Fundamenta Informaticae XX (2016) 1–27
DOI 10.3233/FI-2012-0000
IOS Press

Algorithmic Completeness of Imperative Programming Languages

Yoann Marquer[*]
Université Paris-Est Créteil (UPEC),
Laboratoire d'Algorithmique, Complexité et Logique (LACL),
IUT Sénart-Fontainebleau

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

Introduction

The notion of "algorithm" is currently a work in progress in the theoretical computer science community. Most of the ancient algorithms[1] were sequential algorithms, but nowadays there are also parallel, distributed, real-time, bio-inspired or quantum algorithms. Although this paper focuses only on sequential 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[2] 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[3], which can be seen as `if` *cond* `then` *actions* commands like in [2]. Moreover, the ASMs are closer to the imperative framework, 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[4] (see [10] for the *min* and [27] for the *gcd*). This `Loop` language has been extended in [1] with an `exit` command to obtain every Arithmetical Primitive Recursive Algorithm[5]. This work inspired us in our attempt to define an algorithmically complete imperative programming language.

An imperative language such as Jones' `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 language `While` at Section 2 by using only sequences, updates, conditionals and (unbounded) conditional loops. We will prove at Section 3 our main theorem:

Theorem 0.1. `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

[1]Such as the famous Euclidean algorithm, or the Babylonian method for approximating square roots.

[2]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).

[3]Gurevich implemented his ASMs via the programming language AsmL (see [9] for a comparison).

[4]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.

[5]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.

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[6] 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.

1. 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.

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 $\tau_A : S(A) \rightarrow S(A)$

Remark 1.1. 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[7] done before reaching a final state:

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

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

Remark 1.2. 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[8] $\overline{s}^X : \mathcal{U}_X^k \to \mathcal{U}_X$ 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 χ_R. Therefore every symbol in $\mathcal{L}_X$ is a function. Moreover, partial functions can be implemented with a special value $undef$.

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

The second postulate can be seen as a claim assuming that every data structure can be formalized as a first-order structure[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 τ_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 $true$, $false$, 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, $f \in \mathcal{L}_A$ be a dynamic k-ary function symbol, $a_1, \dots, a_k, b \in \mathcal{U}_X$. $(f, a_1, \dots, a_k)$ denotes a location of X and $(f, a_1, \dots, a_k, b)$ denotes an **update** on X at the location $(f, a_1, \dots, a_k)$.

If u is an update then $X + u$ is a new structure of language $\mathcal{L}_A$ and universe $\mathcal{U}_X$ such that the interpretation of a function symbol $f \in \mathcal{L}_A$ is:

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

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

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

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

If X and Y are two states of the same algorithm A then there exists a unique consistent set $\Delta = \{(f, \vec{a}, \overline{f}^Y(\vec{a}))\ ;\ \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\}$ of non trivial updates such that $Y = X + \Delta$. This Δ 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[10] 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 $(f, a_1, \dots, a_k, b) \in \Delta(A, X)$ then $a_1, \dots, a_k, b$ are interpretations in X of terms in T. So, since T is finite there exists a bounded number of $a_1, \dots, a_k, b$ such that $(f, a_1, \dots, a_k, b) \in \Delta(A, X)$. Moreover, since $\mathcal{L}_A$ is finite there exists a bounded number of dynamic symbols f. Therefore, $\Delta(A, X)$ has a bounded number of elements, and for every step of the algorithm only a bounded amount of work is done.

[10] For 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.

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 1.3. (The Lambda-Calculus)

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

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

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

This rule can be implemented in a machine:

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

These notations and this machine are directly taken from Krivine's [21]. In this machine π is a stack of terms. The symbol $\star$ is a separator between the current program and the current state of the memory. $\succ$ represents one step of computation, where only substitutions have a cost, not explorations inside a term, as is the case in the contextual transition rule. Programs in the machine are closed terms, so final states have the form $\lambda x.t \star \varnothing$.

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:

$$(t)t \rightarrow_p (t)t(t)t \rightarrow_p (t)t(t)t(t)t(t)t \rightarrow_p (t)t(t)t(t)t(t)t(t)t(t)t(t)t(t)t \rightarrow_p \dots$$

Indeed, at the step i exactly 2^{i-1} β-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 1.4. (Temporary Variables)

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

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

This simulation is very natural, but a fresh variable i is necessary. So, the language $\mathcal{L}_1$ of the simulating program must be bigger than the language $\mathcal{L}_2$ of the simulated program.

Notation 1.5. We follow the notation from [11], where $X|_{\mathcal{L}_2}$ denotes the **restriction** of the $\mathcal{L}_1$-structure X to the language $\mathcal{L}_2$. The language of $X|_{\mathcal{L}_2}$ is $\mathcal{L}_2$, its universe is the same than X, and every symbol $s \in \mathcal{L}_2$ has the same interpretation in $X|_{\mathcal{L}_2}$ than in X.

