Algorithmic Completeness
of Imperative Programming Languages

Yoann Marquer∗†

Université Paris-Est Créteil (UPEC)
Laboratoire d’Algorithmique, Complexité et Logique (LACL)
IUT Sénart-Fontainebleau, France
marquer.yoann@hotmail.fr

Abstract. According to the Church-Turing Thesis, effectively calculable functions are functions computable by a Turing machine. Models that compute these functions are called Turing-complete. For example, we know that common imperative languages (such as C, Ada or Python) are Turing complete (up to unbounded memory).

Algorithmic completeness is a stronger notion than Turing-completeness. It focuses not only on the input-output behavior of the computation but more importantly on the step-by-step behavior. Moreover, the issue is not limited to partial recursive functions, it applies to any set of functions. A model could compute all the desired functions, but some algorithms (ways to compute these functions) could be missing (see [10, 27] for examples related to primitive recursive algorithms).

This paper’s purpose is to prove that common imperative languages are not only Turing-complete but also algorithmically complete, by using the axiomatic definition of the Gurevich’s Thesis and a fair bisimulation between the Abstract State Machines of Gurevich (defined in [16]) and a version of Jones’ while programs. No special knowledge is assumed, because all relevant material will be explained from scratch.

Keywords: Algorithm, ASM, Completeness, Computability, Imperative, Simulation.

∗This work is partially supported by the french program ANR 12 BS02 007 01.
†Address for correspondence: Université Paris-Est Créteil (UPEC), Laboratoire d’Algorithmique, Complexité et Logique (LACL), IUT Sénart-Fontainebleau, France.

Received November 2014; revised April 2019.
1. Introduction

The notion of “algorithm” is currently a work in progress in the theoretical computer science commu-
nity. Most of the ancient algorithms\footnote{Such as the famous Euclidean algorithm, or the Babylonian method for approximating square roots.} were sequential algorithms, but nowadays there are also parallel, 
distributed, real-time, bio-inspired or quantum algorithms. Although this paper focuses only on se-
quential algorithms (on Section 1), there is still no consensus on a formal definition of algorithms as 
there is for functions or programs.

To formalize the sequential algorithms we followed the axiomatization\footnote{There are other formalizations of the notion of algorithm, such as the equivalence class of Yanofsky in [30, 31] (criticized in [6] by Gurevich) and the recursors of Moschovakis in [26] (also criticized in [4] by Gurevich).} of Yuri Gurevich, based 
on the three postulates of sequential time, abstract states and bounded exploration. He proved in 
[16] that his axiomatic approach is identical to his Abstract State Machines\footnote{Gurevich implemented his ASMs via the programming language AsmL (see [9] for a comparison).}, which can be seen as if cond then actions commands like in [2]. Moreover, the ASMs are closer to the imperative frame-
work, therefore they seemed more appropriate for our purpose, which is to characterize algorithmic 
classes by using imperative programming languages.

We know that an imperative language such as Albert Meyer and Dennis Ritchie’s Loop defined 
in [25] can compute any primitive recursive function, but cannot compute some “better” algorithms\footnote{But in general the “best” algorithm computing a desired function may not exist. Blum proved in [7] that there exists a total recursive function $f$ such that for every machine $M_i$ computing $f$ there exists a machine $M_j$ computing $f$ exponentially faster for almost inputs.} (see [10] for the \texttt{min} and [27] for the \texttt{gcd}). This Loop language has been extended in [1] with an exit 
command to obtain every Arithmetical Primitive Recursive Algorithm\footnote{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.}. This work inspired us in our 
attempt to define an algorithmically complete imperative programming language.

An imperative language such as Jones’ \texttt{While} defined in [19] is Turing-complete, which means 
that it can simulate the input-output relation of a Turing machine (see [29]). In other words (according 
to the Church Thesis) it can compute (see [28]) every function calculable by a human using pen and 
paper. But this is only functional completeness. For example, a one-tape Turing machine can simulate 
the results of a two-tape Turing machine, but the palindrome recognition can be done in $O(n)$ steps 
with a two-tape Turing machine while requiring at least (see [3]) $O(n^2/\log(n))$ steps with a one-tape 
Turing machine.

So algorithmic completeness is not an input-output behavior but a step-by-step behavior.

We will define at Section 1 that a model is algorithmically complete if for a given initial state it 
can compute the same execution as a given algorithm. According to the Gurevich Thesis, it means 
computing the same execution as a given ASM. We will formalize the imperative programming lan-
guage \texttt{While} at Section 2 by using only sequences, updates, conditionals and (unbounded) conditional 
loops. We will prove at Section 3 our main theorem:

\textbf{Theorem 1.1.} \texttt{While} and ASM can fairly simulate each other.
We will define the notion of “fair simulation” at the end of Section 1. Notice that the While language of our paper is different from Jones’ language not because of its control structures (we consider sequences, updates, if and while commands as the core features of “real” imperative programming languages) but because of its data structures. Jones’ language only uses lists, but we wanted to study the algorithmic completeness with respect to the oracular nature of algorithms. Therefore, our theorem is not about data structures (which frees us from any particular technological implementation) but about control structures: we will prove that sequences, updates, if and while commands are sufficient to simulate every sequential algorithm by using available data structures.

2. Sequential algorithms (Algo)

In [16] 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 the “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.

2.1. Three postulates

Postulate 1. (Sequential Time)
A sequential algorithm $A$ is given by:
- a set of states $S(A)$
- a set of initial states $I(A) \subseteq S(A)$
- a transition function $\tau_A : S(A) \to S(A)$

Remark 2.1. According to this postulate, two sequential algorithms $A$ and $B$ are the same (see [6]) 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 $\vec{X} = X_0, X_1, X_2, ...$ such that:
- $X_0$ is an initial state
- for every $i \in \mathbb{N}$, $X_{i+1} = \tau_A(X_i)$

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

$$\text{time}(A, X) = \text{def} \ min \{ i \in \mathbb{N} ; \tau_A^i(X) = \tau_A^{i+1}(X) \}$$

6By “oracular nature” we mean that every algorithm is written using a set of static functions considered as oracles. For example, moving the head, reading the scanned symbol and changing the state are static operations given for free in Turing machines.

7In the definition of time, $f^i$ is the iteration of $f$ defined by $f^0 = \text{id}$ and $f^{i+1} = f(f^i)$. 
Remark 2.2. 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:

- A language $\mathcal{L}_X$
- A universe (or base set) $\mathcal{U}_X$
- An interpretation $\pi^X : \mathcal{U}_X^k \to \mathcal{U}_X$ for every $k$-ary symbol $s \in \mathcal{L}_X$

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

This formalization can be seen as a representation of a computer data storage. For example, the interpretation $\pi^X$ 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$^9$. Moreover, since the representation of states should be independent from their concrete implementation (for example the name of the objects), isomorphic states will be considered as equivalent:

Postulate 2. (Abstract States)

- The states of an algorithm $A$ are first-order structures. The states of $A$ have the same (finite) language $\mathcal{L}_A$. The transition function $\tau_A$ preserves the universe of a state.
- $S(A)$ and $I(A)$ are closed under isomorphisms.
  
  Every isomorphism between $X$ and $Y$ is an isomorphism between $\tau_A(X)$ and $\tau_A(Y)$.

