Fundamenta Informaticae XX (2016) 1–50
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

Contents

*This work is partially supported by the french program ANR 12 BS02 007 01

Abstract

According to the Church-Turing Thesis, effectively calculable functions are the functions computable by a Turing machine. Models that compute these functions are called Turing-complete. For example, we know that usual imperative languages (like *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 but also on the step-by-step behavior of the computation. Moreover, the issue is not limited to the set of partial recursive functions, and applies to any desired set of functions. Indeed, a model could compute all the desired functions, but some algorithms (ways to compute these functions) could be missing (see [11] for an example related to primitive recursive algorithms).

This paper is devoted to prove that usual imperative languages are not only Turing-complete but also algorithmically complete, using the axiomatic definition of the Gurevich's Thesis and a fair bisimulation between his Abstract State Machines and a version of Jones' `While` programs. No special knowledge is assumed, because all relevant material will be explained from scratch.

Keywords 1. Theory of Algorithms, Models of Computation, Completeness.

Introduction

Algorithms have been studied for ages, even before the famous Euclidean algorithm[1]. The study of algorithms can be seen as a field between computability theory and computer programming, but unlike functions and programs there is still no consensus for a formal definition of algorithms. Most of the ancient algorithms were sequential algorithms, but nowadays there exists also parallel, distributed, real-time, bio-inspired or quantum algorithms, which involve many fields in science.

Because the notion of algorithm is currently a work in progress in the scientific community I thought convenient to explain my approach with a dialog between the Author and Quisani, an inquisitive former student responding to Gurevich in [4]. Knuth himself wrote on the pedagogical role of the dialog in [21].

Q: I read[2] that, according to the Church Thesis, Turing machines can compute every function calculable by a human with pencil and paper. Of course, usual imperative languages like *C*, *Ada* or *Python* can implement a Turing Machine, so they must be algorithmically complete. What is new in your approach?

A: Indeed, like recursive functions or the lambda-calculus, imperative languages compute the same set of functions as Turing machines. These models of computation are called Turing-complete. But Turing-completeness is not algorithmic completeness.

Q: Can you explain the difference?

[1] One example is the Babylonian method for approximating square roots.

[2] For example, in [26], where Robert I. Soare insists on the intentional difference between recursive functions and computable functions.

A. Sure. The notion of Turing-completeness describes an input-output behavior. Up to simulation, a model is said to be Turing-complete if for a given initial state it can compute the same final state than a Turing machine. This is only functional completeness. Algorithmic completeness is a notion describing a step-by-step behavior. Up to simulation, a model is said algorithmically complete if for a given initial state it can compute the same execution as a given algorithm. From an algorithmic point of view, intermediate states matter.

Q: Do you mean that different executions have different costs in space and time? We already know how to distinguish some classes of complexity. For example, Neil D. Jones characterized *LogSpace* or *Ptime* with the imperative `While` language in [20].

A: Yes, the complexity is one, and probably the most common, way to distinguish algorithms. For example, the palindrome recognition can be done in $O(n)$ steps with a two-tape Turing machine, but requires at least (see [3]) $O(n^2/log(n))$ steps with a one-tape Turing machine. These models are not algorithmically equivalent. But we know that a one-tape Turing machine can simulate the results of a two-tape Turing machine, so they are functionally equivalent.

Q: So, if I have understood correctly... A one-tape Turing machine can simulate the results of a two-tape Turing machine. But some algorithms requiring $O(n)$ steps in the two-tape Turing machines are missing in the one-tape Turing machines. So the two-tape Turing machines are strictly "algorithmically" stronger than the one-tape Turing machines. And the one-tape Turing machines cannot be algorithmically complete... but I guess this depends on the set of algorithms you consider.

A: You are right. For example, we have interesting results about primitive recursive algorithms. Loïc Colson proved in [11] that the minimum function cannot be computed in $O(min(m, n))$ steps with primitive recursion. So the following algorithm is missing:

`Inputs` m, n
`Initialization` $x := 0$
`Do` $x := x + 1$ `Until` $x = m$ `or` $x = n$
`Output` x

Yiannis N. Moschovakis proved a similar result in [25] about *LogTime* algorithms for the greatest common divisor. For the same reasons an equivalent (see [23]) imperative language like the `Loop` of Albert Meyer and Dennis Ritchie cannot be complete for the primitive recursive algorithms. But Philippe Andary, Bruno Patrou and Pierre Valarcher proved in [1] that adding an `exit` command is sufficient to obtain all the primitive recursive algorithms on unary integers.

Q: Why do you need all the algorithms?

A: To be sure we have the best one[3], whatever can be our criteria.

Q: Ok... but I realize that I am not really sure of what you call "algorithms"... I mean, without a clear formalization of what an algorithm is, you simply cannot determine if a model of computation is algorithmically complete or not.

A: Nowadays, unlike functions or programs, there is no clear consensus about the formalization of algorithms, even if we consider only sequential algorithms. To my knowledge, there is three main approaches:

[3]Only if a "best" algorithm can be found for the desired criteria. For example, Blum proved in [8] 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.

- Noson S. Yanofsky, in [28], considered algorithms as an equivalence class of programs on functions[4]. He was criticized by Yuri Gurevich in [7], because as Yanofsky said "whether or not two programs are the same... is really a subjective decision".

- Yuri Gurevich formalized sequential algorithms as Abstract State Machines[5] by `if` *cond* `then` *actions* commands like in [2]. This approach was criticized by Yiannis Moschovakis because algorithms should not be limited to machine systems.

- Yiannis Moschovakis defined in [24] algorithms as a system of fixed-point equations called recursors. Gurevich responded in [5] that these definitions are specifications and not algorithms, and that Moschovakis' non-mechanical algorithms could be implemented by more general machines.

Q: So, for the moment there is no consensus like the one between Church, Gödel, Kleene and Turing about the computable functions after the publication of Turing's article [27]. So, which one have you chosen ?

A: I was convinced by the axiomatization of the Gurevich Thesis, based on the three postulates of sequential time, abstract states and bounded exploration. He proved in [17] that his axiomatic approach is identical to his Abstract State Machines, so the ASMs are a model of reference to determine if another model of computation is algorithmically complete or not. Moreover, the ASMs are closer to an imperative framework than the recursors of Moschovakis, so they seemed more appropriate for my purpose.

Q: So, in a way, according to you, ASMs have the same role in the Gurevich Thesis as Turing machines had in the Church Thesis. I remember what you said about intermediate states... so I suppose that for you a model of computation is algorithmically complete if it simulates the same executions as the ASMs?

A: Exactly. The aim of this paper is the following theorem:

Theorem 1. The imperative programs and the abstract state machines can fairly simulate each other.

Q: What do you mean by a "fair simulation"?

A: The simulation uses temporary variables and a constant temporal dilation, like in [13] where Marie Ferbus-Zanda and Serge Grigorieff proved the algorithmic completeness of the lambda-calculus. Philippe Andary, Bruno Patrou and Pierre Valarcher used the same method in [1] to obtain all the arithmetical primitive recursive algorithms.

Q: But imperative programming languages are very different, how can you prove a theorem for every of them?

A: They have different syntaxes but common features, in particular for structured programming: sequences of assignment statements, conditional branching statements, looping statements in block structures. To implement these common features I will introduce my own version of the Jones' `While` language, which is the core of every "real" imperative programming languages.

[4] And worked in [29] on an interesting Galois theory of algorithms.
[5] Gurevich implemented ASMs via AsmL (see [10] for a comparaison).

Q: Jones' `While` language contains only updates, `if` and `while` commands, and uses only lists as data structure. What is the difference with your version ?

A: I studied algorithmic completeness up to data structures: my `While` language will contain the same data structures as the simulated abstract state machine, to respect the oracular nature[6] of algorithms and free us from particular technological implementations. My result is on control structures: sequences, updates, `if` and `while` commands are sufficient to simulate every sequential algorithm. So every usual imperative programming language is algorithmically complete, at least for algorithms using data structures available in the considered language.

1. Sequential Algorithms (`Algo`)

In [17] Gurevich introduced an axiomatic presentation of the sequential algorithms, giving three postulates:

Gurevich's Thesis 1. Every sequential algorithm satisfies:

- Postulate 1: Sequential Time
- Postulate 2: Abstract States
- Postulate 3: Bounded Exploration

So let `Algo` be the set of the objects satisfying the following three postulates.

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 1. Two sequential algorithms A and B are the same (see [7]) if:

- $S(A) = S(B)$
- $I(A) = I(B)$
- $\tau_A = \tau_B$

In the following, it will be denoted by $A = B$.

[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.

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 said final if $\tau_A(X_m) = X_m$, indeed in that case the execution is $X_0, X_1, X_2, ... X_m, X_m, ...$ so the execution is considered stopped at the state X_m.

An execution is said **terminal** if it contains a final state. Of course, if an execution is terminal then the final state is unique.

Let $\mathbf{time}(A, X) = min\{i \in \mathbb{N} \ ; \ \tau_A^i(X) = \tau_A^{i+1}(X)\}$, where f^i is the iteration of f defined by $f^0 = id$ and $f^{i+1} = f(f^i)$.

Remark 2. Two algorithms A and B have the same set of executions if:

- $I(A) = I(B)$
- $\tau_A = \tau_B$

In this case $S(A) \neq S(B)$ can occurs only for the unreachable states. The condition $S(A) = S(B)$ can be seen as unnecessary, but it does not matter in the following.

For the postulate 2, the memory of the execution will be formalized by a (first-order) **structure** X given by:

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

These notations are from [12].

To have a uniform presentation Gurevich considers constant symbols of the language as 0-ary function symbols, and relation symbols R as their indicator function χ_R ($R(\vec{a})$ iff $\chi_R(\vec{a}) = true$), so every symbol in $\mathcal{L}_X$ is a function. Partial functions can be implemented with a special value $undef$.

In the following, 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 is a claim assuming that every data structure can be formalized as a first-order structure [7]. Because the states are independent from their implementation (for example the name of the objects), isomorphic states are considered equivalent:

Postulate 2. (Abstract States)

[7] I will discuss in the conclusion of a constructive Postulate 2 for usual data structures (integers, words, lists, arrays and graphs) but this is not the point of this article.

- The states of an algorithm A are first-order structures.

  The states of A have the same (finite) language $\mathcal{L}_A$ and the same universe $\mathcal{U}_A$[8].

- $S(A)$ 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. So, the interpretation of the static symbols is fixed by the initial state.

Moreover I will distinguish the constructors whose interpretation is uniform for every initial state, and the parameters. The symbols depending on the initial state are the dynamic symbols and the parameters: they are the **inputs**. The constructors and their interpretation are the representation of data structures.

The logical variables will not be used in this paper: every term will be closed and every formula will be closed and 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
- $a_1, ... \ a_k, b \in \mathcal{U}_A$

$(f, a_1, ... \ a_k)$ is a location of X and $(f, a_1, ... \ a_k, b)$ is an **update** on X at the location $(f, a_1, ... \ a_k)$.