This notation is extended to a set of updates:

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

But fresh function symbols could be "too powerful", for example a dynamical unary symbol *env* alone would be able to store an unbounded amount of information. In order to obtain a fair simulation, we assume that the difference $\mathcal{L}_1 \setminus \mathcal{L}_2$ between 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[12] only on the constructors[13]. Because this initialization is independent (up to isomorphism) from the initial states, we call it a **uniform initialization**.

Example 1.6. (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 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.

[11]We will define p.11 the precise syntax of the imperative programs.

[12]Even the values of the fresh variables in the initial states can be irrelevant. See the program P_Π p.21 where the variables $\vec{v}$ are explicitly updated with the value of the terms $\vec{t}$ before being read.

[13]See the program Π_P p.16 where the boolean variable b_P is initialized with *true* and the others with *false*.

In other words, if we only observe an execution $X_0, X_1, X_2, X_3, X_4, X_5, X_6, ...$ 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 $Y_0, \dots, Y_m, Y_m, ...$ could have finished, but the simulating execution $X_0, \dots, X_{md}, X_{md+1}, \dots, X_{md+(d-1)}, X_{md}, X_{md+1}, ...$ may continue forever. So, an ending condition like $time(A, X) = d \times time(B, X) + e$ is necessary, and corresponds to the usual consideration for asymptotic time complexity.

Definition 1.7. (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 $d \in \mathbb{N}^\star$ 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 $i \in \mathbb{N}$, $X_{d\times i}|_{\mathcal{L}(P_2)} = Y_i$
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 these models of computation are **algorithmically equivalent**, which is denoted by $M_1 \simeq M_2$.

Remark 1.8. The second condition $X_{d\times i}|_{\mathcal{L}(P_2)} = Y_i$ implies for $i = 0$ that the initial states are the same, up to temporary variables.

2. Models of Computation

In this section the Gurevich's Abstract State Machines are defined, and we use his theorem `Algo` = `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 $M \simeq$ `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.

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 $\neg$ and the binary operations $\wedge$.

Definition 2.1. (ASM programs)

$$\begin{aligned} \Pi =_{def} & f t_1 ... t_k := t_0 \\ & \mid \texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif} \\ & \mid \texttt{par } \Pi_1 \| ... \| \Pi_n \texttt{ endpar} \end{aligned}$$

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

The sets $Read(\Pi)$ of the terms read by Π and $Write(\Pi)$ of the terms written by Π can be used to define the exploration witness of Π. But we will also use them in the rest of the article, especially to define the μ-formula F_Π p.23.

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

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

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

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

Remark 2.3. The exploration witness of Π is the closure by subterms of $Read(\Pi) \cup Write(\Pi)$ and not only $Read(\Pi)$ because the updates of a command could be trivial.

As said p.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 Π determines a transition function $\tau_\Pi(X) = X + \Delta(\Pi, X)$, where the set of updates $\Delta(\Pi, X)$ done by Π on X is defined by induction:

Definition 2.4. (Operational Semantics of the ASMs)

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

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

Remark 2.5. For every states X and Y, if the terms of $\mathit{Read}(\Pi)$ have the same interpretation on X and Y then $\Delta(\Pi, X) = \Delta(\Pi, Y)$.

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

Definition 2.6. An Abstract State Machine M with language $\mathcal{L}$ is given by:

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

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

```
     par   if $F_1$ then $\Pi_1$ endif
       ||  if $F_2$ then $\Pi_2$ endif
           ⋮
       ||  if $F_c$ then $\Pi_c$ endif
  endpar
```

where F_i are "guards", which means that for every state X one and only one F_i is true, and the programs Π_i have the form `par` $u_1 \| ... \| u_{m_i}$ `endpar`, where $u_1, \ldots, u_{m_i}$ are update commands.

Remark 2.7. $\Delta(\Pi_A, X) = \Delta(A, X) = \tau_A(X) - X$, 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 2.8. (Gurevich, 2000)

$$\mathtt{Algo} = \mathtt{ASM}$$

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

Remark 2.9. 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.

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 2.10. (Syntax of the `While` programs)

$$\begin{aligned}
\text{(commands) } c =_{def}\ & f t_1 ... t_k := t_0 \\
\mid\ & \texttt{if } F\ \{s_1\} \texttt{ else } \{s_2\} \\
\mid\ & \texttt{while } F\ \{c; s\} \\
\text{(sequences) } s =_{def}\ & \epsilon \mid c; s \\
\text{(programs) } P =_{def}\ & \{s\}
\end{aligned}$$

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

Notation 2.11. The symbol ϵ 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 `if` F $\{s_1\}$. Let `skip` be the command `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 1.3 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 2.12. Operational semantics of the `While` programs:

$$\begin{aligned}
\{f t_1 ... t_k := t_0; s\} \star X &\succ \{s\} \star X + (f, \overline{t_1}^X, \dots, \overline{t_k}^X, \overline{t_0}^X) & \\
\{\texttt{if } F\ \{s_1\} \texttt{ else } \{s_2\}; s_3\} \star X &\succ \{s_1; s_3\} \star X & \text{if } \overline{F}^X = true \\
\{\texttt{if } F\ \{s_1\} \texttt{ else } \{s_2\}; s_3\} \star X &\succ \{s_2; s_3\} \star X & \text{if } \overline{F}^X = false \\
\{\texttt{while } F\ \{s_1\}; s_2\} \star X &\succ \{s_1; \texttt{while } F\ \{s_1\}; s_2\} \star X & \text{if } \overline{F}^X = true \\
\{\texttt{while } F\ \{s_1\}; s_2\} \star X &\succ \{s_2\} \star X & \text{if } \overline{F}^X = false
\end{aligned}$$

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 2.13. If $P_1 \star X_1 \succ_i P_2 \star X_2$ and $P_2 \star X_2 \succ_j P_3 \star X_3$ then $P_1 \star X_1 \succ_{i+j} P_3 \star X_3$, so in a sense $\succ_i$ is a transitive relation.

