoir accès à $|X|$.

[2]↑ Like in [APV10], with the difference that we choose to have each execution of P_{step} after each command and not before. This will make sense when we will add one occurrence of P_{step} before the program to initialize the μ-formula F_Π.

program stops at the same time than Π, so we need to detect the end of the execution.

Notation. The μ-formula[1] of Π is defined by:

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

In P_{step} the temporary variables $\vec{v_t}$ store the value of the terms $\vec{t}$ read by Π, then the terms $\vec{t}$ are updated in the block corresponding to the current state. So, after each execution of P_{step}, the $\vec{v_t}$ have the old values of the terms, and the terms $\vec{t}$ have the new values. So, if $\vec{v_t} = \vec{t}$ for all terms in $Read(\Pi)$ then the new values are equal to the old values, so the program had not updated the symbols, so the ASM has stopped:

Lemma 3.4 (μ-formula)**.**

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

Proof. Remind that:

$$time(A, X_0) =_{def} \begin{cases} min\{i \in \mathbb{N} \mid \tau_A^i(X_0) = \tau_A^{i+1}(X_0)\} & \text{if } \vec{X} \text{ is terminal} \\ \infty & \text{else} \end{cases}$$

The two sides of:

$$\tau_\Pi^i(X|_{\mathcal{L}(\Pi)}) = \tau_\Pi^{i+1}(X|_{\mathcal{L}(\Pi)}) \Leftrightarrow \overline{F_\Pi}^{P_{step}^{i+1}(X)} = true$$