The symbols of $\mathcal{L}_A$ are distinguished between the dynamic symbols whose interpretation can change during an execution, and the static symbols. More specifically, the interpretation of the static symbols is fixed by the initial state.

Moreover we distinguish the constructors (for example $\text{true}$, $\text{false}$, 0, $S$, etc.) whose interpretation is uniform (up to isomorphism) for every initial state, from the parameters. The symbols depending only on the initial state are the dynamic symbols and the parameters, so we call them the inputs.

---

$^8$In our paper we use the notations from [11] for the interpretation of terms and the restriction of structures in order to have more concise expressions.

$^9$We will discuss in the conclusion a constructive second postulate for common data structures (integers, words, lists, arrays, and graphs) but this is not the point of this article.
The logical variables are not used in this paper: every term and every formula will be closed, and the formulas will be 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 $A$, $f \in L_A$ be a dynamic $k$-ary function symbol, $a_1, \ldots, a_k, b \in U_X$. $(f, a_1, \ldots, a_k)$ denotes a location of $X$ and $(f, a_1, \ldots, a_k, b)$ denotes an update on $X$ at the location $(f, a_1, \ldots, a_k)$.

If $u$ is an update then $X + u$ is a new structure of language $L_A$ and universe $U_X$ such that the interpretation of a function symbol $f \in L_A$ is:

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

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

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

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

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

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

During an execution, if more and more updates are done 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 [16] that if $(f, a_1, \ldots, a_k, b) \in \Delta(A, X)$ then $a_1, \ldots, a_k, b$ are interpretations in $X$ of terms in $T$. So, since $T$ is finite there exists a bounded number of $a_1, \ldots, a_k, b$ such that $(f, a_1, \ldots, a_k, b) \in \Delta(A, X)$. Moreover, since $L_A$ 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.

---

10For example, in a graph such that at every step a vertex becomes blue whenever one of its neighbors is already blue. We will give a more formal example p.6 with the parallel lambda-calculus.
2.2. 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, such as:

**Example 2.3. (The Lambda-Calculus)**

Syntax of the Programs:  
\[ t = \text{def } x \mid \lambda x.t \mid (t_1)t_2 \]

\[ \beta\text{-reduction: } (\lambda x.t_1)t_2 \rightarrow_{\beta} t_1[t_2/x] \]

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:

Call-by-Name Context:  
\[ Cn\{\cdot \} = \text{def } \cdot \mid Cn\{\cdot \}t \]

Transition Rule:  
\[ Cn\{(\lambda x.t_1)t_2\} \rightarrow_n Cn\{t_1[t_2/x]\} \]

This rule can be implemented in a machine:

Operational semantics:  
\[ t_1t_2 \star \pi \rhd_0 t_1 \star t_2, \pi \]
\[ \lambda x.t_1 \star t_2, \pi \rhd_1 t_1[t_2/x] \star \pi \]

These notations and this machine are directly taken from Krivine’s [21]. In this machine \( \pi \) is a stack of terms. The symbol \( \star \) is a separator between the current program and the current state of the memory. \( \rhd \) 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 \( \lambda x.t \star \emptyset \).

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:

\[ (t)t \rightarrow_p (t)t(t)t \rightarrow_p (t)t(t)t(t)t \rightarrow_p (t)(t)t(t)t(t)t(t) \rightarrow_p \ldots \]

Indeed, at the step \( i \) exactly \( 2^{i-1} \) \( \beta \)-reduction 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 [14] 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 2.4. (Temporary Variables)

In this example a programmer is trying to simulate a repeat $n \{ s \}$ command in an imperative programming language\textsuperscript{11} containing while commands. The well-known solution is to use a temporary variable $i$ in the new program:

$$\{ i := 0; \text{while } i < n \{ s; i := i + 1; \}; \}$$

This simulation is very natural, but a fresh variable $i$ is necessary. So, the language $L_1$ of the simulating program must be bigger than the language $L_2$ of the simulated program.

Notation 2.5. We follow the notation from [11], where $X|_{L_2}$ denotes the restriction of the $L_1$-structure $X$ to the language $L_2$. The language of $X|_{L_2}$ is $L_2$, its universe is the same than $X$, and every symbol $s \in L_2$ has the same interpretation in $X|_{L_2}$ than in $X$.

This notation is extended to a set of updates:

$$\Delta|_L = \text{def} \{(f, \vec{a}, b) \in \Delta ; f \in L\}$$

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 $L_1 \setminus L_2$ between the two languages 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 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\textsuperscript{12} only on the constructors\textsuperscript{13}. Because this initialization is independent (up to isomorphism) from the initial states, we call it a uniform initialization.

Example 2.6. (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 the Turing machines. But if we consider that every

\textsuperscript{11}We will define p.11 the precise syntax of the imperative programs.
\textsuperscript{12}Even the values of the fresh variables in the initial states can be irrelevant. See the program $P_1$ p.21 where the variables $\vec{v}$ are explicitly updated with the value of the terms $t$ before being read.
\textsuperscript{13}See the program $P_{19}$ p.16 where the boolean variable $b_p$ is initialized with $true$ and the others with $false$. 
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 $X_0, X_1, X_2, X_3, X_4, X_5, X_6, \ldots$ of $M_3$ every three steps (the unobserved states are in gray) then we will obtain an execution defined by $Y_i = X_{3 \times i}$, 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 $Y_0, \ldots, Y_m, Y_m, \ldots$ could have finished, but the simulating execution $X_0, \ldots, X_{md}, X_{md+1}, \ldots, X_{md+(d-1)}, X_{md}, X_{md+1}, \ldots$ may continue forever. So, an ending condition like $\text{time}(A, X) = d \times \text{time}(B, X) + e$ is necessary, and corresponds to the usual consideration for asymptotic time complexity.

**Definition 2.7.** (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 $d \in \mathbb{N}^*$ 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 $i \in \mathbb{N}$, $X_{d \times i} \mid \mathcal{L}(P_2) = Y_i$

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

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

**Remark 2.8.** The second condition $X_{d \times i} \mid \mathcal{L}(P_2) = Y_i$ implies for $i = 0$ that the initial states are the same, up to temporary variables.

3. **Models of computation**

In this section the Gurevich’s Abstract State Machines are defined, and we use his theorem $\text{Algo} = \text{ASM}$ to get a constructive (from an operational point of view) occurrence of the sequential algorithms. So, we will say that a model of computation $M$ is **algorithmically complete** if $M \simeq \text{ASM}$. For example, in [12], Marie Ferbus-Zanda and Serge Grigorieff proved that the lambda-calculus is algorithmically complete up to the oracular nature of the algorithms.
We use the same method in this paper. Using Jones’ While language as core for imperative languages, we will prove in the next section the algorithmic completeness of While. In order to do so, we will prove a bisimulation between ASM and While, using the same data structures in these two models of computation.

3.1. Abstract State Machines (ASM)

Without going into details, the Gurevich’s Abstract State Machines (ASM) require only the equality =, the constants true and false, the unary operation ¬ and the binary operations ∧.

**Definition 3.1.** (ASM programs)

\[
\Pi = \text{def} \ ft_1\ldots ft_k := t_0 |
\text{if } F \text{ then } \Pi_1 \text{ else } \Pi_2 \text{ endif} |
\text{par } \Pi_1\|\ldots\|\Pi_n \text{ endpar}
\]

where \( f \) is a dynamic \( k \)-ary function symbol, \( t_0, t_1, \ldots, t_k \) are closed terms, and \( F \) is a formula.