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

We could have introduced a rule $\{\} \star X \succ \{\} \star X$ and defined the termination like Gurevich did for `Algo`: $P \star X$ is terminal if $P \star X \succ P \star X$. But if $\overline{F}^X = \mathit{true}$ then $\{\texttt{while } F\ \{\}; s\} \star X \succ \{\texttt{while } F\ \{\}; s\} \star X$. 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 2.10 the commands `while` F $\{\}$. As a consequence, if $P_1 \star X_1 \succ P_2 \star X_2$ then $P_1 \neq P_2$.

Notation 2.14. P **terminates** on X, denoted by $P \downarrow X$, if there exists i and X' such that:

$$P \star X \succ_i \{\} \star X'$$

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

Example 2.15. This program computes the minimum of two integers m and n in $O(\mathit{min}(m, n))$ steps, and stores the result in the output variable x:

$$P_{min} = \{x := 0;\ \texttt{while } \neg(x = m \vee x = n)\ \{x := x + 1;\};\}$$

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

$$\begin{array}{lr}
 & \{x := 0;\ \texttt{while } \neg(x = 2 \vee x = 3)\ \{x := x + 1;\};\} \star X \\
\succ & \{\texttt{while } \neg(x = 2 \vee x = 3)\ \{x := x + 1;\};\} \star X + (x, 0) \\
\succ & \{x := x + 1;\ \texttt{while } \neg(x = 2 \vee x = 3)\ \{x := x + 1;\};\} \star X + (x, 0) \\
\succ & \{\texttt{while } \neg(x = 2 \vee x = 3)\ \{x := x + 1;\};\} \star X + (x, 1) \\
\succ & \{x := x + 1;\ \texttt{while } \neg(x = 2 \vee x = 3)\ \{x := x + 1;\};\} \star X + (x, 1) \\
\succ & \{\texttt{while } \neg(x = 2 \vee x = 3)\ \{x := x + 1;\};\} \star X + (x, 2) \\
\succ & \{\} \star X + (x, 2)
\end{array}$$

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

Notation 2.16. Let s_P be the sequence such that $P = \{s_P\}$. The **composition** P_1P_2 of the imperative programs P_1 and P_2 is defined by $P_1P_2 = \{s_{P_1}; s_{P_2}\}$.

`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 2.17. (Composition of Programs)
P_1P_2 terminates on X if and only if P_1 terminates on X and P_2 terminates on $P_1(X)$, such that $P_1P_2(X) = P_2(P_1(X))$ and $time(P_1P_2, X) = time(P_1, X) + time(P_2, P_1(X))$.

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 2.18. (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 $P \star X \succ_i P' \star X'$. Let $\tau_X^i(P)$ be that P' and $\tau_P^i(X)$ be that X', so:

$$P \star X \succ_i \tau_X^i(P) \star \tau_P^i(X)$$

If $i > time(P, X)$ we can assume that $\tau_X^i(P) = \{\}$ and $\tau_P^i(X) = P(X)$.

Remark 2.19. τ_P^i is not a transition function in the sense of the first section, because:

$$\tau_P^i(X) \neq \tau_P \circ \cdots \circ \tau_P(X)$$

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

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

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

Definition 2.20. The set of updates made by P on X is:

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

Remark 2.21. If P terminates on X then the cardinal of $\Delta(P, X)$ is bounded by $time(P, X)$.

In imperative programming languages, an **overwrite** occurs when a variable is updated to a value, then is updated to another value later in the execution. In our framework, this means that there exists in $\Delta(P, X)$ two updates $(f, \vec{a}, b)$ and $(f, \vec{a}, b')$ with $b \neq b'$, which makes $\Delta(P, X)$ inconsistent. So, we say that P is without overwrite on X if $\Delta(P, X)$ is consistent.

Proposition 2.22. (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]. □

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

3. Algorithmic Completeness

We want to prove that `While` is algorithmically complete. According to Gurevich's theorem p.10, this would mean that `While` $\simeq$ `ASM`. Considering our definition p.8, we will have to prove that `ASM` fairly simulates `While`, and that `While` fairly simulates `ASM`.

`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[15].