are proven in [**?**] and [FI]. □

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

```
loop c except F_Π
   if ¬F_Π {P_step}
   loop N except F_Π
      if ¬F_Π {P_step}
      ⋱
         loop N 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 dilatation becomes $d = t_\Pi + 2$ because entering in the conditionals costs one step more. But there are two problems remaining:

[1]↑ We choose this name because this formula is extremely similar to the μ-schema of the recursive functions, see our paper [Mar15] for a version with `while` commands.

1. Actually the program cannot evaluate properly the first μ-formula F_Π because the temporary variables have not been initialized yet. But notice that F_Π becomes true after $time(\Pi, X|_{\mathcal{L}(\Pi)}) + 1$ steps. So we add an occurrence of P_{step} at the beginning of the program, and after this initialization there will be exactly $time(\Pi, X|_{\mathcal{L}(\Pi)}) + 1$ repetitions of P_{step} until F_Π becomes true.

2. The simulation is correct until F_Π becomes true, and after that the remaining steps consist to erase the last `loop` commands. Because their number depends of the current depth, determined by the initial state, we cannot set a constant ending time for the simulation. But the number of remaining `loop` commands is bounded by $deg(\varphi_\Pi)+1$, so the current ending time can be bounded too. In fact, for each remaining `loop` commands two steps are done: erase the loop then erase the following `if` F $\{P_{step}\}$, so the ending time is bounded by $max_{end} = 2 \times (deg(\varphi_\Pi) + 1)$.

Using a fresh variable i_{end} counting the number of steps done after F_Π became true, we can add at the end of the program the program `skip` $i_{end} \to max_{end}$[1] defined by:

$$\begin{array}{ll} \texttt{skip } i \to 0 & =_{def} \texttt{end} \\ \texttt{skip } i \to m+1 & =_{def} \texttt{if } i = m+1 \texttt{ \{end\} else \{skip } i \to m\}\texttt{; end} \end{array}$$

For all 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 each `if` $\neg F_\Pi$ $\{P_{step}\}$ by `if` $\neg F_\Pi$ $\{P_{step}\}$ `else` $\{i_{end} := i_{end}+3\texttt{; end}\}$, and now $max_{end} = 3 \times (deg(\varphi_\Pi) + 1)$.

3.3 The Simulation

For each ASM program Π we obtain at the figure 11 p.36 its translation P_Π simulating the execution of Π:

Theorem 3.5. $\texttt{LoopC}_{\texttt{stat}}$ *simulates* $\texttt{ASM}_{\mathcal{P}ol}$.

Proof. Remind that our simulation p.11 requires three conditions:

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 number of fresh variables depending only of Π.

[1] $\uparrow max_{end}$ does not depend of the initial state so we can use constructors for it and define `skip` $i \to m$ by induction on m. We cannot do that for c because contrary to the conditional which accepts terms, the loop commands can only be bounded by a variable.

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

Figure 11: Translation P_Π of the ASM program Π

2. Until F_Π becomes true the execution alternates between:

   a. t_Π steps of P_{step}, simulating one step of Π according to the proposition 3.3 p.30.

   b. One step to enter in the body of a loop, or erase a loop command.

   c. One step to enter in the conditional `if` $\neg F_\Pi$ $\{P_{step}\}$, then back to a.

   So, each step of Π is simulated by exactly $\boxed{d = t_\Pi + 2}$ steps of its translation P_Π.

   Moreover, the execution is long enough because if c and *size* are initialized respectively with $\sum_{0 \leq n \leq deg(\varphi_\Pi)} max(0, a_n)$ and $|X_0|$ 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, because c does not depend of the chosen initial state we have, according to the definition 1.21 p.16:

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

3. So, $time(\Pi, X|_{\mathcal{L}(\Pi)})$ repetitions of these three steps simulate the ASM program. Then, according to the lemma 3.4 p.34, t_Π more steps for the last iteration of P_{step} make F_Π true[1].

   Then, until `skip` $i_{end} \to max_{end}$ is reached, the execution alternates between:

[1]↑ Even if $time(\Pi, X|_{\mathcal{L}(\Pi)}) = 0$, in which case this is the initial P_{step} which will execute all the steps.

a. One step to erase a loop command.

b. One step to enter in the else part of the conditional `if` $\neg F_\Pi$.

c. One step of $i_{end} := i_{end} + 3$, then back to a.

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 are made by `skip` $i_{end} \to max_{end}$. So, the ending time is:

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

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

□

4 Conclusion and Discussion

Theorem. $\texttt{LoopC}_{\texttt{stat}} \simeq \texttt{Algo}_{\mathcal{P}ol}$.

Proof. According to the theorem 10 p.9 `Algo` = `ASM`, so we need to prove that $\texttt{LoopC}_{\texttt{stat}} \simeq \texttt{ASM}_{\mathcal{P}ol}$.

According to the theorem 2.13 p.26 `ASM` simulates `LoopC`.

But according to the proposition 1.22 p.17, the programs of $\texttt{LoopC}_{\texttt{stat}}$ are in polynomial time, such that $deg(\varphi_P) = depth(P)$. So $\texttt{ASM}_{\mathcal{P}ol}$ simulates $\texttt{LoopC}_{\texttt{stat}}$.