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

- $\overline{f}^{X+u}(\vec{a}) = b$ if $u = (f, \vec{a}, b)$
- $\overline{f}^{X+u}(\vec{a}) = \overline{f}^{X}(\vec{a})$ else

If $\overline{f}^{X}(\vec{a}) = b$ then the update $(f, \vec{a}, b)$ is said trivial in X, because nothing has changed: 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 said consistent if it does not contain two distinct updates with the same location. Indeed, if $(f, \vec{a}, b), (f, \vec{a}, b') \in \Delta$ and $b \neq b'$ then this set tries to update $f(\vec{a})$ with two distinct values. In that case, the entire set of updates clashes and nothing is done:

- $\overline{f}^{X+\Delta}(\vec{a}) = b$ if $(f, \vec{a}, b) \in \Delta$ and Δ is consistent
- $\overline{f}^{X+\Delta}(\vec{a}) = \overline{f}^{X}(\vec{a})$ else

[8]In fact in [17] only the language is the same for the states, and Gurevich only assumes that τ_A does not change the universe of a state. But the difference between two initial states is the interpretation of the inputs, so I assume that the universe is the union of every possible structure for the algorithm, to get a simpler presentation.

If X and Y are two states of the same algorithm A then there exists a unique consistent set Δ of non trivial updates such that $Y = X + \Delta$. This Δ is the **difference** between the two sets and is denoted $Y - X$. Indeed, because X and Y have the same language and the same base set they have the same locations, so let Δ be the set $\{(f, \vec{a}, \overline{f}^Y(\vec{a}))\ ;\ \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\}$.

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

The first and the second postulates are not sufficient: only local and bounded changes and explorations are reasonable for each step of a sequential algorithm[9]. So only a bounded number of terms must be read or updated during a step of the execution:

Postulate 3. (Bounded Exploration)

For each algorithm A there is a finite set T of terms such that if two states X and Y coincide over $Sub(T)$ then $\Delta(A, X) = \Delta(A, Y)$, where:

- X and Y coincide over T means that the terms of T have the same interpretation in X and Y
- $Sub(T)$ is the set of the subterms of T

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

Gurevich proved in [17] that if $(f, a_1, ... a_k, b) \in \Delta(A, X)$ then $a_1, ... a_k, b$ are interpretations in X of terms in $sub(T)$. So because T is finite there exists a bounded number of $a_1, ... a_k, b$ such that $(f, a_1, ... a_k, b) \in \Delta(A, X)$. Moreover because $\mathcal{L}_A$ is finite there exists a bounded number of dynamic symbols f. So $\Delta(A, X)$ has a bounded number of elements, and for each step of the algorithm only a bounded amount of work is done.

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 the 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, for example:

Example 1.1. (The Lambda-Calculus [10])

(Syntax of the Programs) $\quad t := x \mid \lambda x.t \mid (t_1)t_2$

(The β-reduction) $\quad (\lambda x.t_1)t_2 \to_\beta t_1[t_2/x]$

To be deterministic the strategy of the transition system must be specified, for example the call-by-name strategy defined by context:

(Call-by-Name Context) $\quad C_n\{.\} := . \mid C_n\{.\}t$

(Transition Rule) $\quad C_n\{(\lambda x.t_1)t_2\} \to_n C_n\{t_1[t_2/x]\}$

This rule can be implemented in a machine:

[9] I give an example p.9 with the parallel lambda-calculus.

[10] The notations and the machine are from Krivine's [22].

(Operational semantics) $t_1 t_2 \star \pi \succ_0 t_1 \star t_2, \pi$

$\lambda x.t_1 \star t_2, \pi \succ_1 t_1[t_2/x] \star \pi$

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. $\succ_i$ represents i steps of calculus, so here only substitutions have a cost, not explorations inside a term, like for 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 elementary operation this model satisfies postulate 3, because only one term is pushed or popped per step. But the lambda-calculus with parallel reductions does not, for example with the term $t = \lambda x.(x)x(x)x$ applied to itself: $(t)t \to_p (t)t(t)t \to_p (t)t(t)t(t)t(t)t \to_p (t)t(t)t(t)t(t)t(t)t(t)t(t)t(t)t \to_p$ .... Indeed, at the step i exactly 2^{i-1} β-reduction are done, which is unbounded.

Sometimes the identity between two models of computation can be proven. For example, Serge Grigorieff and Pierre Valarcher proved in [15] that formal classes of the EMAs (a variant of the Gurevich's ASMs) not only simulate step by step but can be identified to Turing Machines, Random Access Machines or other sequential models of computation. But generally only a simulation can be proven between two models of computation.

In this framework, a computation model M_1 can simulate an other 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 of P_2. What can be used in a fair simulation is detailed in the following two examples:

Example 1.2. (Temporary Variables)

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

$\{i := 0;$ `while` $i < n$ $\{s;\ i := i + 1;\};\}$[11]

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

The language $\mathcal{L}_1$ of the simulating program must be bigger than the language $\mathcal{L}_2$ of the simulated program. But new function symbols could be too powerful, for example a unary symbol *env* containing every value of the environment.

To get a fair simulation, I assume that $\mathcal{L}_1 \setminus \mathcal{L}_2$ is a set containing only a bounded number of variables. The initial values of these fresh variables could be a problem if they depend on the inputs. So in this paper we will use an initialization depending[12] only of the constructors[13]. Because this initialization will be independent (up to isomorphism) from the initial states, we will call it a **uniform initialization**.

So, structures of the simulating program P_1 will be defined with the language $\mathcal{L}_1$, and the structures of the simulating program P_2 will be defined with the language $\mathcal{L}_2 \subseteq \mathcal{L}_1$.

[11] This syntax will be defined p.15.

[12] The values of the fresh variables in the initial states can also be irrelevant, for example p.25 in the program P_Π the variables $\vec{v}$ are explicitly updated with the value of the terms $\vec{t}$ before being read.

[13] For example p.22 in the program Π_P the boolean variable b_P is initialized with true and the other with false.

Notation 1. $X|_{\mathcal{L}_2}$ from [12] is the **restriction** of the $\mathcal{L}_1$-structure X to the language $\mathcal{L}_2$, defined by:

- The language is $\mathcal{L}_2$.
- The universe is the same.
- For every $s \in \mathcal{L}_2$: $\overline{s}^{X|_{\mathcal{L}_2}} = \overline{s}^X$

This notation will be extended to set of updates: $\Delta|_{\mathcal{L}} =_{def} \{(f, \vec{a}, b) \in \Delta \ ; \ f \in \mathcal{L}\}$. Notice that $(X + \Delta)|_{\mathcal{L}} = X|_{\mathcal{L}} + \Delta|_{\mathcal{L}}$.

Example 1.3. (Temporal Dilation)
At each 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

The notion of elementary action is arbitrary. In this case they can be seen as distinct actions that require three steps of calculus (model M_1) or as one single multi-action that requires only one step of calculus (model M_2). So, one action in M_2 requires three actions of M_1.

Let $\vec{X}$ be an execution $X_0, X_1, X_2, X_3, X_4, X_5, X_6, ...$ of M_1, and let $\vec{Y}$ be the black execution: for every i $Y_i = X_{3\times i}$. $\vec{Y}$ is an execution of M_2.

Imagine that M_1 and M_2 are implemented on real machines such that M_1 is three times faster than M_2. In that case if an external observer starts the two machines at the same time and looks at their states at every step of M_2 then the two machines cannot be distinguished.

The time unit is arbitrary.

In the following a (constant) temporal dilation d will be allowed: the simulation is said step-by-step, and strictly step-by-step if $d = 1$. Contrary to the previous example this constant may depend on the simulated program.

But sadly this temporal dilation is not sufficient to ensure the termination of the simulation. Indeed, the simulated execution $Y_0, ... Y_m, Y_m, ...$ could have finished, but the simulating simulation may have not:

$X_0, ... X_{md}, X_{md+1}, ... X_{md+(d-1)}, X_{md}, X_{md+1}, ...$

So an ending condition like $time(A, X) = d \times time(B, X) + e$ is necessary. In the following I consider only models of computation with idle moves[14], so with $d' = max(d, e)$ it is possible to end the simulating program at the same time (up to temporal dilation) than the simulated program.

[14]If the temporal dilation is $\leq d$ instead of $= d$ then the idle moves (`skip` commands) are not necessary in this paper. This weakened simulation respects also the time complexity.

Definition 1.4. (Fair Simulation)
Let M_1, M_2 be two computation models.
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)

   There exists $d \in \mathbb{N}^\star$ and $e \in \mathbb{N}$ (depending only of P_2) such that
   for every execution $\vec{Y}$ of P_2 there exists an execution $\vec{X}$ of P_1 such that:

2. for 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 said **algorithmically equivalent**, denoted by $M_1 \simeq M_2$.

Remark 3. If the simulated execution requires n steps of calculus then the simulating execution requires $O(n)$ steps, so the simulation respects the time complexity. Moreover $Y_i = X_{d\times i}|_{\mathcal{L}(P_2)}$ 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 a model of computation M will be said **algorithmically complete** if $M \simeq$ `ASM`.

For example, in [13], Marie Ferbus-Zanda and Serge Grigorieff proved that the lambda-calculus is algorithmically complete up to the oracular nature of the algorithms: constant symbols must be added in the lambda-calculus to get the same language than the ASMs.

I will use the same method in this paper. Taking Jones' `While` language as core for imperative languages, I will prove in the next section the algorithmic completeness of `While` via a bisimulation between `ASM` and `While`, using the same data structures in these two models of computation.

Abstract State Machines (`ASM`)

Because the constructors have a uniform interpretation, for a simpler presentation in the following I will not distinguish their syntax from their semantics. The Gurevich's Abstract State Machines (`ASM`) require only:

- The equality $=$.
- The booleans: the constants $true$ and $false$, the unary operation $\neg$ and the binary operations $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$[15].

[15] All these symbols are not necessary because they can be simulated. And indeed only $\neg$ and $\wedge$ are necessary for formulas page 13.

Definition 2.1. (`ASM` programs)

$\Pi =_{def} f t_1 ... t_k := t_0$
$\quad\quad$ | `if` F `then` Π_1 `else` Π_2 `endif`
$\quad\quad$ | `par` $\Pi_1 \| ... \| \Pi_n$ `endpar`

where:

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

Notation 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 will write only `if` F `then` Π `endif`.