**Notation 3.2.** For \( n = 0 \) a par command is an empty program, so let skip be the command \( \text{par endpar} \). If the else part of an if is a skip we only write \( \text{if } F \text{ then } \Pi \text{ endif} \).

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

**Read(\( \Pi \))** is defined by induction on \( \Pi \):

\[
\text{Read}(ft_1\ldots ft_k := t_0) = \text{def} \ \{t_1, \ldots, t_k, t_0\}
\]
\[
\text{Read}(\text{if } F \text{ then } \Pi_1 \text{ else } \Pi_2 \text{ endif}) = \text{def} \ \{F\} \cup \text{Read}(\Pi_1) \cup \text{Read}(\Pi_2)
\]
\[
\text{Read}(\text{par } \Pi_1\|\ldots\|\Pi_n \text{ endpar}) = \text{def} \ \text{Read}(\Pi_1) \cup \cdots \cup \text{Read}(\Pi_n)
\]

**Write(\( \Pi \))** is defined by induction on \( \Pi \):

\[
\text{Write}(ft_1\ldots ft_k := t_0) = \text{def} \ \{ft_1\ldots ft_k\}
\]
\[
\text{Write}(\text{if } F \text{ then } \Pi_1 \text{ else } \Pi_2 \text{ endif}) = \text{def} \ \text{Write}(\Pi_1) \cup \text{Write}(\Pi_2)
\]
\[
\text{Write}(\text{par } \Pi_1\|\ldots\|\Pi_n \text{ endpar}) = \text{def} \ \text{Write}(\Pi_1) \cup \cdots \cup \text{Write}(\Pi_n)
\]

**Remark 3.3.** The exploration witness of \( \Pi \) 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.6, defining the syntax of the programs is not enough to obtain a model of computation, we still have to define their semantics. An ASM program \( \Pi \) determines a transition function \( \tau_\Pi(X) = X + \Delta(\Pi, X) \), where the set of updates \( \Delta(\Pi, X) \) done by \( \Pi \) on \( X \) is defined by induction:
Definition 3.4. (Operational Semantics of the ASMs)

\[
\Delta(t_1 \ldots t_k := t_0, X) = \text{def}\ \{(f, t_1^X, \ldots, t_k^X, t_0^X)\}
\]

\[
\Delta(\text{if } F \text{ then } \Pi_1 \text{ else } \Pi_2 \text{ endif}, X) = \text{def} \ \Delta(\Pi_i, X)
\]

where \(i = 1\) if \(F\) is true on \(X\)

and \(i = 2\) if \(F\) is false on \(X\)

\[
\Delta(\text{par } \Pi_1 || \ldots || \Pi_n \text{ endpar}, X) = \text{def} \ \Delta(\Pi_1, X) \cup \cdots \cup \Delta(\Pi_n, X)
\]

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

Remark 3.5. For every states \(X\) and \(Y\), if the terms of \(\text{Read}(\Pi)\) have the same interpretation on \(X\) and \(Y\) then \(\Delta(\Pi, X) = \Delta(\Pi, Y)\).

We can now define the set \(\text{ASM}\) of Abstract States Machines:

Definition 3.6. An Abstract State Machine \(M\) with language \(L\) is given by:

- an ASM program \(\Pi\) on \(L\)
- a set \(S(M)\) of \(L\)-structures closed by isomorphisms and \(\tau_{\Pi}\)
- a subset \(I(M) \subseteq S(M)\) closed by isomorphisms
- an application \(\tau_M\), which is the restriction of \(\tau_{\Pi}\) to \(S(M)\)

For every sequential algorithm \(A\), the finiteness of the exploration witness in the third postulate allows us (see [16]) to write a finite \(\text{ASM}\) program \(\Pi_A\), which has the same set of updates than \(A\) for every state. These programs \(\Pi_A\) have the same normal form:

\[
\text{par } \text{if } F_1 \text{ then } \Pi_1 \text{ endif}
\]
\[
\ || \ \text{if } F_2 \text{ then } \Pi_2 \text{ endif}
\]
\[
\vdots
\]
\[
\ || \ \text{if } F_c \text{ then } \Pi_c \text{ endif}
\]
\text{endpar}

where \(F_i\) are “guards”, which means that for every state \(X\) one and only one \(F_i\) is \textit{true}, and the programs \(\Pi_i\) have the form \(\text{par } u_1 || \ldots || u_m \text{ endpar}\), where \(u_1, \ldots, u_m\) are update commands.

Remark 3.7. \(\Delta(\Pi_A, X) = \Delta(A, X) = \tau_A(X) - X\), 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 3.8. (Gurevich, 2000)

\[
\text{Algo} = \text{ASM}
\]

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

Remark 3.9. 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.

3.2. Imperative programming (While)

We use a variant of Neil Jones' While (see [19]) language, because this language is minimal. The programs are only sequences of updates, if or while commands. So, if While is algorithmically complete then every imperative language containing these control structures (including common programming languages such as C, Java or Python) will be algorithmically complete too.

The difference with Neil Jones' While is that the data structures are not fixed. As is the case for the ASMs, the equality and the booleans are needed, 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 ASM and While is proven for control structures, up to data structures.

Definition 3.10. (Syntax of the While programs)

\[
\begin{align*}
\text{(commands)} \quad c &= \text{def} \ f t_1 \ldots t_k := t_0 \\
&\quad | \text{if } F \{s_1\} \text{ else } \{s_2\} \\
&\quad | \text{while } F \{c; s\}
\end{align*}
\]

\[
\begin{align*}
\text{(sequences)} \quad s &= \text{def} \ \epsilon | c; s
\end{align*}
\]

\[
\begin{align*}
\text{(programs)} \quad P &= \text{def} \ \{s\}
\end{align*}
\]

where \( f \) is a dynamic \( k \)-ary function symbol, \( t_0, t_1, \ldots, t_k \) are closed terms, and \( F \) is a formula.

Notation 3.11. The symbol \( \epsilon \) denotes the empty sequence. For the sake of simplicity, the empty program will be written \( \{\} \) instead of \( \{\epsilon\} \). As is the case for the ASM programs, if the else part of an if is empty then we will only write \( \text{if } F \{s_1\} \). Let \( \text{skip} \) be the command \( \text{if } true \ \{\} \), which changes nothing but costs one step.

The sequence \( c; s \) of commands can be generalized by induction to a sequence of sequences \( s_1; s_2 \) by \( \epsilon; s_2 = s_2 \) and \( (c; s_1); s_2 = c; (s_1; s_2) \).