According to the theorem 3.5 p.35 $\texttt{LoopC}_{\texttt{stat}}$ simulates $\texttt{ASM}_{\mathcal{P}ol}$, such that $depth(P_\Pi) = deg(\varphi_\Pi)$.

So, according to our definition of the simulation p.?? $\texttt{LoopC}_{\texttt{stat}} \simeq \texttt{Algo}_{\mathcal{P}ol}$. Morevoer, the degree of the complexity is the depth p.16 of the program. □

In other words, because the programs of $\texttt{LoopC}_{\texttt{stat}}$ are in polynomial time, and $\texttt{LoopC}_{\texttt{stat}} \simeq \texttt{Algo}_{\mathcal{P}ol}$, we have that $\texttt{LoopC}_{\texttt{stat}}$ is **algorithmically complete**.

From the programmer's point of view :

- It's easy to test the practical complexities the degree of the polynom which is readable directly from the program.
- But $\texttt{LoopC}_{stat}$ is not fully compositional. Indeed, in $P_1; P_2$, we must verify that the inputs of P_2 are not outputs of P_1.
- And moreover, it's difficult to program in $\texttt{LoopC}_{stat}$ because the complexity must be anticipated before writing the program.

But remind that $\texttt{LoopC}_{stat}$ does not need constraints on data structures to be polytime, unlike $\texttt{LoopC}_{neer}$. To have the better of the two worlds, it would be nice to construct an intermediary language, between $\texttt{LoopC}_{stat}$ and $\texttt{LoopC}_{neer}$.

References

[APV10] Philippe Andary, Bruno Patrou, and Pierre Valarcher. A theorem of representation for primitive recursive algorithms. *Fundamenta Informaticae*, XX:1–18, 2010. 33

[APV11] Ph. Andary, B. Patrou, and P. Valarcher. A theorem of representation for primitive recursive algorithms. *Fundamenta Informaticae*, 107:313–330, 2011. 2, 3

[BC92] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992. 2, 15

[BDG09] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. When are two algorithms the same? *Bullettin of Symbolic Logic*, 15:145–168, 2009. 6

[Bon06] Guillaume Bonfante. Some programming languages for logspace and ptime. In *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, pages 66–80, 2006. 2

[Bör05] Egon Börger. Abstract state machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 2005. 20

[CF98] L. Colson and D. Fredholm. System t, call-by-value and the minimum problem. *Theoretical Computer Science*, 206:301–315, 1998. 2

[CL03] René Cori and Daniel Lascar. *Logique mathématique – Calcul propositionnel, algèbre de Boole, calcul des prédicats*, volume 1. Dunod, Masson edition, 2003. 4

[Col89] L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 372:194–206, 1989. 2

[Col96] L. Colson. A unary representation result for system t. *Annals of Mathematics and Artificial Intelligence*, 16:385-403, 1996. 2

[DDG97] S. Dexter, P. Doyle, and Y. Gurevich. Gurevich abstract state machine and schonhage storage modification machines. *J. Universal Computer Science*, 3:279–303, 1997. 2

[DG08] N. Dershowitz and Y. Gurevich. A natural axiomatization of church's thesis. *Bulletin of symbolic logic*, 14(3):299–350, 2008. 2

[FZG10] Marie Ferbus-Zanda and Serge Grigorieff. Asm and operational algorithmic completeness of lambda calculus. *Fields of Logic and Computation*, 2010. 30

[GGV01] Uwe Glaesser, Yuri Gurevich, and Margus Veanes. Universal plug and play machine models. *Technical report MSR-TR-2001-59 Microsoft Research*, 2001. 5

[Gur93] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993. 2

[Gur00a] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000. 2

[Gur00b] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 2000. 5, 7, 8, 9

[GV10] Serge Grigorieff and Pierre Valarcher. Evolving multialgebras unify all usual sequential computation models. *Symposium on Theoretical Aspects of Computer Science*, pages 417–428, 2010. 2

[GV12] Serge Grigorieff and Pierre Valarcher. Classes of algorithms: Formalization and comparison. *Bulletin of the EATCS*, (107), 2012. 21

[Kri07] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order and Symbolic Computation*, (20):199–207, 2007. 12

[Mar14] Yoann Marquer. Algorithmic completeness of imperative programming languages. *Fundamenta Informaticae*, submitted, 2014. 2, 14

[Mar15] Yoann Marquer. Algorithmic completeness of imperative programming languages. *Fundamenta Informaticae*, 2015. En revue. 34

[Mos01] Y. N. Moschovakis. What is an algorithm ? In Springer, editor, *Mathematics unlimited – 2001 and beyond*, pages 919–936. B. Engquist and W. Schmid, 2001. 2

[Mos03] Y. N. Moschovakis. On primitive recursive algorithms and the greatest common divisor function. *Theor. Comput. Sci.*, 301:1–30, 2003. 2

[MR67] Albert Meyer and Dennis Ritchie. The complexity of loop programs. *ACM national conference*, (22):465–469, 1967. 12, 15

[MV09] D. Michel and P. Valarcher. A total functional programming language that computes apra. In *Studies in Weak Arithmetic*, volume 196 of *CSLI Lecture Notes*. Stanford, 2009. 2

[Nee03] Peter Mølle Neergaard. Ploop: A language for polynomial time. Technical report, 2003. 15, 16

[Nig05] Karl-Heinz Niggl. Control structures in programs and computational complexity. *Ann. Pure Appl. Logic*, 133(1-3):247–273, 2005. 2, 3

[vdD03] Lou van den Dries. Generating the greatest common divisor, and limitations of primitive recursive algorithms. *Foundations of Computational Mathematics*, 3(3):297–324, 2003. 2