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

- $Read(f t_1 ... t_k := t_0) = \{t_1, ... \ t_k, t_0\}$
- $Read($`if` F `then` Π_1 `else` Π_2 `endif`$) = \{F\} \cup Read(\Pi_1) \cup Read(\Pi_2)$
- $Read($`par` $\Pi_1 \| ... \| \Pi_n$ `endpar`$) = Read(\Pi_1) \cup ... \cup Read(\Pi_n)$

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

- $Upd(f t_1 ... t_k := t_0) = \{f t_1 ... t_k\}$
- $Upd($`if` F `then` Π_1 `else` Π_2 `endif`$) = Upd(\Pi_1) \cup Upd(\Pi_2)$
- $Upd($`par` $\Pi_1 \| ... \| \Pi_n$ `endpar`$) = Upd(\Pi_1) \cup ... \cup Upd(\Pi_n)$

An `ASM` program Π induces a transition function $\tau_\Pi(X) = X + \Delta(\Pi, X)$, where the set of updates $\Delta(\Pi, X)$ is defined by induction:

Definition 2.2. (Operational Semantics of ASMs)

- $\Delta(f t_1 ... t_k := t_0, X) = \{(f, \overline{t_1}^X, ... \ \overline{t_k}^X, \overline{t_0}^X)\}$
- $\Delta($`if` F `then` Π_1 `else` Π_2 `endif`$, X) = \Delta(\Pi_i, X)$
  where $i = 1$ if $\overline{F}^X = true$ and $i = 2$ if $\overline{F}^X = false$
- $\Delta($`par` $\Pi_1 \| ... \| \Pi_n$ `endpar`$, X) = \Delta(\Pi_1, X) \cup ... \cup \Delta(\Pi_n, X)$

The semantics of the `par` is a set of updates done simultaneously, contrary to the imperative language on the following subsection that is strictly sequential.

Remark 4. If X and Y coincide over $Read(\Pi)$ then: $\Delta(\Pi, X) = \Delta(\Pi, Y)$

The finiteness of the set T in the third postulate allows (see [17]) to write a finite `ASM` program Π_A with the same set of updates than A for every its states:

Proposition 2.3. (Algorithmic Completeness of the ASMs)
For every sequential algorithm A there exists an `ASM` program Π_A of the same language such that for every state X of A: $\Delta(\Pi_A, X) = \Delta(A, X)$.

Proof:
Let X be a state of A.

Gurevich proved in [17] that if $(f, a_1, ... a_k, b) \in \Delta(A, X)$ then $a_1, ... a_k, b$ are interpretations $\overline{t_1}^X, ... \overline{t_k}^X, \overline{t_0}^X$ of terms in $sub(T)$.

Let Π_X be the `ASM` program:
`par` $f_1^X(\vec{t_1^X}) := t_1^X \ \| f_2^X(\vec{t_2^X}) := t_2^X \ ... \ \| f_{m_X}^X(\vec{t_{m_X}^X}) := t_{m_X}^X$ `endpar`
such that $\Delta(\Pi_X, X) = \Delta(A, X)$.

Let E_X be the binary relation on terms of $sub(T)$ such that $E_X(t_1, t_2)$ is $true$ if and only if $\overline{t_1}^X = \overline{t_2}^X$.

Gurevich proved in [17] that if X and Y are states of A such that $E_X = E_Y$, then $\Delta(\Pi_X, Y) = \Delta(A, Y)$.

By postulate 3 $sub(T)$ is finite, so there exists only a finite number $E_{X_1}, ... E_{X_c}$ of these relations.

Let $F_i =_{def} \wedge_{j \neq k}(t_j \ \epsilon_{i,j,k} \ t_k)$, where $\epsilon_{i,j,k}$ is $=$ if $E_{X_i}(t_j, t_k)$ is $true$, and $\epsilon_{i,j,k}$ is $\neq$ if $E_{X_i}(t_j, t_k)$ is $false$.

So $\overline{F_i}^X = true$ if and only if $E_X = E_{X_i}$.

Π_A is the program:
`par if` F_1 `then` Π_{X_1} `endif`
$\|$`if` F_2 `then` Π_{X_2} `endif`
$\vdots$
$\|$`if` F_c `then` Π_{X_c} `endif`
`endpar` □

Remark 5. The F_i are guards: for every state X one and only one F_i is $true$.

Moreover, because $\Delta(\Pi_A, X) = \Delta(A, X) = \tau_A(X) - X$, $\Delta(\Pi_A, X)$ is consistent without trivial updates.

Such an `ASM` program Π_A is said in **normal form**.

Definition 2.4. 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 $S(M)$

Let `ASM` be the set of the Abstract States Machines.

Gurevich's Theorem 1. `Algo` = `ASM`

Proof:
Let A be a sequential algorithm.

Let M be the ASM given by:

- the `ASM` program Π_A
- the set $S(M) = S(A)$
- the subset $I(M) = I(A) \subseteq S(A) = S(M)$
- the application τ_M is τ_{Π_A} restricted to $S(A)$

For every $X \in S(A)$:
$\tau_A(X)$
$= X + (\tau_A(X) - X)$
$= X + \Delta(A, X)$
$= X + \Delta(\Pi_A, X)$
$= \tau_{\Pi_A}(X)$
$= \tau_M(X)$
So $A = M$ (see p.5)

On the other way, by definition an ASM M satisfies the two first postulates, and its exploration witness is $Read(\Pi) \cup Upd(\Pi)$.[16]

So M is a sequential algorithm. □

So Gurevich proved that his axiomatic presentation for the sequential algorithms defines the same objects than his operational presentation of the Abstract State Machines.

Remark 6. 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`)

I define a variant of the Neil Jones' `While` (see [20]) language, because this language is minimal: the programs are only sequentialized updates, `if` or `while` commands. So, if `While` is algorithmically complete then every imperative language containing these control structures (including usual programming languages like `C`, `Java` or `Python`) will be algorithmically complete too.

The difference with the Neil Jones' `While` is that the data structures are not fixed. Like for the ASMs the equality and the booleans are needed, but the other data structures will be seen as oracular: if they can be implemented in a sequential algorithm they will be implemented using the same language, universe and interpretation in this programming language. So, the fair simulation between `ASM` and `While` will be proved for the control structures, up to data structures.

[16] And not only $Read(\Pi)$ because the update of a command could be trivial.

Definition 2.5. (Syntax of the `While` programs)

$$\begin{aligned}
\text{(commands)}\ c &=_{def} ft_1...t_k := t_0\\
&\mid \texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}\\
&\mid \texttt{while}\ F\ \{c; s\}\\
\text{(sequences)}\ s &=_{def} \epsilon \mid c; s\\
\text{(programs)}\ P &=_{def} \{s\}
\end{aligned}$$

where:

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

Notation 3. The symbol ϵ denotes the empty sequence. For simplicity, the empty program will be written $\{\}$ instead of $\{\epsilon\}$.

As for the `ASM` programs, if the `else` part of an `if` is empty I will write only `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 extended by induction to sequence of sequences $s_1; s_2$ by $\epsilon; s_2 = s_2$ and $(c; s_1); s_2 = c; (s_1; s_2)$.

Like for the example 1.1 p.8 the operational semantics of this `While` programming language will be formalized by a state transition system, where a state of the system is a pair $P \star X$ of a `While` program and a structure, and a transition is determined only by the head command and the current structure:

Definition 2.6. Operational semantics of the `While` programs:

$$\begin{aligned}
\{ft_1...t_k := t_0; s\} \star X &\succ \ \{s\} \star X + (f, \overline{t_1}^X, ...\ \overline{t_k}^X, \overline{t_0}^X)\\
\{\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}; s_3\} \star X &\succ \ \{s_1; s_3\} \star X \text{ if } \overline{F}^X = true\\
\{\texttt{if}\ F\ \{s_1\}\ \texttt{else}\ \{s_2\}; s_3\} \star X &\succ \ \{s_2; s_3\} \star X \text{ if } \overline{F}^X = false\\
\{\texttt{while}\ F\ \{s_1\}; s_2\} \star X &\succ \ \{s_1; \texttt{while}\ F\ \{s_1\}; s_2\} \star X \text{ if } \overline{F}^X = true\\
\{\texttt{while}\ F\ \{s_1\}; s_2\} \star X &\succ \ \{s_2\} \star X \text{ if } \overline{F}^X = false
\end{aligned}$$

The successors are unique, so this transition system is **deterministic**. A succession of transition steps $\succ_n$ is defined by induction on i :

- $P_1 \star X_1 \succ_0 P_2 \star X_2$ if $P_1 = P_2$ and $X_1 = X_2$
- $P_1 \star X_1 \succ_{i+1} P_2 \star X_2$ if there exists $P_3 \star X_3$ such that $P_1 \star X_1 \succ P_3 \star X_3$ and $P_3 \star X_3 \succ_i P_2 \star X_2$

Remark 7. $\succ_0$ is $=$ and $\succ_1$ is $\succ$.

By induction on i 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: they are the terminating states. I 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 $\overline{F}^X = true$ then $\{\texttt{while } F\ \{\};s\} \star X \succ \{\texttt{while } F\ \{\};s\} \star X$

So it should be considered as a terminal state too. Because this problem will occur in the following simulations, I forbade in definition 2.5 the commands `while` F $\{\}$. So in the following if $P_1 \star X_1 \succ P_2 \star X_2$ then $P_1 \neq P_2$.

Notation 4. 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 state transition system is deterministic i and X' are unique, so X' will be denoted $P(X)$ and i will be denoted $time(P, X)$.

So if $P \downarrow X$ then $P \star X \succ_{time(P,X)} \{\} \star P(X)$.

Notice that $time(P, X) = 0$ iff $P = \{\}$.

A program is said terminal if it terminates on all its structures.

Example 2.7. A program for the minimum of two integers m and n in an execution time of $O(min(m, n))$:

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

where m and n are the inputs and x is the output.

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

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

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

Notation 5. Let s_P be the sequence such that $P = \{s_P\}$. The **composition** P_1P_2 of the `While` 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 $\{s_{P_1};\ s_{P_2}\}$, so to avoid the 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. So cP is a notation for the program $\{c; s_P\}$.

The rules have the form $cP \star X \succ P'P \star X'$ where P is a contextual program. Notice that if there exists P_1 such that $cP_1 \star X \succ P'P_1 \star X'$ then for every P_2: $cP_2 \star X \succ P'P_2 \star X'$. The substitution of P_1 by P_2 is called a **context switch**[17].

Context switches are very useful, but they cannot be extended to $\succ_n$ in any case. For example:

Example 2.8. Let:

[17]The following lemmas must be modified if a rule without context switch is added, for example: $\{\texttt{exit}; s\} \star X \succ \{\} \star X$, but it will not be the case in this paper.