Example 3.1. The imperative program P_{min} of the example 2.15 p.12:

```
0 :  $x := 0$
1 :  while $\neg(x = m \vee x = n)$
2 :      $x := x + 1$
```

could be translated into the following ASM program:

```
    par  if $line = 0$ then
             par $x := 0 \parallel line := 1$ 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 $x := x + 1 \parallel line := 1$ endpar
         endif
endpar
```

Remark 3.2. The number of a line is between 0 and $length(P)$. So, a finite number of booleans $b_0, b_1, \ldots, b_{length(P)}$ can be used[16] instead of an integer $line$.

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

Example 3.3. (Marked `While`)

[15]Programs of this form are called control state ASMs (see [8]).
[16]Remember that booleans must be in the data structure, but integers may not.

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

$$P = \{\texttt{while}\ \mathit{true}\ \{x := x + 1;\ \texttt{while}\ \mathit{true}\ \{x := x + 1;\};\};\}$$

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

P_1	$\succ P_2$	$\succ P_3$	$\succ P_4$	$\succ \dots$
`while` *true*	$\bbox[border: 1px dashed]{x := x + 1}$	`while` *true*	$\boxed{x := x + 1}$	
$\bbox[border: 1px dashed]{x := x + 1}$	`while` *true*	$\boxed{x := x + 1}$	`while` *true*	
`while` *true*	$\boxed{x := x + 1}$	`while` *true*	$\boxed{x := x + 1}$	
$\boxed{x := x + 1}$	`while` *true*	$\bbox[border: 1px dashed]{x := x + 1}$	`while` *true*	
	$\bbox[border: 1px dashed]{x := x + 1}$	`while` *true*	$\bbox[border: 1px dashed]{x := x + 1}$	
	`while` *true*	$\boxed{x := x + 1}$	`while` *true*	
	$\boxed{x := x + 1}$		$\boxed{x := x + 1}$	

We could have replaced P_2 by P_4 in this execution without changing anything except the boxes. $\bbox[border: 1px dashed]{x := x + 1}$ 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 $b_0, b_1, \dots, b_{length(P)}$ indexed by the lines of the program, but booleans indexed by the possible states of the program during the execution. The possible executions of a program will be represented by a graph where the edges are the possible transitions, and the vertices are the possible programs:

Example 3.4. (The Graph of Execution of P_{min})

$$\begin{tikzcd}
\begin{array}{l} x := 0 \\ \texttt{while}\ \neg(x = m \lor x = n) \\ \quad x := x + 1 \end{array} \arrow[r] & \begin{array}{l} \texttt{while}\ \neg(x = m \lor x = n) \\ \quad x := x + 1 \end{array} \arrow[r, bend left] \arrow[d] & \begin{array}{l} x := x + 1 \\ \texttt{while}\ \neg(x = m \lor x = n) \\ \quad x := x + 1 \end{array} \arrow[l, bend left] \\
 & \{\} &
\end{tikzcd}$$

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

$$\mathcal{G}(P_{min}) = \Big\{ \begin{array}{r} \{x := 0;\ \texttt{while}\ \neg(x = m \lor x = n)\ \{x := x + 1;\};\}, \\ \{\texttt{while}\ \neg(x = m \lor x = n)\ \{x := x + 1;\};\}, \\ \{x := x + 1;\ \texttt{while}\ \neg(x = m \lor x = n)\ \{x := x + 1;\};\}, \\ \{\} \end{array} \Big\}$$

Notation 3.5. In order to define graphs of execution we need to introduce the notation:

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

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

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

Definition 3.6. (Graph of Execution)

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

$$\begin{aligned}\mathcal{G}(ft_1 \dots t_k := t_0) &=_{def} \boldsymbol{\{}\{ft_1 \dots t_k := t_0;\}\boldsymbol{\}}\\ \mathcal{G}(\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}) &=_{def} \boldsymbol{\{}\{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\};\}\boldsymbol{\}} \cup \ \mathcal{G}(P_1) \cup \mathcal{G}(P_2)\\ \mathcal{G}(\texttt{while } F \ \{P_1\}) &=_{def} \mathcal{G}(P_1)\{\texttt{while } F \ \{P_1\};\}\end{aligned}$$

As intended, we can prove (see [23]) that $card(\mathcal{G}(P)) \leq length(P) + 1$. So, only a finite number of guards depending only on P are necessary. Notice that for some programs (like P_{min} in example 3.4 p.15) which do not follow example 3.3 p.14, $card(\mathcal{G}(P)) = length(P) + 1$ can be reached, so the bound is optimal.