As seen in example 2.3 p.6 the operational semantics of this While programming language is formalized by a state transition system. A state of the system is a pair \( P \star X \) of a While program and a structure. Its transitions are determined only by the head command and the current structure:
Definition 3.12. Operational semantics of the While programs:

\[
\{ft_1...t_k := t_0; s\} \ast X \preceq \{s\} \ast X + (f, t_1^X, \ldots, t_k^X, t_0^X)
\]

\[
\begin{align*}
\{\text{if } F \{s_1\} \text{ else } \{s_2\}; s_3\} & \ast X \succ \{s_1; s_3\} \ast X \text{ if } F^X = \text{true} \\
\{\text{if } F \{s_1\} \text{ else } \{s_2; s_3\} \ast X \succ \{s_2; s_3\} \ast X \text{ if } F^X = \text{false} \\
\{\text{while } F \{s_1\}; s_2\} \ast X \succ \{s_1; \text{while } F \{s_1\}; s_2\} \ast X \text{ if } F^X = \text{true} \\
\{\text{while } F \{s_1\}; s_2\} \ast X \succ \{s_2\} \ast X \text{ if } F^X = \text{false}
\end{align*}
\]

The successors are unique, so this transition system is deterministic. We denote by \(\succ_i\) a succession of \(i\) transition steps, which can be defined by induction on \(i\).

Remark 3.13. If \(P_1 \ast X \succ_i P_2 \ast X\) and \(P_2 \ast X \succ_j P_3 \ast X\) then \(P_1 \ast X \succ_{i+j} P_3 \ast X\), so in a sense \(\succ_i\) is a transitive relation.

Only the states \(\{\} \ast X\) have no successor, so they are the terminating states.

We could have introduced a rule \(\{\} \ast X \succ \{\} \ast X\) and defined the termination like Gurevich did for Algo: \(P \ast X\) is terminal if \(P \ast X \succ P \ast X\). But if \(F^X = \text{true}\) then \(\{\text{while } F \{\}; s\} \ast X \succ \{\text{while } F \{\}; s\} \ast X\). So it should be seen as a terminal state too, which seems weird.

Because this problem occurs in the following simulations, we forbade in definition 3.10 the commands \(\text{while } F \{\}\). As a consequence, if \(P_1 \ast X \succ P_2 \ast X\) then \(P_1 \neq P_2\).

Notation 3.14. \(P\) terminates on \(X\), denoted by \(P \downarrow X\), if there exists \(i\) and \(X'\) such that:

\(P \ast X \succ_i \{\} \ast X'\)

Because the transition system is deterministic, \(i\) and \(X'\) are unique. So \(X'\) is denoted \(P(X)\) and \(i\) is denoted time\((P, X)\). A program is terminal if it terminates for every initial state.

Example 3.15. 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 \(x\):

\[P_{\text{min}} = \{x := 0; \text{while } \neg(x = m \lor x = n) \{x := x + 1; \}; \}
\]

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

\[
\begin{align*}
\{x := 0; \text{while } \neg(x = 2 \lor x = 3) \{x := x + 1; \}; \} \ast X \\
\succ \{\text{while } \neg(x = 2 \lor x = 3) \{x := x + 1; \}; \} \ast X + (x, 0) \\
\succ \{x := x + 1; \text{while } \neg(x = 2 \lor x = 3) \{x := x + 1; \}; \} \ast X + (x, 0) \\
\succ \{x := x + 1; \text{while } \neg(x = 2 \lor x = 3) \{x := x + 1; \}; \} \ast X + (x, 1) \\
\succ \{x := x + 1; \text{while } \neg(x = 2 \lor x = 3) \{x := x + 1; \}; \} \ast X + (x, 2) \\
\succ \{\} \ast X + (x, 2)
\end{align*}
\]

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

Notation 3.16. Let \(s_P\) be the sequence such that \(P = \{s_P\}\). The composition \(P_1 P_2\) of the imperative programs \(P_1\) and \(P_2\) is defined by \(P_1 P_2 = \{s_{P_1}; s_{P_2}\}\).
while $F \, P_1 \, P_2$ can be read as while $F \{s_{P_1}; s_{P_2}\}$ or while $F \{s_{P_1}; s_{P_2}\}$. In order to avoid this ambiguity, braces will be added when necessary.

It can be convenient to consider a command $c$ as a program, so let $\{c;\}$ be the program $c$. In particular, $cP$ is a notation for the program $\{c; s_P\}$.

We can now prove that the composition of programs behaves as intended. It can be done by using only the determinism and the transitivity of the transition system. The proof is admitted in this paper, but can be checked in the longer version [23].

**Proposition 3.17.** (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 $P_1 P_2(X) = P_2(P_1(X))$ and $time(P_1 P_2, X) = time(P_1, X) + time(P_2, P_1(X))$.

As a consequence, we can prove by induction that every imperative program using only sequences of updates and conditionals terminates for every initial state:

**Corollary 3.18.** (Termination of Programs without while)

If the imperative program $P$ has no while command then $P$ is terminal.

Because the transition system is deterministic, if $i \leq time(P, X)$\(^{14}\) then there exists a unique $P'$ and $X'$ such that $P \times X \succ_i P' \times X'$. Let $\tau^i_X(P)$ be that $P'$ and $\tau^i_P(X)$ be that $X'$, so:

$$P \times X \succ_i \tau^i_X(P) \times \tau^i_P(X)$$

If $i > time(P, X)$ we can assume that $\tau^i_X(P) = \emptyset$ and $\tau^i_P(X) = P(X)$.

**Remark 3.19.** $\tau^i_P$ is not a transition function in the sense of the first section, because:

$$\tau^i_P(X) \neq \tau_P \circ \cdots \circ \tau_P(X)$$

Indeed, if $P_0 \times X_0 \succ P_1 \times X_1 \succ \cdots \succ P_{i-1} \times X_{i-1} \succ P_i \times X_i$ then:

$$\tau^i_{P_0}(X_0) = X_i = \tau_{P_{i-1}}(X_{i-1}) = \cdots = \tau_{P_{i-1}} \circ \cdots \circ \tau_{P_1} \circ \tau_{P_0}(X_0) \neq \tau_{P_0} \circ \cdots \circ \tau_{P_0}(X_0)$$

The succession of updates made by $P$ on $X$ is $\tau^1_P(X) - \tau^0_P(X)$, then $\tau^2_P(X) - \tau^1_P(X)$, then... In our transition system a structure is updated only with an update command and only one update per update command. Therefore, $\tau^{i+1}_P(X) - \tau^i_P(X)$ is empty or is a singleton.

**Definition 3.20.** The set of updates made by $P$ on $X$ is:

$$\Delta(P, X) = def \bigcup_{i \in \mathbb{N}} \tau^{i+1}_P(X) - \tau^i_P(X)$$

**Remark 3.21.** If $P$ terminates on $X$ then the cardinal of $\Delta(P, X)$ is bounded by $time(P, X)$.

\(^{14}\)If $P$ does not terminate on $X$, we could assume that $time(P, X) = \infty$. 
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 $(f,\bar{a},b)$ and $(f,\bar{a},b')$ 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.

**Proposition 3.22.** (Updates of a Non-Overwriting Program)

If $P$ terminates on $X$ without overwrite then $\Delta(P,X) = P(X) - X$.

**Proof:**
The proof is admitted in this paper, but is detailed in the longer version [23].

4. **Algorithmic completeness**

We want to prove that While is algorithmically complete. According to Gurevich’s theorem p.11, this would mean that $\text{While} \simeq \text{ASM}$. Considering our definition p.8, we will have to prove that ASM fairly simulates While, and that While fairly simulates ASM.

4.1. **ASM simulates While**

The intuitive idea for translating While 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\textsuperscript{15}.