$P_1 = \{i := 0;\ i := i + 1;\}$
$P_2 = \{i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\}$
$P_3 = \{i := 0;\ \texttt{while}\ true\ \{i := i + 1;\};\}$

- In this case:

$$\begin{aligned}
P_1P_2 \star X &= \{i := 0;\ i := i + 1;\ i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X \\
&\succ \{i := i + 1;\ i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 0) \\
&\succ \{i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 1) \\
&= P_2 && \star X + (x, 1)
\end{aligned}$$

- The transitive context switch works:

$$\begin{aligned}
P_1P_3 \star X &= \{i := 0;\ i := i + 1;\ i := 0;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X \\
&\succ \{i := i + 1;\ i := 0;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 0) \\
&\succ \{i := 0;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 1) \\
&= P_3 && \star X + (x, 1)
\end{aligned}$$

- But in that case:

$$\begin{aligned}
P_1P_2 \star X &= \{i := 0;\ i := i + 1;\ i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X \\
&\succ \{i := i + 1;\ i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 0) \\
&\succ \{i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 1) \\
&\succ \{\texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 2) \\
&\succ \{i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 2) \\
&= P_2 && \star X + (x, 2)
\end{aligned}$$

- The transitive context switch does not work:

$$\begin{aligned}
P_1P_3 \star X &= \{i := 0;\ i := i + 1;\ i := 0;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X \\
&\succ \{i := i + 1;\ i := 0;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 0) \\
&\succ \{i := 0;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 1) \\
&\succ \{\texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 0) \\
&\succ \{i := i + 1;\ \texttt{while}\ true\ \{i := i + 1;\};\} && \star X + (x, 0) \\
&= P_2 && \star X + (x, 0) \\
&\neq P_3 && \star X + (x, 2)
\end{aligned}$$

It works in the first case and not in the second case because the execution must not "enter" in P_2, so i must be bounded by $time(P_1, X)$:

Lemma 2.9. (Transitive Context Switch)
Let $P_1 \downarrow X$ and $i \leq time(P_1, X)$.
If there exists P', X' and P_2 such that $P_1P_2 \star X \succ_i P'P_2 \star X'$
then for every P_3: $P_1P_3 \star X \succ_i P'P_3 \star X'$

Proof:
The proof p.33 is made by induction on $time(P_1, X)$. □

But in the following we will only use context switches for $n = time(P_1, X)$:

Corollary 2.10. (Intermediate States)
If $P_1 \downarrow X$ then $P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$

Proof:
$P_1 \downarrow X$
So $P_1 \star X \succ_{time(P_1,X)} \{\} \star P_1(X)$
By lemma 2.9 on the context $\{\}$:
$P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$ □

This corollary is useful to prove that composition of programs works as intended:

Proposition 2.11. (Composition of Programs)
$P_1P_2 \downarrow X$ if and only if $P_1 \downarrow X$ and $P_2 \downarrow P_1(X)$, such that:

- $P_1P_2(X) = P_2(P_1(X))$
- $time(P_1P_2, X) = time(P_1, X) + time(P_2, P_1(X))$

Proof:
Both sides of the equivalence are proven p.35 using:

- lemma A.2: If $P_1P_2 \downarrow X$ then $P_1 \downarrow X$
- corollary 2.10: If $P_1 \downarrow X$ then $P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$
- and the transitivity of the transition

□

In particular, we can now simply prove that every imperative program without loop terminates for every initial state:

Corollary 2.12. (Termination of Programs without `while`)
If P has no `while` command then P is terminal.

Proof:
The proof p.35 is made by induction on P. □

The transition system is deterministic, which means that if $i \leq time(P, X)$[18] then there exists a unique P' and X' such that: $P \star X \succ_i P' \star X'$.

Let $\tau^i_X(P)$ be that P' and $\tau^i_P(X)$ be that X', so $P \star X \succ_i \tau^i_X(P) \star \tau^i_P(X)$.

If $i > time(P, X)$ I assume that $\tau^i_X(P) = \{\}$ and $\tau^i_P(X) = P(X)$, like in the Gurevich's framework.

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

Remark 8. τ_P^i is not a transition function in the sense of the first section, because $\tau_P^i(X) \neq \tau_P \circ ... \circ \tau_P(X)$.

Indeed, if $P_0 \star X_0 \succ P_1 \star X_1 \succ ... \succ P_{i-1} \star X_{i-1} \succ P_i \star X_i$ then $\tau_{P_0}^i(X_0) = X_i = \tau_{P_{i-1}}(X_{i-1}) = ... = \tau_{P_{i-1}} \circ ... \circ \tau_{P_1} \circ \tau_{P_0}(X_0) \neq \tau_{P_0} \circ ... \circ \tau_{P_0}(X_0)$.

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

Definition 2.13. The set of the 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 9. If $P \downarrow X$ then $\Delta(P, X)$ is finite (see lemma A.3 p.36).

P is said without **overwrite** on X if $\Delta(P, X)$ is consistent.

Indeed, for imperative programs there is an overwrite if we set a variable with one value and during the execution we change this value. 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 means that $\Delta(P, X)$ is inconsistent.

Proposition 2.14. (Updates of a Non-Overwriting Program)
If $P \downarrow X$ without overwrite then $\Delta(P, X) = P(X) - X$.

Proof:
The proof p.37 is made by induction on $time(P, X)$:

$\Delta(P, X) = (\tau_P^1(X) - X) \cup \Delta(\tau_X^1(P), X) = (\tau_P^1(X) - X) \cup (P(X) - \tau_P^1(X))$ by induction hypothesis.

If $\tau_P^1(X) - X = \varnothing$ then $\Delta(P, X) = P(X) - \tau_P^1(X) = P(X) - X$.

Else $\tau_P^1(X) - X = \{u\}$ and $\Delta(P, X) = \{u\} \cup (P(X) - (X + u))$

Because $\Delta(P, X)$ is consistent $u \in P(X) - X$, so $\Delta(P, X) = P(X) - X$ by lemma A.4. □

3. Algorithmic Completeness

`ASM` simulates `While`

The intuitive idea for translating `While` programs onto `ASM` programs is to translate separately every command and add a variable to keep the track of the current command[19].

Example 3.1. The program of the example 2.7 p.16:

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

could be translated onto:

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

[19] Programs of this form are called control state ASMs (see [9]).

$\|$`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 10. The number of a line is the length of the program before the current command so numbers of lines are all between 0 and $length(P)$, and $line = length(P)$ is the end of the program. So, instead of one integer only a finite number of booleans $b_0, b_1, ... \, b_{length(P)}$ can be used[20].

This approach has been suggested in [16], and is fitted for a line-based programming language (for example with `goto` instructions) but not the structured language `While`. Indeed, the lines can distinguish two commands which are the same for the operational semantics of `While`:

Example 3.2. (Marked `While`)

$\{$`while` $true$ $\{\underline{x := x + 1};$ `while` $true$ $\{\underline{\underline{x := x + 1}};\};\};\}$
$\succ \{\underline{x := x + 1};$ `while` $true$ $\{\underline{\underline{x := x + 1}};\};$
`while` $true$ $\{\underline{x := x + 1};$ `while` $true$ $\{\underline{\underline{x := x + 1}};\};\};\}$
$\succ \{$`while` $true$ $\{\underline{\underline{x := x + 1}};\};$
`while` $true$ $\{\underline{x := x + 1};$ `while` $true$ $\{\underline{\underline{x := x + 1}};\};\};\}$
$\succ \{\underline{\underline{x := x + 1}};$ `while` $true$ $\{\underline{x := x + 1};$ `while` $true$ $\{\underline{\underline{x := x + 1}};\};\};\}$

The second and fourth programs are the same in the operational semantics of While, which means that there is no difference between starting from one or the other. The same problem occurs for the `if` commands :

$\{$`if` $true$ `then` $\{y := 0;\ \underline{x := x + 1};\}$ `else` $\{y := 1;\ \underline{\underline{x := x + 1}};\};\}$
$\succ \{y := 0;\ \underline{x := x + 1};\}$
$\succ \{\underline{x := x + 1};\}$

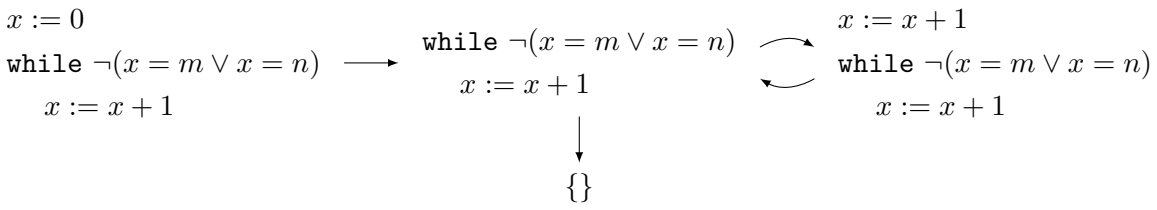
$\{$`if` $false$ `then` $\{y := 0;\ \underline{x := x + 1};\}$ `else` $\{y := 1;\ \underline{\underline{x := x + 1}};\};\}$
$\succ \{y := 1;\ \underline{\underline{x := x + 1}};\}$
$\succ \{\underline{\underline{x := x + 1}};\}$

A translation based on the lines could be defined, but commands must be marked with their depth (for nested `while` commands) and their path (for nested `if` commands) to be distinguished, and still have the same operational semantics. But we do not follow this unnatural and unnecessary approach.

We will not use booleans $b_0, b_1, ... \, b_{length(P)}$ indexed by lines of the program, but instead booleans indexed by 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:

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

Example 3.3. (The Graph of Execution of P_{min})



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

$$\begin{aligned}\mathcal{G}(P_{min}) = \{\ & \{x := 0;\ \texttt{while}\ \neg(x = m \vee x = n)\ \{x := x + 1;\};\}\ ,\\ & \{\texttt{while}\ \neg(x = m \vee x = n)\ \{x := x + 1;\};\}\ ,\\ & \{x := x + 1;\ \texttt{while}\ \neg(x = m \vee x = n)\ \{x := x + 1;\};\}\ ,\\ & \{\}\\ \}\end{aligned}$$

Notation 6. To define a graph of execution I need to introduce the notation $\mathcal{G}(P_1)P_2$. Let $\mathcal{G}$ be a set of `While` programs and P be a `While` program.

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

Remark 11. $card(\mathcal{G}P) = card(\mathcal{G})$
$\mathcal{G}_1P \cup \mathcal{G}_2P = (\mathcal{G}_1 \cup \mathcal{G}_2)P$
$\mathcal{G}_1 \subseteq \mathcal{G}_2 \Rightarrow \mathcal{G}_1P \subseteq \mathcal{G}_2P$

Let P be a `While` program. So $\mathcal{G}(P)$ will be the set of all the possible $\tau^i_X(P)$, not depending on an initial state X:

Definition 3.4. (Graph of Execution)

- $\mathcal{G}(\{\}) = \{\{\}\}$
- $\mathcal{G}(uP) = \{uP\} \cup \mathcal{G}(P)$
- $\mathcal{G}(\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P)$
  $= \{\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P\} \cup \mathcal{G}(P_1)P \cup \mathcal{G}(P_2)P \cup \mathcal{G}(P)$
- $\mathcal{G}(\texttt{while}\ F\ \{P_1\}\ P) = \mathcal{G}(P_1)\ \texttt{while}\ F\ \{P_1\}\ P \cup \mathcal{G}(P)$

Remark 12. By induction on P: $\{\}, P \in \mathcal{G}(P)$

Lemma 3.5. (Finiteness of Graph of Execution)
$card(\mathcal{G}(P)) \leq length(P) + 1$

Proof:
The proof p.40 is made by induction on P, and the length of P is defined in the usual way. □

So only a finite number of guards depending only of P are necessary. Notice that for some programs (like P_{min} in the example 3.3 p.21) not in the case of the example 3.2 p.20, $card(\mathcal{G}(P)) = length(P) + 1$ can be reached.