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

Proposition 3.7. (Operational Closure of Graph of Execution)

$$\begin{aligned}&\text{If } ft_1 \dots t_k := t_0 \ P' \in \mathcal{G}(P) \text{ then } P' \in \mathcal{G}(P)\\ \text{If } &\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\} \ P' \in \mathcal{G}(P) \text{ then } P_1P' \text{ and } P_2P' \in \mathcal{G}(P)\\ &\text{If } \texttt{while } F \ \{P_1\} \ P' \in \mathcal{G}(P) \text{ then } P_1 \ \texttt{while } F \ \{P_1\} \ P' \text{ and } P' \in \mathcal{G}(P)\end{aligned}$$

Notation 3.8. The fresh boolean variables will be denoted $b_{P_{\mathcal{G}}}$ where $P_{\mathcal{G}} \in \mathcal{G}(P)$. Only one $b_{P_{\mathcal{G}}}$ 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 3.7 ensures that the following translation is well-defined:

Definition 3.9. (Translation of imperative programs into ASM)

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

where P^{tr} is defined by induction:

$$\begin{aligned}
\{\}^{tr} &=_{def} \texttt{skip} \\
(ft_1 \dots t_k := t_0\ P')^{tr} &=_{def} \texttt{par}\ b_{ft_1 \dots t_k := t_0\ P'} := false \\
&\qquad \parallel ft_1 \dots t_k := t_0 \\
&\qquad \parallel b_{P'} := true \\
&\qquad \texttt{endpar} \\
(\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P')^{tr} &=_{def} \texttt{par}\ b_{\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'} := false \\
&\qquad \parallel \texttt{if}\ F\ \texttt{then}\ b_{P_1 P'} := true\ \texttt{else}\ b_{P_2 P'} := true\ \texttt{endif} \\
&\qquad \texttt{endpar}
\end{aligned}$$

$$\begin{aligned}
(\texttt{while}\ F\ \{P_1\}\ P')^{tr} &=_{def} \texttt{par}\ b_{\texttt{while}\ F\ \{P_1\}\ P'} := false \\
&\qquad \parallel \texttt{if}\ F\ \texttt{then}\ b_{P_1\ \texttt{while}\ F\ \{P_1\}\ P'} := true\ \texttt{else}\ b_{P'} := true\ \texttt{endif} \\
&\qquad \texttt{endpar}
\end{aligned}$$

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

Proposition 3.10. (Step-by-Step Simulation)

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

Theorem 3.11. `ASM` fairly simulates `While`.

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

1. $\mathcal{L}_{\Pi_P} = \mathcal{L}_P \cup \{b_{P_\mathcal{G}}\ ;\ P_\mathcal{G} \in \mathcal{G}(P)\}$
   where $card(\{b_{P_\mathcal{G}}\ ;\ P_\mathcal{G} \in \mathcal{G}(P)\}) \le length(P) + 1$.

2. Using proposition 3.10, we can prove by induction on $i \le time(P, X)$ that:
   $\tau_{\Pi_P}^i(X[b_P]) = \tau_P^i(X)[b_{\tau_X^i(P)}]$
   So $\tau_{\Pi_P}^i(X[b_P])|_{\mathcal{L}_P} = \tau_P^i(X)$
   And the temporal dilation is $\boxed{d = 1}$.

3. If $i = time(P, X)$ then $\tau_X^i(P) = \{\}$
   So $\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}]) = \varnothing$, and $\tau_{\Pi_P}^{i+1}(X[b_P]) = \tau_{\Pi_P}^i(X[b_P])$
   So $time(\Pi_P, X) \le time(P, X)$ (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 $i < time(P, X)$, $b_{\tau_X^i(P)}$ is updated, so $\tau_{\Pi_P}^{i+1}(X[b_P]) \neq \tau_{\Pi_P}^i(X[b_P])$
   So $time(\Pi_P, X) \ge time(P, X)$ (2)
   Therefore, according to (1) and (2), $time(\Pi_P, X) = time(P, X)$, and $\boxed{e = 0}$.

□

`While` simulates `ASM`

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

Definition 3.12. (Syntactical Translation of the `ASM` programs)

$$\begin{aligned}(ft_1 \dots t_k := t_0)^{tr} &=_{def} \{ft_1 \dots t_k := t_0;\}\\ (\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})^{tr} &=_{def} \{\texttt{if } F \texttt{ then } \Pi_1^{tr} \texttt{ else } \Pi_2^{tr};\}\\ (\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar})^{tr} &=_{def} \Pi_1^{tr} \ \dots \ \Pi_n^{tr} \text{ (composition)}\end{aligned}$$

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

Example 3.13. Let X be a structure such that $\overline{x}^X = 0$ and $\overline{y}^X = 1$, and Π be the program:

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

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

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

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

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

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:

$$\begin{aligned} & \{x := w; y := v;\} \star X\\ \succ & \qquad \{y := v;\} \star X + \{(x,1)\}\\ \succ & \qquad\qquad \{\} \star X + \{(x,1),(y,0)\}\end{aligned}$$

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

Definition 3.14. (Substitution of a Term by a Variable)

$$\begin{aligned}\{\}[v/t] &=_{def} \{\}\\ (cP)[v/t] &=_{def} c[v/t]P[v/t]\end{aligned}$$

$$\begin{aligned}(ft_1 \dots t_k := t_0)[v/t] &=_{def} ft_1[v/t]...t_k[v/t] := t_0[v/t]\\ (\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\})[v/t] &=_{def} \texttt{if } F[v/t]\ \{P_1[v/t]\} \texttt{ else } \{P_2[v/t]\}\\ (\texttt{while } F\ \{P_1\})[v/t] &=_{def} \texttt{while } F[v/t]\ \{P_1[v/t]\}\end{aligned}$$

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

Remark 3.15. 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 $t_1, t_2, \dots, t_k$ and k fresh distinct variables $v_1, v_2, \dots, v_k$, the notation $P[\vec{v}/\vec{t}]$ is not ambiguous, because the substitutions can be made in any order.