**Example 4.1.** The imperative program $P_{min}$ of the example 3.15 p.12:

\begin{align*}
0 & : \ x := 0 \\
1 & : \ \textbf{while} \ \neg(x = m \vee x = n) \\
2 & : \ x := x + 1
\end{align*}

could be translated into the following ASM program:

\begin{verbatim}
par if line = 0 then
  par x := 0 || line := 1 endpar
endif
|| if line = 1 then
  if \neg(x = m \vee x = n) then line := 2 else line := 3 endif
endif
|| if line = 2 then
  par x := x + 1 || line := 1 endpar
endif
endpar
\end{verbatim}

\textsuperscript{15}Programs of this form are called control state ASMs (see [8]).
Remark 4.2. The number of a line is between 0 and length\((P)\). So, a finite number of booleans \(b_0, b_1, \ldots, b_{\text{length}(P)}\) can be used\(^{16}\) instead of an integer line.

This approach has been suggested in [15], and is fitted for a line-based programming language (for example with \texttt{goto} instructions) but not the structured language \texttt{While}. Indeed, the positions in the program can distinguish two commands even if they are identical for the operational semantics of \texttt{While}:

Example 4.3. (Marked \texttt{While})

To make an easy example, let’s compare the two updates \(x := x + 1\) in the program:

\[
P = \{\texttt{while true \{x := x + 1; \texttt{while true \{x := x + 1;}\}};};\}
\]

Because their position is not the same in the program they have a different number of line. So, we mark them with boxes \(\boxed{x := x + 1}\) and \(\boxed{x := x + 1}\) to distinguish each one from the other. The conditionals are always \texttt{true}, so we can ignore the structures and focus only on the evolution of the program:

\[
\begin{array}{cccc}
P_1 & > & P_2 & > \cdots \\
\texttt{while true} & \boxed{x := x + 1} & \texttt{while true} & \boxed{x := x + 1} \\
\texttt{while true} & \boxed{x := x + 1} & \texttt{while true} & \boxed{x := x + 1} \\
\texttt{while true} & \boxed{x := x + 1} & \texttt{while true} & \boxed{x := x + 1} \\
\texttt{while true} & \boxed{x := x + 1} & \texttt{while true} & \boxed{x := x + 1} \\
\texttt{while true} & \boxed{x := x + 1} & \texttt{while true} & \boxed{x := x + 1} \\
\end{array}
\]

We could have replaced \(P_2\) by \(P_1\) in this execution without changing anything except the boxes. \(\boxed{x := x + 1}\) and \(\boxed{x := x + 1}\) are identical for the operational semantics, so we should find another way to keep track of the current command.

We will not use booleans \(b_0, b_1, \ldots, b_{\text{length}(P)}\) 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 4.4. (The Graph of Execution of \(P_{\text{min}}\))

\[
\begin{align*}
x := 0 & \quad \text{while } \neg(x = m \lor x = n) \\
x := x + 1 & \quad \text{while } \neg(x = m \lor x = n) \\
\downarrow & \quad \text{while } \neg(x = m \lor x = n) \\
\{ & \\
\end{align*}
\]

\(^{16}\)Remember that booleans must be in the data structure, but integers may not.
In the following only the vertices of the graph are needed, so the graph of execution of $P_{\text{min}}$ will be denoted by the set of possible programs:

$$
G(P_{\text{min}}) = \{ \{ x := 0; \text{while } \neg(x = m \lor x = n) \{ x := x + 1; \}; \}, \\
\{ \text{while } \neg(x = m \lor x = n) \{ x := x + 1; \}; \}, \\
\{ x := x + 1; \text{while } \neg(x = m \lor x = n) \{ x := x + 1; \}; \}, \\
\{ \} \}
$$

**Notation 4.5.** In order to define graphs of execution we need to introduce the notation:

$$GP = \{ P_G; P_G \in G \}$$

where $G$ is a set of imperative programs and $P$ is an imperative program.

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

**Definition 4.6.** (Graph of Execution)

$$
G(\{\}) = \{\}\ 
G(cP) = \{G(c)P \cup G(P) \}
$$

$$
G(ft_1 \ldots t_k := t_0) = \{\{ft_1 \ldots t_k := t_0;\}\}
$$

$$
G(\text{if } F \text{ then } \{P_1\} \text{ else } \{P_2\}) = \{[\text{if } F \text{ then } \{P_1\} \text{ else } \{P_2\}];\} \cup G(P_1) \cup G(P_2)
$$

$$
G(\text{while } F \{P_1\}) = \{G(P_1) \{\text{while } F \{P_1\};\}\}
$$

As intended, we can prove (see [23]) that $\text{card}(G(P)) \leq \text{length}(P) + 1$. So, only a finite number of guards depending only on $P$ are necessary. Notice that for some programs (like $P_{\text{min}}$ in example 4.4 p.15) which do not follow example 4.3 p.15, $\text{card}(G(P)) = \text{length}(P) + 1$ can be reached, so the bound is optimal.

Again, to focus on the simulation, we admit in this paper the proof (see [23]) stating that a graph of execution is closed for the operational semantics of the imperative programs:

**Proposition 4.7.** (Operational Closure of Graph of Execution)

If $ft_1 \ldots t_k := t_0$ $P' \in G(P)$ then $P' \in G(P)$

If $\text{if } F \text{ then } \{P_1\} \text{ else } \{P_2\}$ $P' \in G(P)$ then $P_1P'$ and $P_2P' \in G(P)$

If $\text{while } F \{P_1\}$ $P' \in G(P)$ then $P_1$ $\text{while } F \{P_1\} P'$ and $P' \in G(P)$

**Notation 4.8.** The fresh boolean variables will be denoted $b_{P_G}$ where $P_G \in G(P)$. Only one $b_{P_G}$ will be true for each step of an execution, so in the following we will write $X[b_{P_i}]$ if $b_{P_i}$ is true and the other $b_{P_j}$ are false, where $X$ denotes a $\mathcal{L}_P$-structure. Notice that $X[b_{P_i}]{\mathcal{L}_P} = X$. 


The proposition 4.7 ensures that the following translation is well-defined:

**Definition 4.9.** (Translation of imperative programs into ASM)

$$\Pi P = \text{def} \begin{array}{l} \text{par} \quad \text{if} \ b_{P_G} \ \text{then} \ P_G^{tr} \ \text{endpar} \end{array}$$

where $$P^{tr}$$ is defined by induction:

$$\{\}^{tr} = \text{def} \ \text{skip}$$

$$(ft_1 \ldots t_k := t_0 \ P)^{tr} = \text{def} \begin{array}{l} \text{par} \ b_{ft_1 \ldots t_k := t_0} \ P' := \text{false} \\ \| ft_1 \ldots t_k := t_0 \\ \| b_{P'} := \text{true} \end{array}$$

endpar

$$(\text{if } F \ \text{then } \{P_1\} \ \text{else } \{P_2\} \ P')^{tr} = \text{def} \begin{array}{l} \text{par} \ b_{\text{if } F \ \text{then } \{P_1\} \ \text{else } \{P_2\} \ P'} := \text{false} \\ \| \text{if } F \ \text{then } b_{P_1} \ P' := \text{true} \ \text{else } b_{P_2} \ P' := \text{true} \ \text{endif} \end{array}$$

endpar

$$(\text{while } F \ \{P_1\} \ P')^{tr} = \text{def} \begin{array}{l} \text{par} \ b_{\text{while } F \ \{P_1\} \ P'} := \text{false} \\ \| \text{if } F \ \text{then } b_{P_1} \ \text{while } F \ \{P_1\} \ P' := \text{true} \ \text{else } b_{P'} := \text{true} \ \text{endif} \end{array}$$

endpar

Notice that for every $$P_G \in \mathcal{G}(P)$$, $$\Delta(\Pi P, X[b_{P_G}]) = \Delta(P_G^{tr}, X[b_{P_G}])$$. We use this fact in [23] to prove by exhaustion on $$\tau^i_X(P)$$ that the translation of the imperative program $$P$$ behaves as intended:

**Proposition 4.10.** (Step-by-Step Simulation)

For every $$i < \text{time}(P, X)$$, $$\tau_{\Pi P}^i(X[b_{\tau^i_X(P)}]) = \tau_{P}^{i+1}(X)[b_{\tau^i_X(P)}]$$

**Theorem 4.11.** ASM fairly simulates While.

**Proof:**

We prove the three conditions of the fair simulation defined p.8:

1. $$\mathcal{L}_{\Pi P} = \mathcal{L}_P \cup \{b_{P_G} ; P_G \in \mathcal{G}(P)\}$$
   where $$\text{card}(\{b_{P_G} ; P_G \in \mathcal{G}(P)\}) \leq \text{length}(P) + 1$$.

2. Using proposition 4.10, we can prove by induction on $$i \leq \text{time}(P, X)$$ that:
   $$\tau_{\Pi P}^i(X[b_P]) = \tau_{P}^i(X)[b_{\tau^i_X(P)}]$$
   So $$\tau_{\Pi P}^i(X[b_P])|_{\mathcal{L}_P} = \tau_{P}^i(X)$$
   And the temporal dilation is $$d = 1$$.
3. If \( i = \text{time}(P, X) \) then \( \tau^i_{\Pi}(P) = \{\} \)

So \( \Delta(\Pi_P, \tau^i_{\Pi}(X[b_{r_{\Pi}(P)}])) = \emptyset \), and \( \tau^{i+1}_{\Pi_P}(X[b_P]) = \tau^i_{\Pi_P}(X[b_P]) \)

So \( \text{time}(\Pi_P, X[b_P]) \leq \text{time}(P, X) \) (1)

But remember (p.12) that if \( P_1 \star X_1 \succ P_2 \star X_2 \) then \( P_1 \neq P_2 \)

So for every \( i < \text{time}(P, X) \), \( b_{\tau^i_{\Pi}(P)} \) is updated, so \( \tau^{i+1}_{\Pi_P}(X[b_P]) \neq \tau^i_{\Pi_P}(X[b_P]) \)

So \( \text{time}(\Pi_P, X[b_P]) \geq \text{time}(P, X) \) (2)

Therefore, according to (1) and (2), \( \text{time}(\Pi_P, X[b_P]) = \text{time}(P, X) \), and \( e = 0 \).

\( \square \)

4.2. \( \text{While} \) simulates ASM

Let \( \Pi \) be an ASM program. The purpose of this section is to find a \( \text{While} \) program simulating the same executions than \( \Pi \). Remember that \( \Pi \) contains only updates, if and par commands. The intuitive solution is to translate the commands directly, without paying attention to the parallelism:

Definition 4.12. (Syntactical Translation of the ASM programs)

\[
\begin{align*}
(f_1 \ldots f_k := t_0)_\text{tr} &\triangleq \{ f_1 \ldots f_k := t_0; \} \\
(\text{if } F \text{ then } \Pi_1 \text{ else } \Pi_2 \text{ endif})_{tr} &\triangleq \{ \text{if } F \text{ then } \Pi_1_{tr} \text{ else } \Pi_2_{tr}; \} \\
(\text{par } \Pi_1 \| \ldots \| \Pi_n \text{ endpar})_{tr} &\triangleq \Pi_1_{tr} \ldots \Pi_n_{tr} \text{ (composition)}
\end{align*}
\]

Updates and if commands are the same in these two models of computation, but the simultaneous commands of ASM must be sequentialized in \( \text{While} \), so this translation does not respect the semantics of the ASM programs:

Example 4.13. Let \( X \) be a structure such that \( x^X = 0 \) and \( y^X = 1 \), and \( \Pi \) be the program:

\[
\Pi = \text{par } x := y \| y := x \text{ endpar}
\]

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

\[
\Pi_{tr} = \{ x := y; y := x; \}
\]

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

\[
\begin{align*}
\{x := y; y := x;\} \star X \\
\succ \{y := x;\} \star X + \{(x, 1)\} \\
\succ \{\} \star X + \{(x, 1), (y, 1)\}
\end{align*}
\]

So \( \tau_\Pi(X) = X + \{(x, 1), (y, 0)\} \neq X + \{(x, 1), (y, 1)\} = \Pi_{tr}(X) \).
To capture the simultaneous behavior of the ASM program, we need to store the values of the variables read in the imperative program. For example, if \( v = x \) and \( w = y \) in \( X \) then:

\[
\{ x := w; y := v; \} \ast X
\]

\[
\succ \{ y := v; \ast X + \{(x, 1)\}
\]

\[
\succ \{ \ast X + \{(x, 1), (y, 0)\}\}
\]

Indeed, even if \( x \) has been updated, its old value is still in \( v \).

**Definition 4.14.** (Substitution of a Term by a Variable)

\[
\{\} [v/t] \overset{\text{def}}{=} \{\}
\]

\[
(cP)[v/t] \overset{\text{def}}{=} c[v/t]P[v/t]
\]

\[
(ft_1 \ldots t_k := t_0)[v/t] \overset{\text{def}}{=} ft_1[v/t] \ldots t_k[v/t] := t_0[v/t]
\]

\[
(\text{if } F \text{ then } \{P_1\} \text{ else } \{P_2\})[v/t] \overset{\text{def}}{=} \text{if } F[v/t] \{P_1[v/t]\} \text{ else } \{P_2[v/t]\}
\]

\[
(\text{while } F \{P\})[v/t] \overset{\text{def}}{=} \text{while } F[v/t] \{P[v/t]\}
\]

where \( t_1[v/t_2] \overset{\text{def}}{=} \begin{cases} v & \text{if } t_1 = t_2 \\ t_1 & \text{else} \end{cases} \)

**Remark 4.15.** If \( P \) is an imperative program, \( t_1 \) and \( t_2 \) are distinct terms, and \( v_1 \) and \( v_2 \) are fresh distinct variables then \( P[v_1/t_1][v_2/t_2] = P[v_2/t_2][v_1/t_1] \). As a consequence, for \( k \) distinct terms \( t_1, t_2, \ldots, t_k \) and \( k \) fresh distinct variables \( v_1, v_2, \ldots, v_k \), the notation \( P[v_1/t_1][v_2/t_2] \) is not ambiguous, because the substitutions can be made in any order.

We defined p.9 the set \( \text{Read}(\Pi) \) of the terms read by \( \Pi \). Let \( r = \text{card}(\text{Read}(\Pi)) \), and \( t_1, \ldots, t_r \) be the distinct terms read by \( \Pi \). We substitute them by the fresh variables \( v_{t_1}, \ldots, v_{t_r} \), each one distinct from the other.