Proposition 3.6. (Operational Closure of Graph of Execution)

$$\begin{aligned} \text{If } uP' \in \mathcal{G}(P) &\quad \text{then } P' \in \mathcal{G}(P) \\ \text{If } \texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P' \in \mathcal{G}(P) &\quad \text{then } P_1P', P_2P' \in \mathcal{G}(P) \\ \text{If } \texttt{while } F\ \{P_1\}\ P' \in \mathcal{G}(P) &\quad \text{then } P_1 \texttt{ while } F\ \{P_1\}\ P', P' \in \mathcal{G}(P) \end{aligned}$$

Proof:
The proof p.43 is made by cases, using:

- the composition of graphs (lemma A.5): $\mathcal{G}(P_1P_2) = \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$
- the subgraphs (lemma A.6): if $Q \in \mathcal{G}(P)$ then $\mathcal{G}(Q) \subseteq \mathcal{G}(P)$

□

Notation 7. 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 (in particular $X[b_{P_i}]|_{\mathcal{L}_P} = X$).

The proposition 3.6 ensures that the following translation is well defined:

Definition 3.7. (Translation of `While` programs onto ASM)
$\Pi_P =_{def} \texttt{par} \parallel_{P_\mathcal{G} \in \mathcal{G}(P)} \texttt{if } b_{P_\mathcal{G}} \texttt{ then } P_\mathcal{G}^{tr} \texttt{ endpar}$
where:

- $\{\}^{tr} = \texttt{skip}$
- $(uP')^{tr} = \texttt{par } b_{uP'} := false \parallel u \parallel b_{P'} := true \texttt{ endpar}$
- $(\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P')^{tr}$
  $= \texttt{par } b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P'} := false$
  $\parallel \texttt{if } F \texttt{ then } b_{P_1P'} := true \texttt{ else } b_{P_2P'} := true \texttt{ endif endpar}$
- $(\texttt{while } F\ \{P_1\}\ P')^{tr}$
  $= \texttt{par } b_{\texttt{while } F\ \{P_1\}\ P'} := false$
  $\parallel \texttt{if } F \texttt{ then } b_{P_1 \texttt{ while } F\ \{P_1\}\ P'} := true \texttt{ else } b_{P'} := true \texttt{ endif endpar}$

Proposition 3.8. (Step by Step Simulation)
For every $i < time(P, X)$: $\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}]) = \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

Proof:
The proof p.44 is made by case on $\tau_X^i P$ using the fact that the $b_{\tau_X^i(P)}$ are fresh, and that for every $P_\mathcal{G} \in \mathcal{G}(P)$: $\Delta(\Pi_P, X[b_{P_\mathcal{G}}]) = \Delta(P_\mathcal{G}^{tr}, X[b_{P_\mathcal{G}}])$. □

Theorem 3.9. `ASM` simulates `While`

Proof:
The proof is made p.46.

1. $\mathcal{L}_{\Pi_P} = \mathcal{L}_P \cup \{b_{P_\mathcal{G}} \ ; \ P_\mathcal{G} \in \mathcal{G}(P)\}$
   where $card(\{b_{P_\mathcal{G}} \ ; \ P_\mathcal{G} \in \mathcal{G}(P)\}) \leq length(P) + 1$ by lemma 3.5.
2. Using proposition 3.8 I prove by induction on $i \leq time(P, X)$ that:
   $\tau^i_{\Pi_P}(X[b_P]) = \tau^i_P(X)[b_{\tau^i_X(P)}]$
   So $\tau^i_{\Pi_P}(X[b_P])|_{\mathcal{L}_P} = \tau^i_P(X)$
   And the temporal dilation is $d = 1$.
3. If $i = time(P, X)$ then $\tau^i_X(P) = \{\}$
   So $\Delta(\Pi_P, \tau^i_P(X)[b_{\tau^i_X(P)}]) = \varnothing$, and $\tau^{i+1}_{\Pi_P}(X[b_P]) = \tau^i_{\Pi_P}(X[b_P])$
   So $time(\Pi_P, X) \leq time(P, X)$ (1)
   But remember p.16: if $P_1 \star X_1 \succ P_2 \star X_2$ then $P_1 \neq P_2$
   So for every $i < time(P, X)$ $b_{\tau^i_X(P)}$ is updated, so $\tau^{i+1}_{\Pi_P}(X[b_P]) \neq \tau^i_{\Pi_P}(X[b_P])$
   So $time(\Pi_P, X) \geq time(P, X)$ (2)
   By (1) and (2): $time(\Pi_P, X) = time(P, X)$, and $e = 0$.

□

`While` simulates `ASM`

Let Π be an `ASM` program. The aim of this section is to find a `While` program simulating the same executions than Π. Remember than Π is made of three kind of commands: updates, `if` and `par`. Firstly we define a syntactical translation between `ASM` and `While`:

Definition 3.10. (Syntactical Translation of the `ASM` programs)

$$(ft_1...t_k := t_0)^{tr} \text{ is } \{ft_1...t_k := t_0;\}$$
$$(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})^{tr} \text{ is } \{\texttt{if } F \texttt{ then } \Pi_1^{tr} \texttt{ else } \Pi_2^{tr};\}$$
$$(\texttt{par } \Pi_1 \| ... \| \Pi_n \texttt{ endpar})^{tr} \text{ is } \Pi_1^{tr} \ ... \ \Pi_n^{tr} \text{ (composition)}$$

Remark 13. The translation of the `skip` = `par endpar` of the `ASM` programs is the empty program $\{\}$ and not the `skip` = `if` *true* `then` $\{\}$ of the `While` programs.

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 3.11. Let Π be the program `par` $x := y \| y := x$ `endpar` and X be a structure such that $\overline{x}^X = 0$ and $\overline{y}^X = 1$.

$\Delta(\Pi, X) = \{(x,1),(y,0)\}$, so: $\tau_\Pi(X) = X + \{(x,1),(y,0)\}$

But the translation of Π is $\{x := y;\ y := x;\}$, and:

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

In this example the semantics of Π is to swap the value of x and y, but the semantics of Π^{tr} is to erase the value of x by the value of y and leave y unchanged.

To capture the simultaneous behavior of the `ASM` program, we need to save the value of the variables read in the `While` program. For example, if $v = x$ and $w = y$ in X then:

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

which is the correct behavior.

Definition 3.12. (Substitution of a Term by a Variable)

$$\begin{aligned} \{\}[v/t] &\text{ is } \{\} \\ (cP)[v/t] &\text{ is } c[v/t]P[v/t] \end{aligned}$$

where :

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

where :

$$\begin{aligned} t_1[v/t_2] &\text{ is } v \text{ if } t_1 = t_2 \\ &\quad\ t_1 \text{ if } t_1 \neq t_2 \end{aligned}$$

Remark 14. If t_1 and t_2 are distinct terms, P is a program and v_1 and v_2 are fresh distinct variables then $P[v_1/t_1][v_2/t_2] = P[v_2/t_2][v_1/t_1]$.

In particular for k distinct terms $t_1, t_2, ... t_k$ and k fresh distinct variables $v_1, v_2, ... v_k$ the notation $P[\vec{v}/\vec{t}]$ is not ambiguous.

Remind that $Read(\Pi)$ is defined p.12. Let $r = card(Read(\Pi))$ and $t_1, ... t_r$ be the (distinct) terms read by Π. They will be substituted by the fresh variables $v_1, ... v_r$ (each distinct from the others).

According to the Gurevich's Theorem p.13 for every ASM there exists an equivalent (same states and same transition function) ASM in normal form, so I assume that Π is in normal form, and:

$\Pi^{tr}[\vec{v}/\vec{t}] = \{$

`if` v_{F_1} `then` $\{$

$f_1^1(\vec{v_1^1}) := v_1^1;$
$f_2^1(\vec{v_2^1}) := v_2^1;$
$\vdots$
$f_{m_1}^1(\vec{v_{m_1}^1}) := v_{m_1}^1;$
};
`if` v_{F_2} `then` {
$f_1^2(\vec{v_1^2}) := v_1^2;$
$f_2^2(\vec{v_2^2}) := v_2^2;$
$\vdots$
$f_{m_2}^2(\vec{v_{m_2}^2}) := v_{m_2}^2;$
};
$\vdots$
`if` v_{F_c} `then` {
$f_1^c(\vec{v_1^c}) := v_1^c;$
$f_2^c(\vec{v_2^c}) := v_2^c;$
$\vdots$
$f_{m_c}^c(\vec{v_{m_c}^c}) := v_{m_c}^c;$
};
}

We want to simulate one step of Π with a `While` program P_Π.

First, the temporary variables $v_1, ... v_r$ must be initialized with the values of $t_1, ... t_r$.

Second, according to definition 1.4 p.11 the temporal dilation must be constant, depending only of Π and not of the current state.

The conditionals are guards: one and only one F_i is *true* during one step, but the execution of one block requires m_i steps of calculus.

So let $m = max\{m_i \ ; \ 1 \leq i \leq c\}$. We add $m - m_j$ `skip` commands at the end of each block F_j, so the execution of a block will require exactly m steps of computation.

$P_\Pi =_{def}$ {
$v_1 := t_1;$
$v_2 := t_2;$
$\vdots$
$v_r := t_r;$
`if` v_{F_1} `then` {
$f_1^1(\vec{v_1^1}) := v_1^1;$
$f_2^1(\vec{v_2^1}) := v_2^1;$
$\vdots$
$f_{m_1}^1(\vec{v_{m_1}^1}) := v_{m_1}^1;$
`skip`;
$\vdots$ ($m - m_1$ times)

```
        skip;
    };
    if v_{F_2} then {
        f_1^2(v_1^2⃗) := v_1^2;
        f_2^2(v_2^2⃗) := v_2^2;
        ⋮
        f_{m_2}^2(v_{m_2}^2⃗) := v_{m_2}^2;
        skip;
        ⋮ (m − m_2 times)
        skip;
    };
    ⋮
    if v_{F_c} then {
        f_1^c(v_1^c⃗) := v_1^c;
        f_2^c(v_2^c⃗) := v_2^c;
        ⋮
        f_{m_c}^c(v_{m_c}^c⃗) := v_{m_c}^c;
        skip;
        ⋮ (m − m_c times)
        skip;
    };
}
```

Let X be a state of the ASM of Π, enriched with the variables $\vec{v}$. As expected, P_Π simulates one step of Π in a constant time:

Proposition 3.13. (Semantical Translation of the `ASM` programs)
There exists t_Π depending only of 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 proof is p.47:

The initialization requires r steps.

For every fresh variable v_k and for every following state Y: $\overline{v_k}^Y = \overline{t_k}^X$

In particular for every conditional F_j: $\overline{v_{F_j}}^Y = \overline{F_j}^X$

The F_j are guards: one and only one is *true* in X. Let F_i be this formula.

One step is required to enter on the block of v_{F_i}, the other conditionals are erased in $c-1$ steps.

The updates of the block are $\Delta(\Pi, X|_{\mathcal{L}_\Pi})$ and require m_i steps, and the `skip` commands require $m - m_i$ steps.