We defined p.9 the set $Read(\Pi)$ of the terms read by Π. Let $r = card(Read(\Pi))$, and $t_1, \dots, t_r$ be the distinct terms read by Π. We substitute them by the fresh variables $v_{t_1}, \dots, v_{t_r}$, 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 Π is in normal form (see p.10). Its translation $\Pi^{tr}[\vec{v_t}/\vec{t}]$ 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:

   $$v_{t_1} := t_1; \dots; v_{t_r} := t_r;$$

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 *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.

   This is an issue because, according to our definition of the fair simulation p.8, every step of the ASM Π must be simulated by d steps, where d depends only on Π.

   In order to obtain a uniform temporal dilation, we will add $m - m_i$ `skip` commands[17] at the end of each block, where m is defined by:

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

Figure 1. Translation of a Normal Form ASM

```
$\Pi =$
par  if $F_1$ then
             par $f_1^1(\vec{t_1^1}) := t_1^1$
              || $f_2^1(\vec{t_2^1}) := t_2^1$
                 $\vdots$
              || $f_{m_1}^1(\vec{t_{m_1}^1}) := t_{m_1}^1$
         endpar
     endif
  || if $F_2$ then
             par $f_1^2(\vec{t_1^2}) := t_1^2$
              || $f_2^2(\vec{t_2^2}) := t_2^2$
                 $\vdots$
              || $f_{m_2}^2(\vec{t_{m_2}^2}) := t_{m_2}^2$
         endpar
     endif
     $\vdots$
  || if $F_c$ then
             par $f_1^c(\vec{t_1^c}) := t_1^c$
              || $f_2^c(\vec{t_2^c}) := t_2^c$
                 $\vdots$
              || $f_{m_c}^c(\vec{t_{m_c}^c}) := t_{m_c}^c$
         endpar
     endif
endpar
```

```
$\Pi^{tr}[\vec{v_t}/\vec{t}] = \{$
    if $v_{F_1}$ then $\{$
        $f_1^1(\vec{v}_{t_1^1}) := v_{t_1^1}$;
        $f_2^1(\vec{v}_{t_2^1}) := v_{t_2^1}$;
        $\vdots$
        $f_{m_1}^1(\vec{v}_{t_{m_1}^1}) := v_{t_{m_1}^1}$;

    $\}$;
    if $v_{F_2}$ then $\{$
        $f_1^2(\vec{v}_{t_1^2}) := v_{t_1^2}$;
        $f_2^2(\vec{v}_{t_2^2}) := v_{t_2^2}$;
        $\vdots$
        $f_{m_2}^2(\vec{v}_{t_{m_2}^2}) := v_{t_{m_2}^2}$;

    $\}$;
    $\vdots$
    if $v_{F_c}$ then $\{$
        $f_1^c(\vec{v}_{t_1^c}) := v_{t_1^c}$;
        $f_2^c(\vec{v}_{t_2^c}) := v_{t_2^c}$;
        $\vdots$
        $f_{m_c}^c(\vec{v}_{t_{m_c}^c}) := v_{t_{m_c}^c}$;

    $\}$;
$\}$
```

Figure 2. Translation P_Π of one step of Π

```
$P_\Pi =_{def}$ {
    $v_{t_1} := t_1$;
    $v_{t_2} := t_2$;
    $\vdots$
    $v_{t_r} := t_r$;
    if $v_{F_1}$ then {
        $f_1^1(\vec{v}_{t_1^1}) := v_{t_1^1}$;
        $f_2^1(\vec{v}_{t_2^1}) := v_{t_2^1}$;
        $\vdots$
        $f_{m_1}^1(\vec{v}_{t_{m_1}^1}) := v_{t_{m_1}^1}$;
        skip;
        $\vdots$ $(m - m_1$ times)
        skip;
    };
    if $v_{F_2}$ then {
        $f_1^2(\vec{v}_{t_1^2}) := v_{t_1^2}$;
        $f_2^2(\vec{v}_{t_2^2}) := v_{t_2^2}$;
        $\vdots$
        $f_{m_2}^2(\vec{v}_{t_{m_2}^2}) := v_{t_{m_2}^2}$;
        skip;
        $\vdots$ $(m - m_2$ times)
        skip;
    };
    $\vdots$
    if $v_{F_c}$ then {
        $f_1^c(\vec{v}_{t_1^c}) := v_{t_1^c}$;
        $f_2^c(\vec{v}_{t_2^c}) := v_{t_2^c}$;
        $\vdots$
        $f_{m_c}^c(\vec{v}_{t_{m_c}^c}) := v_{t_{m_c}^c}$;
        skip;
        $\vdots$ $(m - m_c$ times)
        skip;
    };
}
```

We obtain at figure 2 p.21 the translation P_Π of one step of the ASM program Π. Let X be a state of the ASM with program Π, extended with the variables $\vec{v_t}$. As intended, we prove that P_Π simulates one step of Π in a constant time t_Π:

Proposition 3.16. (Semantical Translation of the `ASM` programs)

There exists t_Π, depending only on Π, such that for every state X of P_Π:

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

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

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

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

Since for every following state Y, $\overline{v_{t_k}}^Y = \overline{t_k}^X$, the set of updates done in the block of F_i is $\Delta(\Pi, X|_{\mathcal{L}_\Pi})$. These updates require m_i steps, then the `skip` 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 Π:

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

The updates done by P_Π are the initial updates and the updates done in the block of F_i:

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

P_Π contains only updates and conditionals, so according to proposition 2.18 p.13 this program is terminal. Moreover, the fresh variables are updated only once, and since Π is in normal form, $\Delta(\Pi, X|_{\mathcal{L}_\Pi})$ is consistent. So, P_Π terminates without overwrite on X, and according to proposition 2.22 p.13:

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

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 3.17. $P_\Pi^i(X)|_{\mathcal{L}_\Pi} = \tau_\Pi^i(X|_{\mathcal{L}_\Pi})$

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

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_Π we have:

$$\overline{v_{t_k}}^{P_\Pi^{i+1}(X)} = \overline{t_k}^{P_\Pi^i(X)}$$

This remark is particularly useful because it allows us to detect the end of the execution of Π. The program P_Π simulates one step of Π 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 Π, but we need to define an appropriate F.

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

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

We call it the "μ-formula" because it is similar to the minimization operator μ from recursive functions (see [11]):

Lemma 3.18. (The μ-formula)

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

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

Notice that F_Π becomes *true* after $\mathit{time}(\Pi, X|_{\mathcal{L}_\Pi}) + 1$ steps, so we execute the program P_Π one more time after the end of Π. 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 μ-formula F_Π. We do that simply by adding an occurrence of P_Π at the beginning of the program. So, in a sense, the correct control structure should be a `do while` and not a `while`.

Theorem 3.19. `While` fairly simulates `ASM`.

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

1. There are $r = \mathit{card}(\mathit{Read}(\Pi))$ fresh variables $\vec{v_t}$.

2. According to lemma 3.18 p.23, the execution of this program on X is:

$$\begin{aligned} P_\Pi \ \texttt{while}\ \neg F_\Pi\ \{P_\Pi\} \star X &\succ_{t_\Pi} \quad \texttt{while}\ \neg F_\Pi\ \{P_\Pi\} \star P_\Pi(X) \\ &\succ \ P_\Pi\ \texttt{while}\ \neg F_\Pi\ \{P_\Pi\} \star P_\Pi(X) \\ &\succ_{t_\Pi} \quad \texttt{while}\ \neg F_\Pi\ \{P_\Pi\} \star P_\Pi^2(X) \\ &\succ \ P_\Pi\ \texttt{while}\ \neg F_\Pi\ \{P_\Pi\} \star P_\Pi^2(X) \\ &\vdots \\ &\succ \ P_\Pi\ \texttt{while}\ \neg F_\Pi\ \{P_\Pi\} \star P_\Pi^{time(\Pi, X|_{\mathcal{L}_\Pi})}(X) \\ &\succ_{t_\Pi} \quad \texttt{while}\ \neg F_\Pi\ \{P_\Pi\} \star P_\Pi^{time(\Pi, X|_{\mathcal{L}_\Pi})+1}(X) \\ &\succ \qquad\qquad \{\} \star P_\Pi^{time(\Pi, X|_{\mathcal{L}_\Pi})+1}(X) \end{aligned}$$

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

But, according to corollary 3.17 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.
$$\begin{aligned} time(P, X) &= (t_\Pi + 1) \times (time(\Pi, X|_{\mathcal{L}_\Pi}) + 1) \\ &= d \times time(\Pi, X|_{\mathcal{L}_\Pi}) + e \end{aligned}$$

where $\boxed{e = t_\Pi + 1}$.

□

According to theorem 3.11, every program in `While` can be simulated by an ASM Π, which itself can be simulated by P_Π `while` $\neg F_\Pi$ $\{P_\Pi\}$. So, an interesting corollary of theorem 3.19 is that every program in `While` is equivalent to a program using only one `while` command. This result can be seen as an algorithmic Kleene's Normal Form Theorem (see [20]).

Conclusion

We have proven the two sides of the simulation p.17 and p.23. Therefore, according to our definition p.8, `While` $\simeq$ `ASM`. Consequently, imperative programming languages are algorithmically complete, up to data structures.

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

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 Π, where r is the number of terms read by Π, c is the number of conditionals of Π, and m is the maximal number of updates per block of Π.