According to the Gurevich’s Theorem, every ASM is equivalent to an ASM in normal form, so we can assume that \( \Pi \) is in normal form (see p.11). Its translation \( \Pi^{tr}[\vec{v}/\vec{t}] \) has the form shown at figure 1 p.20. But 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.7), we have to update the variables \( \vec{v}_t \) explicitly at the beginning of the program with a sequence of updates:

   \[
v_{t_1} := t_1; \ldots; v_{t_r} := t_r;
   \]

2. The execution time is not constant.

   Because the ASM is in normal form, every \( F \) is a guard, which means that one and only one \( F_i \) is \textit{true} for the current state \( X \). The block of updates requires \( m_i \) steps to be computed by the imperative program, so the number of steps depends on the current state.
\[ \Pi = \begin{align*}
\text{par if } F_1 \text{ then} & \\
\quad & \text{par } f_1^1(\vec{t}_1) := t_1^1 \\
\quad & \quad || f_1^2(\vec{t}_2) := t_1^2 \\
\quad & \quad \vdots \\
\quad & \quad || f_{m_1}^1(\vec{t}_{m_1}) := t_{m_1}^1 \\
\text{endpar} \\
\text{endif}
\end{align*} \]

\[ \begin{align*}
\Pi^{tr}[\vec{v}/\vec{t}] = \begin{cases}
\text{if } v_{F_1} \text{ then } \{ & \\
\quad & f_1^1(\vec{v}_1) := v_{t_1}^1; \\
\quad & f_1^2(\vec{v}_2) := v_{t_2}^1; \\
\quad & \vdots \\
\quad & f_{m_1}^1(\vec{v}_{m_1}) := v_{t_{m_1}}^1; \\
\text{endif} \\
\text{endpar}
\end{cases} \]

\[ \text{endpar} \]

This is an issue because, according to our definition of the fair simulation p.8, every step of the ASM \( \Pi \) must be simulated by \( d \) steps, where \( d \) depends only on \( \Pi \). In order to obtain a uniform temporal dilation, we will add \( m - m_i \) \texttt{skip} commands\(^{17}\) at the end of each block, where \( m \) is defined by:

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

\(^{17}\)It may seem strange in an algorithmic purpose to lose time, but these \texttt{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.
$P_{\Pi} = \text{def } \{ \\
\quad v_{t_1} := t_1; \\
\quad v_{t_2} := t_2; \\
\quad \vdots \\
\quad v_{t_r} := t_r; \\
\quad \text{if } v_{F_1} \text{ then } \{ \\
\quad f^1_1(\vec{v}_{t_1}) := v_{t_1}; \\
\quad f^1_2(\vec{v}_{t_2}) := v_{t_2}; \\
\quad \vdots \\
\quad f^1_{m_1}(\vec{v}_{t_m_1}) := v_{t_{m_1}}; \\
\quad \text{skip;} \\
\quad \vdots (m - m_1 \text{ times}) \\
\quad \text{skip;} \\
\quad \}; \\
\quad \text{if } v_{F_2} \text{ then } \{ \\
\quad f^2_1(\vec{v}_{t_1}) := v_{t_1}; \\
\quad f^2_2(\vec{v}_{t_2}) := v_{t_2}; \\
\quad \vdots \\
\quad f^2_{m_2}(\vec{v}_{t_{m_2}}) := v_{t_{m_2}}; \\
\quad \text{skip;} \\
\quad \vdots (m - m_2 \text{ times}) \\
\quad \text{skip;} \\
\quad \}; \\
\quad \vdots \\
\quad \text{if } v_{F_c} \text{ then } \{ \\
\quad f^c_1(\vec{v}_{t_1}) := v_{t_1}; \\
\quad f^c_2(\vec{v}_{t_2}) := v_{t_2}; \\
\quad \vdots \\
\quad f^c_{m_c}(\vec{v}_{t_{m_c}}) := v_{t_{m_c}}; \\
\quad \text{skip;} \\
\quad \vdots (m - m_c \text{ times}) \\
\quad \text{skip;} \\
\quad \}; \\
\quad \} \\
\}

Figure 2. Translation $P_{\Pi}$ of one step of $\Pi$
We obtain at figure 2 p.21 the translation $P_{\Pi}$ of one step of the ASM program $\Pi$. Let $X$ be a state of the ASM with program $\Pi$, extended with the variables $\vec{v}_t$. As intended, we prove that $P_{\Pi}$ simulates one step of $\Pi$ in a constant time $t_{\Pi}$:

**Proposition 4.16.** (Semantical Translation of the ASM programs)

There exists $t_{\Pi}$, depending only on $\Pi$, such that for every state $X$ of $P_{\Pi}$:

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

**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 $\vec{v}$. So, in the state $Y$ after these updates, $\vec{v}_{t_k}^Y = \vec{t}_k^X$. Moreover, in the rest of the program the variables $\vec{v}_t$ are not updated, so for every following state $Y$, $\vec{v}_{t_k}^Y = \vec{t}_k^X$.

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

$i - 1$ 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 $Y$, $\vec{v}_{t_k}^Y = \vec{t}_k^X$, 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* commands require $m - m_i$ steps. So the commands in the block require $m_i + (m - m_i) = m$ steps, and the execution time depends only on $\Pi$:

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

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

$$\Delta(P_{\Pi}, X) = \{(v_{t_1}, \vec{t}_1^X), \ldots, (v_{t_r}, \vec{t}_r^X)\} \cup \Delta(\Pi, X|_{\mathcal{L}_{\Pi}})$$

$P_{\Pi}$ contains only updates and conditionals, so according to proposition 3.18 p.13 this program is terminal. Moreover, the fresh variables are updated only once, and since $\Pi$ is in normal form, $\Delta(\Pi, X|_{\mathcal{L}_{\Pi}})$ is consistent. So, $P_{\Pi}$ terminates without overwrite on $X$, and according to proposition 3.22 p.14:

$$\Delta(P_{\Pi}, X) = P_{\Pi}(X) - X$$

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

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

**Corollary 4.17.** $P_{\Pi}^i(X)|_{\mathcal{L}_{\Pi}} = \tau_{\Pi}^i(X|_{\mathcal{L}_{\Pi}})$
For any initial state, 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 the program the variables $\vec{v}_t$ have the old values of the terms. For example, if the initial state is $P_\Pi^i(X)$, after one execution of $P_\Pi$ we have:

$$\vec{v}_{t_k}^{P_\Pi^{i+1}(X)} = \vec{t}_k^{P_\Pi_i(X)}$$

This remark is particularly useful because it allows us to detect the end of the execution of $\Pi$. The program $P_\Pi$ simulates one step of $\Pi$ so we need to repeat it a sufficient number of times to simulate the full execution. So, a program like while $F \{P_\Pi\}$ is our candidate to fairly simulate $\Pi$, but we need to define an appropriate $F$.