$\Delta(P_\Pi, X) = \{(\vec{v}, \vec{t}^X)\} \cup \Delta(\Pi, X|_{\mathcal{L}_\Pi})$
By proposition 2.14: $\Delta(P_\Pi, X) = P_\Pi(X) - X$
So $(P_\Pi(X) - X)|_{\mathcal{L}_\Pi} = \Delta(\Pi, X|_{\mathcal{L}_\Pi})$
Moreover $time(P_\Pi, X) = r + c + m$, which depends only of Π. □

Corollary 3.14. $P^i_\Pi(X)|_{\mathcal{L}_\Pi} = \tau^i_\Pi(X|_{\mathcal{L}_\Pi})$

Proof:
The proof p.49 is made by induction on i. □

Remark 15. Because of the initial updates, for every i and k: $\overline{v_k}^{P^{i+1}_\Pi(X)} = \overline{t_k}^{P^i_\Pi(X)}$

With P_Π we have successfully simulated one step of Π in a constant time, not depending on the current state. So, we need to repeat P_Π until enough steps of Π have been simulated.

This is the role of a `while` command but we need a formula F_Π able to detect the end of Π:

Lemma 3.15. (The μ-formula)
Let $F_\Pi =_{def} (v_1 = t_1 \wedge \cdots \wedge v_k = t_k)$.
$time(\Pi, X|_{\mathcal{L}_\Pi}) = min\{i \in \mathbb{N} \ ; \ \overline{F_\Pi}^{P^{i+1}_\Pi(X)} = true\}$

Proof:
Remind that: $time(\Pi, X|_{\mathcal{L}_\Pi}) = min\{i \in \mathbb{N} \ ; \ \tau^i_\Pi(X|_{\mathcal{L}_\Pi}) = \tau^{i+1}_\Pi(X|_{\mathcal{L}_\Pi})\}$

Both sides of $\tau^i_\Pi(X|_{\mathcal{L}_\Pi}) = \tau^{i+1}_\Pi(X|_{\mathcal{L}_\Pi})$ iff $\overline{F_\Pi}^{P^{i+1}_\Pi(X)} = true$ are proven p.49, using the remark p.12 on $Read(\Pi)$. □

F_Π is called a μ-formula because it is similar to the minimization operator μ of the recursive functions (see [12]).

Theorem 3.16. `While` simulates `ASM`

Proof:
The program simulating Π in `While` is $P = P_\Pi$ `while` $\neg F_\Pi$ $\{P_\Pi\}$

1. It contains only $r = card(Read(\Pi))$ fresh variables $\vec{v}$.
2. By lemma 3.15 the execution of this program on X is:
   P_Π `while` $\neg F_\Pi$ $\{P_\Pi\} \star X$
   $\succ_{t_\Pi}$ `while` $\neg F_\Pi$ $\{P_\Pi\} \star P_\Pi(X)$ (corollary 2.10 p. 18)
   $\succ P_\Pi$ `while` $\neg F_\Pi$ $\{P_\Pi\} \star P_\Pi(X)$
   $\succ_{t_\Pi}$ `while` $\neg F_\Pi$ $\{P_\Pi\} \star P^2_\Pi(X)$
   $\succ P_\Pi$ `while` $\neg F_\Pi$ $\{P_\Pi\} \star P^2_\Pi(X)$
   $\vdots$
   $\succ P_\Pi$ `while` $\neg F_\Pi$ $\{P_\Pi\} \star P^{time(\Pi, X|_{\mathcal{L}_\Pi})}_\Pi(X)$

$\succ_{t_\Pi}$ `while` $\neg F_\Pi$ $\{P_\Pi\} \star P_\Pi^{time(\Pi, X|_{\mathcal{L}_\Pi})+1}(X)$

$\succ \{\} \star P_\Pi^{time(\Pi, X|_{\mathcal{L}_\Pi})+1}(X)$ (lemma 3.15)

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

Where $d = t_\Pi + 1$.

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

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

3. $time(P, X)$

   $= (t_\Pi + 1) \times (time(\Pi, X|_{\mathcal{L}_\Pi}) + 1)$

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

   Where $e = t_\Pi + 1$.

□

Conclusion

We have proven `While` $\simeq$ `ASM` which ensures an algorithmic equivalence between imperative programs and abstract state machines.

The cost in space of the simulation is $O(length)$ in the two 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 $d = card(Read(\Pi)) + c + m + 1$ steps to simulate one step of an ASM Π, where 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`: they are algorithmically equivalent but `ASM` remains stronger.

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

$$\begin{aligned}
\{(f_1\vec{t_1}, ... \, f_k\vec{t_k}) := (t_1, ... \, t_k); s\} \star X \succ_1 \; & \{s\} \star X + \{(f_1, \vec{t_1}^X, \overline{t_1}^X), ... \, (f_k, \vec{t_k}^X, \overline{t_k}^X)\} \\
\{\texttt{if } F \, \{s_1\} \texttt{ else } \{s_2\}; s_3\} \star X \succ_0 \; & \{s_1; s_3\} \star X \text{ if } \overline{F}^X = true \\
\{\texttt{if } F \, \{s_1\} \texttt{ else } \{s_2\}; s_3\} \star X \succ_0 \; & \{s_2; s_3\} \star X \text{ if } \overline{F}^X = false \\
\{\texttt{while } F \, \{s_1\}; s_2\} \star X \succ_0 \; & \{s_1; \texttt{while } F \, \{s_1\}; s_2\} \star X \text{ if } \overline{F}^X = true \\
\{\texttt{while } F \, \{s_1\}; s_2\} \star X \succ_0 \; & \{s_2\} \star X \text{ if } \overline{F}^X = false
\end{aligned}$$

This stronger `While` can still be fairly simulated by abstract state machines[21], and $P = P_\Pi$ `while` $\neg F_\Pi$ $\{P_\Pi\}$ can simulate an ASM program Π in normal form with a temporal dilation $d = 1$ (the simulation is sttrictly step-by-step), where P_Π is:

`if` F_1 `then` $\{(v_1^1, \dots\ v_{m_1}^1, f_1^1\vec{t}_1^1, \dots\ f_{m_1}^1\vec{t}_{m_1}^1) := (t_1^1, \dots\ t_{m_1}^1, t_1^1, \dots\ t_{m_1}^1);\}$;
`if` F_2 `then` $\{(v_1^2, \dots\ v_{m_2}^2, f_1^2\vec{t}_1^2, \dots\ f_{m_2}^2\vec{t}_{m_2}^2) := (t_1^2, \dots\ t_{m_2}^2, t_1^2, \dots\ t_{m_2}^2);\}$;
$\vdots$
`if` F_c `then` $\{(v_1^c, \dots\ v_{m_c}^c, f_1^c\vec{t}_1^c, \dots\ f_{m_c}^c\vec{t}_{m_c}^c) := (t_1^c, \dots\ t_{m_c}^c, t_1^c, \dots\ t_{m_c}^c);\}$;

But this model for imperative programming language is not usual, and the theorem is stronger with a minimal core for imperative behavior like Jones' `While` language.

Contrary to our `While` which allows only a step-by-step simulation, this stronger `While` allows a strictly step-by-step simulation, which is the identity[22] of executions up to fresh variables.

So, the algorithmic difference between `ASM` and `While` does not really lie on control structures. But, remember that I proved my simulation up to data structures: I assumed an identity between data structures of `ASM` and `While`. But according to Postulate 2 these data structures are first-order structures, which can hardly be seen as "real" data structures.

The remaining problems are:

- Are these first-order structures are equivalent to data structures used in computer programming, or is there an example of structure of one model that can not faithfully be represented in the other?
- In particular, is it possible to fairly characterize usual data types (like integers, words, lists, arrays and graphs) in this logical framework, in other words to get a constructive postulate 2?
- What is the "size" of an element, or the "cost" of an operation? How to characterize classes of algorithms depending of their cost (in space or time) for any model of computation?

[21] Using one boolean per tuple of updates, and combinations of the original conditionals depending on the path between two tuples.

[22] Serge Grigorieff and Pierre Valarcher worked in [15] with their Evolving Multialgebras (EMAs) on a stronger notion: natural classes of EMAs correspond via "literal identification" to slight extension of usual models of computation.

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 *: The Underlying Logic of Hoare Logic*, Bull. of the Euro. Assoc. for Theoretical Computer Science 70 (2000) 82-110

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

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

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

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

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

[10] 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

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

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

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

[14] 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)

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

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

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

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

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

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

[21] Donald Erin Knuth *: Surreal Numbers: How Two Ex-Students Turned on to Pure Mathematics and Found Total Happiness: A Mathematical Novelette*, Addison-Wesley Professional (1974)

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

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

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

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

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

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

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

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

A. Appendix

Composition of Programs

Lemma A.1. (Execution Chains)
For every $i, j \in \mathbb{N}$:
if $P_1 \star X_1 \succ_i P_2 \star X_2$
and $P_1 \star X_1 \succ_{i+j} P_3 \star X_3$
then $P_2 \star X_2 \succ_j P_3 \star X_3$

This proof does not use the rules of the transition system, only its determinism:

Proof:
By induction on i :

- $i = 0$

  In that case $P_1 \star X_1 = P_2 \star X_2$

  So if $P_1 \star X_1 \succ_{0+j} P_3 \star X_3$

  then $P_2 \star X_2 \succ_j P_3 \star X_3$

- $i = 1$

  In that case $P_1 \star X_1 \succ P_2 \star X_2$ (1)

  Because $P_1 \star X_1 \succ_{1+j} P_3 \star X_3$ there exists $P' \star X'$ such that:

  $P_1 \star X_1 \succ P' \star X'$ (2)

  $P' \star X' \succ_j P_3 \star X_3$ (3)

  Because the transition system is deterministic with (1) and (2):

  $P_2 \star X_2 = P' \star X'$

  So with (3):

  $P_2 \star X_2 \succ_j P_3 \star X_3$

- $i \to i + 1$

  In that case $P_1 \star X_1 \succ_{i+1} P_2 \star X_2$ (1)

  Because $P_1 \star X_1 \succ_{i+j+1} P_3 \star X_3$ there exists $P' \star X'$ such that:

  $P_1 \star X_1 \succ P' \star X'$ (2)

  $P' \star X' \succ_{i+j} P_3 \star X_3$ (3)

  Because the case $j = 1$ is true, with (1) and (2):

  $P' \star X' \succ_i P_2 \star X_2$

  But (3), so by induction hypothesis:

  $P_2 \star X_2 \succ_j P_3 \star X_3$

□

Lemma 1. (2.9 p.17)
(Transitive Context Switch)
Let $P_1 \downarrow X$ and $i \leq time(P_1, X)$.
If there exists P', X' and P_2 such that $P_1P_2 \star X \succ_i P'P_2 \star X'$
then for every P_3: $P_1P_3 \star X \succ_i P'P_3 \star X'$

This proof does not use the rules of the transition system, only its determinism and context switch:

Proof:
By induction on $time(P_1, X)$:

- $time(P_1, X) = 0$

  In that case $i = 0$.