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:

$$\begin{aligned}
\{(f_1\vec{t_1},\dots,f_k\vec{t_k}) := (t_1,\dots,t_k); s\} \star X &\succ_1 \quad \{s\} \star X + \{(f_1, \vec{t_1}^X, \overline{t_1}^X), \dots, (f_k, \vec{t_k}^X, \overline{t_k}^X)\} \\
\{\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}; s_3\} \star X &\succ_0 \{s_i; s_3\} \star X \\
&\qquad \text{where } i = \begin{cases} 1 \text{ if } \overline{F}^X = true \\ 2 \text{ if } \overline{F}^X = false \end{cases} \\
\{\texttt{while}\ F\ \{s_1\}; s_2\} \star X &\succ_0 \{s; s_2\} \star X \\
&\qquad \text{where } s = \begin{cases} s_1; \texttt{while}\ F\ \{s_1\};\ \text{if } \overline{F}^X = true \\ \epsilon \text{ if } \overline{F}^X = false \end{cases}
\end{aligned}$$

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

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

[2] Jacques Arsac *: Algorithmique et langages de programmation*, Bulletin de l'EPI 64 (1991) 115-124

[3] Therese Biedl, Jonathan F. Buss, Erik D. Demaine, Martin L. Demaine, Mohammadtaghi Hajiaghayi, Tomas Vinar. *: Palindrome recognition using a multidimensional tape*, Theoretical Computer Science 302 (2003)

[4] Andreas Blass , Yuri Gurevich *: Algorithms vs. Machines*, Bulletin of the European Association for Theoretical Computer Science Number 77 (2002) 96-118

[5] Andreas Blass , Yuri Gurevich. *: Abstract state machines capture parallel algorithms*, ACM transactions on Computational Logic Voulme 9 Issue 3 (2008)

[6] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich *: When are two algorithms the same?*, Bull. Symbolic Logic Volume 15, Issue 2 (2009), 145-168

[7] Blum, Manuel *: A Machine-Independent Theory of the Complexity of Recursive Functions*, Journal of the ACM Volume 14 (1967), 322-336

[8] Egon Börger *: Abstract State Machines: A Unifying View of Models of Computation and of System Design Frameworks*, Annals of Pure and Applied Logic (2005)

[9] Patrick Cégielski, Irène Guessarian *: Normalization of Some Extended Abstract State Machines*, Fields of Logic and Computation, Lecture Notes in Computer Science Volume 6300 (2010) 165-180

[10] Loïc Colson *: About primitive recursive algorithms*, Theoretical Computer Science 83 (1991) 57–69

[11] René Cori, Daniel Lascar and Donald Pelletier *: Mathematical Logic: A Course With Exercises : Part I and II*, Paris, Oxford University Press (2000, 2001)

[12] Marie Ferbus-Zanda, Serge Grigorieff *: ASM and Operational Algorithmic Completeness of Lambda Calculus*, Fields of Logic and Computation (2010)

[13] Uwe Glaesser, Yuri Gurevich, Margus Veanes. *: Universal Plug and Play Machine Models*, Proc. of IFIP World Computer Congress, Stream 7 on Distributed and Parallel Embedded Systems (2002)

[14] Serge Grigorieff, Pierre Valarcher *: Evolving Multialgebras unify all usual models for computation in sequential time*, Symposium on Theoretical Aspects of Computer Science (2010)

[15] Serge Grigorieff, Pierre Valarcher *: Classes of Algorithms: Formalization and Comparison*, Bulletin of the EATCS 107 (2012)

[16] Yuri Gurevich *: Sequential Abstract State Machines Capture Sequential Algorithms*, ACM Transactions on Computational Logic (2000)

[17] Yuri Gurevich *: Interactive Algorithms*, Mathematical Foundations of Computer Science (2005)

[18] James K. Huggins and Wuwei Shen *: The Static and Dynamic Semantics of C*, Technical Report (2000)

[19] Neil D. Jones *: LOGSPACE and PTIME characterized by programming languages*, Theoretical Computer Science 228 (1999) 151-174

[20] Stephen Cole Kleene *: Recursive predicates and quantifiers*, Transactions of the AMS v. 53 n. 1 (1943) 41-73

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

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

[23] Yoann Marquer *: Algorithmic Completeness of Imperative Programming Languages (Long Version)*, dr-apeiron.net/doku.php/en:research:fi-while (2016, technical report)

[24] Yoann Marquer *: An Imperative Language Complete for PTime Algorithms*, dr-apeiron.net/doku.php/en:research:ploopc (2016, in review)

[25] Albert Meyer, Dennis Ritchie *: The complexity of loop programs*, ACM 22nd national conference (1967) 465-469

[26] Yiannis N. Moschovakis *: What is an algorithm?*, Mathematics Unlimited (2001)

[27] Yiannis N. Moschovakis *: On Primitive Recursive Algorithms and the Greatest Common Divisor Function*, Theoretical Computer Science (2003)

[28] Robert I. Soare *: Computability and Recursion*, Bulletin of Symbolic Logic 2 (1996) 284-321

[29] Alan M. Turing *: On Computable Numbers, with an Application to the Entscheidungsproblem*, Proc. London Math. Soc. Issue 2, vol. 42 (1937) 230-265

[30] Noson S. Yanofsky *: Towards a Definition of an Algorithm*, Journal of Logic and Computation (2010)

[31] Noson S. Yanofsky *: Galois Theory of Algorithms*, Kolchin Seminar in Differential Algebra (2010)