$\Pi$ terminates when no more updates are done. In that case, the old values of the terms read by $\Pi$ 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:

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

We call it the “$\mu$-formula” because it is similar to the minimization operator $\mu$ from recursive functions (see [11]):

**Lemma 4.18.** (The $\mu$-formula)

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

**Proof:**

$\text{time}(\Pi, X|_{\ell_{\Pi}}) = \min \{ i \in \mathbb{N} \mid \tau_{\Pi}^i(X|_{\ell_{\Pi}}) = \tau_{\Pi}^{i+1}(X|_{\ell_{\Pi}}) \}$, so all that is left to prove (see [23]) is that $\tau_{\Pi}^i(X|_{\ell_{\Pi}}) = \tau_{\Pi}^{i+1}(X|_{\ell_{\Pi}})$ if and only if $F_{\Pi}^{P_\Pi^{i+1}(X)} = \text{true}$, by using the remark p.10 on Read($\Pi$). \qed

Notice that $F_{\Pi}$ becomes $\text{true}$ after $\text{time}(\Pi, X|_{\ell_{\Pi}}) + 1$ steps, so we execute the program $P_\Pi$ one more time after the end of $\Pi$. This is not an issue because we can estimate a bound for the ending time, as required by the third condition of the fair simulation.

Moreover, in the program while $\neg F_{\Pi} \{P_\Pi\}$ the variables $\vec{v}_t$ must be properly initialized to obtain a correct value for the $\mu$-formula $F_{\Pi}$. We do that simply by adding an occurrence of $P_\Pi$ at the beginning of the program. So, in a sense, the correct control structure should be a do while and not a while.

**Theorem 4.19.** While fairly simulates ASM.

**Proof:**

We prove that the imperative program simulating $\Pi$ is $P = P_\Pi \text{ while } \neg F_{\Pi} \{P_\Pi\}$.

1. There are $r = \text{card}(\text{Read}(\Pi))$ fresh variables $\vec{v}_t$. 

2. According to lemma 4.18 p.23, the execution of this program on X is:

\[
P_\Pi \text{ while } \neg F_\Pi \{P_\Pi\} \ast X \succ_{\Pi} \text{ while } \neg F_\Pi \{P_\Pi\} \ast P_\Pi(X)
\]

\[
\succ_{\Pi} \text{ while } \neg F_\Pi \{P_\Pi\} \ast P_\Pi^2(X)
\]

\[
\vdots
\]

\[
\succ_{\Pi} \text{ while } \neg F_\Pi \{P_\Pi\} \ast P_\Pi^{\text{time}(\Pi,X|\mathcal{L}_\Pi)}(X)
\]

\[
\succ_{\Pi} \text{ while } \neg F_\Pi \{P_\Pi\} \ast P_\Pi^{\text{time}(\Pi,X|\mathcal{L}_\Pi)+1}(X)
\]

\[
\succ \{\} \ast P_\Pi^{\text{time}(\Pi,X|\mathcal{L}_\Pi)+1}(X)
\]

So, for every \(0 \leq i \leq \text{time}(\Pi,X|\mathcal{L}_\Pi) + 1\), \(\tau_P^{d \times i}(X) = P_\Pi^i(X)\), where \(d = t_\Pi + 1\).

But, according to corollary 4.17 p.22, \(P_\Pi^i(X)|\mathcal{L}_\Pi = \tau_\Pi^i(X|\mathcal{L}_\Pi)\), so:

\[
\tau_P^{d \times i}(X)|\mathcal{L}_\Pi = \tau_\Pi^i(X|\mathcal{L}_\Pi)
\]

3. \(\text{time}(P,X) = (t_\Pi + 1) \times (\text{time}(\Pi,X|\mathcal{L}_\Pi) + 1)\)

\[
= d \times \text{time}(\Pi,X|\mathcal{L}_\Pi) + e
\]

where \(e = t_\Pi + 1\).

According to theorem 4.11, every program in \texttt{while} can be simulated by an ASM \(\Pi\), which itself can be simulated by \(P_\Pi\) while \(\neg F_\Pi\{P_\Pi\}\). So, an interesting corollary of theorem 4.19 is that every program in \texttt{while} is equivalent to a program using only one \texttt{while} command. This result can be seen as an algorithmic Kleene’s Normal Form Theorem (see [20]).

\section*{Conclusion}

We have proven the two sides of the simulation p.17 and p.23. Therefore, according to our definition p.8, \(\texttt{while} \simeq \texttt{ASM}\). Consequently, imperative programming languages are algorithmically complete, up to data structures.

The cost in space of the simulation is \(O(\text{length})\) in both cases. Indeed, an ASM requires \(\leq \text{length}(P) + 1\) fresh variables to simulate an imperative program \(P\), and an imperative program requires \(\text{card(Read(}\Pi)) \leq (k + 1) \times \text{length}(\Pi)\) fresh variables to simulate an ASM \(\Pi\), where \(k\) is the maximal arity of the dynamic symbols of \(\Pi\).

But the cost in time is not the same. Indeed, an ASM requires a temporal dilation of \(d = 1\) to simulate an imperative program, but an imperative program requires \(d = r + c + m + 1\) steps to simulate one step of an ASM \(\Pi\), where \(r\) is the number of terms read by \(\Pi\), \(c\) is the number of conditionals of \(\Pi\), and \(m\) is the maximal number of updates per block of \(\Pi\).
So, in an Orwellian sense, ASM is “more equivalent” than While, because they are algorithmically equivalent but ASM seems stronger.

This is because, contrary to ASM, only one update can be done per step of computation in While. Moreover, the exploration of control structures is free in ASM but not in While. In all fairness, we can imagine a stronger While with tuples of updates and free exploration of the control structures:

\[(f_1 \vec{t}_1, \ldots, f_k \vec{t}_k) := (t_1, \ldots, t_k); s \} \star X \succsim_1 \{ s \} \star X + \{(f_1 \vec{t}^X_1, t_1^X), \ldots, (f_k \vec{t}^X_k, t_k^X)\}\]

\[\{ \text{if } F \{ s_1 \} \text{ else } \{ s_2 \}; s_3 \} \star X \succsim_0 \{ s_i; s_3 \} \star X\]

where \(i = \begin{cases} 1 & \text{if } F^X = \text{true} \\ 2 & \text{if } F^X = \text{false} \end{cases}\)

\[\{ \text{while } F \{ s_1 \}; s_2 \} \star X \succsim_0 \{ s; s_2 \} \star X\]

where \(s = \begin{cases} s_1; \text{while } F \{ s_1 \}; \text{if } F^X = \text{true} \\ \epsilon \text{ if } F^X = \text{false} \end{cases}\)

The simulation between this stronger While and ASM is not only step-by-step, but strictly step-by-step (the temporal dilation is \(d = 1\)). But this model for imperative programming language is not common, and the theorem is stronger with a minimal core for imperative behavior.

So, the algorithmic difference between ASM and While does not really lie on control structures, but on data structures. According to the second postulate these data structures are first-order structures, which can hardly be seen as “real” data structures. The remaining issues are:

1. Data Structures
   Is it possible to fairly represent any data structure from any model of computation as first-order structures? In other words, is it possible to prove a constructive second postulate? We tried to characterize common data types (such as integers, words, lists, arrays, and graphs) in [22], but the general problem remains open.

2. Implicit Complexity
   What is the “size” of an element, or the “cost” of an operation? How can we characterize classes of algorithms depending on the size of the inputs and the available operations? More specifically, is it possible to characterize relevant classes of algorithms in an imperative framework? We proved in [22] that LoopC with PR data structures characterizes PR-time algorithms, and in [24] that PLoopC characterizes P-time algorithms, but the problem remains open for other classes such as polynomial or logarithmic space algorithms.

References