  If there exists P', X' and P_2 such that $P_1P_2 \star X \succ_0 P'P_2 \star X'$

  Then: $P_1P_2 = P'P_2$ and $X = X'$

  $P_1P_2 = P'P_2$, so (proof by induction on P_1) $P_1 = P'$

  So for every P_3: $P_1P_3 = P'P_3$

  But $X = X'$, so $P_1P_3 \star X \succ_0 P'P_3 \star X'$

- $time(P_1, X) = t + 1$

  $time(P_1, X) \neq 0$ so $P_1 \neq \{\}$.

  So there exists c and P_1' such that $P_1 = cP_1'$.

  By the transition system there exist P" and X" such that:

  $cP_1'P_2 \star X \succ P"P_1'P_2 \star X"$ (1)

  The case $i = 0$ is the same than above, so I assume that $i = j + 1$.

  By hypothesis $j + 1 \leq time(cP_1', X) = t + 1$, so $j \leq t$.

  If there exists P', X' and P_2 such that:

  $cP_1'P_2 \star X \succ_{j+1} P'P_2 \star X'$

  Then, by lemma A.1 and (1): $P"P_1'P_2 \star X" \succ_j P'P_2 \star X'$ (2)

  By hypothesis $cP_1' \downarrow X$ so: $cP_1' \star X \succ_{time(cP_1', X)} \{\} \star cP_1'(X)$

  But, by context switch on (1):

  $cP_1' \star X \succ P"P_1' \star X"$

  So, because $time(cP_1', X) \geq 1$, by lemma A.1 :

  $P"P_1' \star X" \succ_{time(cP_1', X)-1} \{\} \star cP_1'(X)$

  So $P"P_1' \downarrow X"$ and $time(P"P_1', X") = time(cP_1', X) - 1 = t$.

  $P"P_1' \downarrow X"$ and $j \leq t = time(P"P_1', X")$

So by induction hypothesis on (2), for every P_3:

$P"P'_1P_3 \star X" \succ_j P'P_3 \star X'$

But by context switch on (1), for every P_3:

$cP'_1P_3 \star X \succ P"P'_1P_3 \star X"$

So for every P_3: $cP'_1P_3 \star X \succ_{j+1} P'P_3 \star X'$

So $P_1P_3 \star X \succ_i P'P_3 \star X'$

□

Lemma A.2. (Termination of Prefix)
If $P_1P_2 \downarrow X$ then $P_1 \downarrow X$

This proof does not use the rules of the transition system, only its determinism and context switch:

Proof:
By induction on $time(P_1P_2, X)$:

- $time(P_1P_2, X) = 0$

  In that case $P_1P_2 \star X = \{\} \star P_1P_2(X)$

  So $P_1P_2 = \{\}$, so $P_1 = \{\}$

  And $P_1 \downarrow X$

- $time(P_1P_2, X) = t + 1$

  If $P_1 = \{\}$ then $P_1 \downarrow X$

  Else there exists c and P'_1 such that $P_1 = cP'_1$

  By the transition system there exists P' and X' such that:

  $cP'_1 \star X \succ P'P'_1 \star X'$ (1)

  So, by context switch:

  $cP'_1P_2 \star X \succ P'P'_1P_2 \star X'$

  But by hypothesis $cP'_1P_2 \downarrow X$ and $time(cP'_1P_2, X) = t + 1$ so:

  $cP'_1P_2 \star X \succ_{t+1} \{\} \star cP'_1P_2(X)$

  So, by lemma A.1:

  $P'P'_1P_2 \star X' \succ_t \{\} \star cP'_1P_2(X)$

  So $P'P'_1P_2 \downarrow X'$ with $time(P'P'_1P_2, X') = t$.

  So, by induction hypothesis on t: $P'P'_1 \downarrow X'$

  So by definition $P'P'_1 \star X' \succ_{time(P'P'_1,X)} \{\} \star P'P'_1(X')$

  But (1): $cP'_1 \star X \succ P'P'_1 \star X'$

  So $cP'_1 \star X \succ_{time(P'P'_1,X)+1} \{\} \star P'P'_1(X')$

  And $P_1 \downarrow X$

□

Proposition 1. (2.11 p.18)
(Composition of Programs)
$P_1P_2 \downarrow X$ if and only if $P_1 \downarrow X$ and $P_2 \downarrow P_1(X)$, such that:

- $P_1P_2(X) = P_2(P_1(X))$
- $time(P_1P_2, X) = time(P_1, X) + time(P_2, P_1(X))$

This proof does not use the rules of the transition system, only its determinism and context switch:

Proof:
The proof is made with the pair of statements:

- For $\Rightarrow$ I assume that $P_1P_2 \downarrow X$.
  $P_1P_2 \downarrow X$ so by definition: $P_1P_2 \star X \succ_{time(P_1P_2,X)} \{\} \star P_1P_2(X)$
  $P_1P_2 \downarrow X$ so by lemma A.2: $P_1 \downarrow X$
  So, by corollary 2.10 $P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$
  So $time(P_1P_2, X) \geq time(P_1, X)$, indeed :
  - If $P_2 = \{\}$ then $P_1P_2 = P_1$, so $time(P_1P_2, X) = time(P_1, X)$
  - Else $P_2 \star P_1(X)$ is not a terminal state, so $time(P_1P_2, X) > time(P_1, X)$

  So, by lemma A.1: $P_2 \star P_1(X) \succ_{time(P_1P_2,X)-time(P_1,X)} \{\} \star P_1P_2(X)$
  So $P_2 \downarrow P_1(X)$
  with $P_2(P_1(X)) = P_1P_2(X)$
  and $time(P_2, P_1(X)) = time(P_1P_2, X) - time(P_1, X)$
- For $\Leftarrow$ I assume that $P_1 \downarrow X$ and $P_2 \downarrow P_1(X)$
  $P_1 \downarrow X$ so by corollary 2.10 $P_1P_2 \star X \succ_{time(P_1,X)} P_2 \star P_1(X)$
  $P_2 \downarrow P_1(X)$ so by definition: $P_2 \star P_1(X) \succ_{time(P_2,P_1(X))} \{\} \star P_2(P_1(X))$
  So by transitivity: $P_1P_2 \star X \succ_{time(P_1,X)+time(P_2,P_1(X))} \{\} \star P_2(P_1(X))$
  So $P_1P_2 \downarrow X$
  with $P_1P_2(X) = P_2(P_1(X))$
  and $time(P_1P_2, X) = time(P_1, X) + time(P_2, P_1(X))$

□

Corollary 1. (2.12 p.18)
(Termination of Programs without `while`)
If P has no `while` command then P is terminal.

Proof:
By induction on P:

- If $P = \{\}$ then P is terminal.

  Else there exists c and P' such that $P = cP'$.
  cP' has no `while` command so P' has no `while` command too.
  So, by induction hypothesis P' is terminal.
  Let X be a structure, there is two cases:

- c is an update u

  $uP' \star X \succ P' \star X + u$

  But $P' \star X + u \succ_{time(P', X+u)} \{\} \star P'(X + u)$

  So, by transitivity $uP' \star X \succ_{time(P', X+u)+1} \{\} \star P'(X + u)$

  And $P = uP' \downarrow X$

- c is `if` F $\{P_1\}$ `else` $\{P_2\}$

  $cP' \star X \succ P_i P' \star X$ where $i = 1$ if $\overline{F}^X = true$ and else $i = 2$

  cP' has no `while` command so P_i has no `while` command too.

  So, by induction hypothesis P_i is terminal.

  $P_i \downarrow X$ and $P' \downarrow P_i(X)$, so by proposition 2.11: $P_i P' \downarrow X$

  So $P_i P' \star X \succ_{time(P_i P', X)} \{\} \star P_i P'(X)$

  By transitivity $cP' \star X \succ_{time(P_i P', X)+1} \{\} \star P_i P'(X)$

  So $P =$ `if` F $\{P_1\}$ `else` $\{P_2\}$ $P' \downarrow X$

□

Updates and Overwrites

Lemma A.3. (Finiteness of the Updates)
If $P \downarrow X$ then $card(\Delta(P, X)) \leq time(P, X)$.

Proof:
If $P \downarrow X$ then $\forall i \geq time(P, X)\ \tau_P^i(X) = P(X)$.
So $\forall i \geq time(P, X)\ \tau_P^{i+1}(X) - \tau_P^i(X) = P(X) - P(X) = \varnothing$.
So $\Delta(P, X) = \bigcup_{0 \leq i < time(P,X)} \tau_P^{i+1}(X) - \tau_P^i(X)$.
But $\tau_P^{i+1}(X) - \tau_P^i(X)$ is empty or a singleton, so $card(\tau_P^{i+1}(X) - \tau_P^i(X)) \leq 1$.

$$\begin{aligned} \text{So } card(\Delta(P, X)) &= card(\textstyle\bigcup_{0 \leq i < time(P,X)} \tau_P^{i+1}(X) - \tau_P^i(X)) \\ &\leq \textstyle\sum_{0 \leq i < time(P,X)} card(\tau_P^{i+1}(X) - \tau_P^i(X)) \\ &\leq \textstyle\sum_{0 \leq i < time(P,X)} 1 \\ &= time(P, X). \end{aligned}$$

□

Lemma A.4. For every states X and Y, and for every $u \in Y - X$:
$(Y - (X + \{u\})) \cup \{u\} = Y - X$

Proof:
Let $u = (\varphi, \vec{\alpha}, \beta)$.
Because $u \in Y - X$, $\overline{\varphi}^Y(\vec{\alpha}) = \beta$ and $\overline{\varphi}^Y(\vec{\alpha}) \neq \overline{\varphi}^Y(\vec{\alpha})$.

If $f \neq \varphi$ then $\overline{f}^{X+\{u\}}(\vec{a}) = \overline{f}^X(\vec{a})$
Else if $\vec{a} \neq \vec{\alpha}$ then $\overline{\varphi}^{X+\{u\}}(\vec{a}) = \overline{\varphi}^X(\vec{a})$
Else $\overline{\varphi}^{X+\{u\}}(\vec{\alpha}) = \beta$, so :

$$\begin{aligned}
Y - (X + \{u\}) &= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \ ; \ \overline{f}^Y(\vec{a}) \neq \overline{f}^{X+\{u\}}(\vec{a})\} \\
&= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \ ; \ f \neq \varphi \text{ and } \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \ ; \ \vec{a} \neq \vec{\alpha} \text{ and } \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \ ; \ \overline{\varphi}^Y(\vec{\alpha}) \neq \beta\} \\
(Y - X)\backslash\{u\} &= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u \ ; \ \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\} \\
&= \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u \ ; \ f \neq \varphi \text{ and } \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \neq u \ ; \ \vec{a} \neq \vec{\alpha} \text{ and } \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\} \\
&\sqcup \{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \neq u \ ; \ \overline{\varphi}^Y(\vec{\alpha}) \neq \overline{\varphi}^X(\vec{\alpha})\}
\end{aligned}$$

$f \neq \varphi$ implies that $(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u$, so :

$$\begin{aligned}
&\{(f, \vec{a}, \overline{f}^Y(\vec{a})) \ ; \ f \neq \varphi \text{ and } \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\} \\
&\qquad = \{(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u \ ; \ f \neq \varphi \text{ and } \overline{f}^Y(\vec{a}) \neq \overline{f}^X(\vec{a})\}
\end{aligned}$$

$\vec{a} \neq \vec{\alpha}$ implies that $(f, \vec{a}, \overline{f}^Y(\vec{a})) \neq u$, so :

$$\begin{aligned}
&\{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \ ; \ \vec{a} \neq \vec{\alpha} \text{ and } \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\} \\
&\qquad = \{(\varphi, \vec{a}, \overline{\varphi}^Y(\vec{a})) \neq u \ ; \ \vec{a} \neq \vec{\alpha} \text{ and } \overline{\varphi}^Y(\vec{a}) \neq \overline{\varphi}^X(\vec{a})\}
\end{aligned}$$

Because $\overline{\varphi}^Y(\vec{\alpha}) = \beta$: $\{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \ ; \ \overline{\varphi}^Y(\vec{\alpha}) \neq \beta\} = \varnothing$
Because $(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) = u$: $\{(\varphi, \vec{\alpha}, \overline{\varphi}^Y(\vec{\alpha})) \neq u \ ; \ \overline{\varphi}^Y(\vec{\alpha}) \neq \overline{\varphi}^X(\vec{\alpha})\} = \varnothing$
So $Y - (X + \{u\}) = (Y - X)\backslash\{u\}$.
But $u \in Y - X$, so $(Y - (X + \{u\})) \cup \{u\} = ((Y - X)\backslash\{u\}) \cup \{u\} = Y - X$. □

Proposition 2. (2.14 p.19)
(Updates of a Non-Overwriting Program)
If $P \downarrow X$ without overwrite then $\Delta(P, X) = P(X) - X$.

Proof:
By induction on $time(P, X)$:

- $time(P, X) = 0$
  In that case $P \star X = \{\} \star P(X)$
  So $\Delta(P, X)$
  $= \bigcup_{0 \leq i < time(P,X)} \tau_P^{i+1}(X) - \tau_P^i(X)$
  $= \varnothing$
  $= P(X) - X$

- $time(P, X) = t + 1$
  $time(P, X) \neq 0$ so there exists c and $\tilde{P}$ such that $P = c\tilde{P}$
  By the transition system there exists P' and X' such that:
  $c\tilde{P} \star X \succ P'\tilde{P} \star X'$
  But by hypothesis :
  $c\tilde{P} \star X \succ_{t+1} \{\} \star c\tilde{P}(X)$
  So, by lemma A.1:
  $P'\tilde{P} \star X' \succ_t \{\} \star c\tilde{P}(X)$
  So $P'\tilde{P} \downarrow X'$ with $time(P'\tilde{P}, X') = t$ and $P'\tilde{P}(X') = c\tilde{P}(X)$

  Let $0 \leq i \leq time(P'\tilde{P}, X')$.
  I prove that $\tau^{i+1}_{c\tilde{P}}(X) = \tau^{i}_{P'\tilde{P}}(X')$:
  By definition $P'\tilde{P} \star X' \succ_i \tau^i_{X'}(P'\tilde{P}) \star \tau^i_{P'\tilde{P}}(X')$
  But $c\tilde{P} \star X \succ P'\tilde{P} \star X'$
  So by transitivity $c\tilde{P} \star X \succ_{i+1} \tau^i_{X'}(P'\tilde{P}) \star \tau^i_{P'\tilde{P}}(X')$
  But by definition $c\tilde{P} \star X \succ_{i+1} \tau^{i+1}_{X}(c\tilde{P}) \star \tau^{i+1}_{c\tilde{P}}(X)$
  So, because the transition system is deterministic:
  $\tau^{i+1}_{c\tilde{P}}(X) = \tau^i_{P'\tilde{P}}(X')$

  $\Delta(c\tilde{P}, X)$
  $= \bigcup_{0 \leq i < time(c\tilde{P}, X)} \tau^{i+1}_{c\tilde{P}}(X) - \tau^i_{c\tilde{P}}(X)$
  $= (\tau^1_{c\tilde{P}}(X) - \tau^0_{c\tilde{P}}(X)) \cup \bigcup_{0 \leq i < time(c\tilde{P}, X)-1} \tau^{i+2}_{c\tilde{P}}(X) - \tau^{i+1}_{c\tilde{P}}(X)$
  $= (X' - X) \cup \bigcup_{0 \leq i < time(P'\tilde{P}, X')} \tau^{i+1}_{P'\tilde{P}}(X') - \tau^i_{P'\tilde{P}}(X')$
  $= (X' - X) \cup \Delta(P'\tilde{P}, X')$
  Because $\Delta(c\tilde{P}, X)$ is consistent, $\Delta(P'\tilde{P}, X')$ is consistent too.
  So $P'\tilde{P}$ is without overwrite on X'.

  $P'\tilde{P} \downarrow X'$ without overwrite, and $time(P'\tilde{P}, X') = t$
  So by induction hypothesis on t: $\Delta(P'\tilde{P}, X') = P'\tilde{P}(X') - X'$
  But $P'\tilde{P}(X') = c\tilde{P}(X)$
  So $\Delta(c\tilde{P}, X)$
  $= (X' - X) \cup \Delta(P'\tilde{P}, X')$
  $= (X' - X) \cup (P'\tilde{P}(X') - X')$
  $= (X' - X) \cup (c\tilde{P}(X) - X')$

  Because $X' = \tau^1_{c\tilde{P}}(X)$, there is two cases:

– $X' - X$ is empty.
  In that case $X' = X$
  So $\Delta(c\tilde{P}, X)$
  $= \varnothing \cup (c\tilde{P}(X) - X')$
  $= c\tilde{P}(X) - X$
– $X' - X$ is a singleton $\{u\}$.
  In that case $X' = X + \{u\}$.
  So $\Delta(c\tilde{P}, X)$
  $= (X' - X) \cup (c\tilde{P}(X) - X')$
  $= \{u\} \cup (c\tilde{P}(X) - (X + \{u\}))$

  I prove that $u \in c\tilde{P}(X) - X$:
  Let $u = (f, \vec{a}, b)$.
  $\{u\} = X' - X = \tau^1_{c\tilde{P}}(X) - \tau^0_{c\tilde{P}}(X)$ so $u \in \Delta(c\tilde{P}, X)$.
  $\Delta(c\tilde{P}, X)$ is consistent.
  So there is in $\Delta(c\tilde{P}, X)$ no other update with that location.
  So for every $1 \leq i \leq time(c\tilde{P}, X)$: $\overline{f}^{\tau^i_{c\tilde{P}}(X)}(\vec{a}) = b$.
  In particular $\tau^{time(c\tilde{P},X)}_{c\tilde{P}}(X) = c\tilde{P}(X)$, so $\overline{f}^{c\tilde{P}(X)}(\vec{a}) = b$.
  But $\overline{f}^{X}(\vec{a}) \neq b$ because $X' - X = \{u\}$.
  So $u = (f, \vec{a}, b) \in c\tilde{P}(X) - X$.

  So by lemma A.4 $\Delta(c\tilde{P}, X) = c\tilde{P}(X) - X$

In any case : $\Delta(P, X) = P(X) - X$

□

Graph of Execution

The **length** of a `While` program is defined by induction:

- $length(\{\}) = 0$
- $length(\{c; s\}) = length(c) + length(\{s\})$

where :

- $length(ft_1...t_k := t_0) = 1$
- $length(\texttt{if } F \ \{s_1\} \ \texttt{else} \ \{s_2\}) = 1 + length(\{s_1\}) + length(\{s_1\})$
- $length(\texttt{while } F \ \{s\}) = 1 + length(\{s\})$

Lemma 2. (3.5 p.21)
(Finiteness of Graph of Execution)
$card(\mathcal{G}(P)) \leq length(P) + 1$

Proof:
By induction on P:

- $P = \{\}$
  $\mathcal{G}(\{\}) = \{\{\}\}$
  So $card(\mathcal{G}(P)) = 1 = length(P) + 1$
- $P = uP'$
  $\mathcal{G}(uP') = \{uP'\} \cup \mathcal{G}(P')$
  So $card(\mathcal{G}(P))$
  $\leq 1 + card(\mathcal{G}(P'))$
  $\leq 1 + length(P') + 1$ (by induction hypothesis)
  $= length(P) + 1$
- $P = \texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'$
  $\mathcal{G}(\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P')$
  $= \{\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'\} \cup \mathcal{G}(P_1)P' \cup \mathcal{G}(P_2)P' \cup \mathcal{G}(P')$
  $= \{\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'\} \cup \mathcal{G}(P_1)P' \backslash \{P'\} \cup \mathcal{G}(P_2)P' \backslash \{P'\} \cup \mathcal{G}(P')$
  Because $P' \in \mathcal{G}(P')$, and $\{\} \in \mathcal{G}(P_1), \mathcal{G}(P_2)$ implies $P' \in \mathcal{G}(P_1)P', \mathcal{G}(P_2)P'$ So $card(\mathcal{G}(P))$
  $\leq 1 + (card(\mathcal{G}(P_1)) - 1) + (card(\mathcal{G}(P_2)) - 1) + card(\mathcal{G}(P'))$
  $\leq 1 + length(P_1) + length(P_2) + length(P') + 1$ (by induction hypothesis)
  $= length(P) + 1$
- $P = \texttt{while}\ F\ \{P_1\}\ P'$
  $\mathcal{G}(\texttt{while}\ \{P_1\}\ P') = \mathcal{G}(P_1)\ \texttt{while}\ \{P_1\}\ P' \cup \mathcal{G}(P')$
  So $card(\mathcal{G}(P))$
  $\leq card(\mathcal{G}(P_1)) + card(\mathcal{G}(P'))$
  $\leq 1 + length(P_1) + length(P') + 1$ (by induction hypothesis)
  $= length(P) + 1$

□

Lemma A.5. (Composition of Graphs of Execution)
$\mathcal{G}(P_1P_2) = \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$

Proof:
By induction on P_1:

- $P_1 = \{\}$
  $\mathcal{G}(P_1P_2)$
  $= \mathcal{G}(P_2)$
  $= \{P_2\} \cup \mathcal{G}(P_2)$
  $= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$
- $P_1 = uP_1'$
  $\mathcal{G}(P_1P_2)$
  $= \mathcal{G}(uP_1'P_2)$
  $= \{uP_1'P_2\} \cup \mathcal{G}(P_1'P_2)$
  $= \{uP_1'\}P_2 \cup \mathcal{G}(P_1')P_2 \cup \mathcal{G}(P_2)$ (by induction hypothesis)
  $= (\{uP_1'\} \cup \mathcal{G}(P_1'))P_2 \cup \mathcal{G}(P_2)$
  $= \mathcal{G}(uP_1')P_2 \cup \mathcal{G}(P_2)$
  $= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$
- $P_1 = \texttt{if } F \texttt{ then } \{P'\} \texttt{ else } \{P''\}\ P_1'$
  $\mathcal{G}(P_1P_2)$
  $= \{P_1P_2\} \cup \mathcal{G}(P')P_1'P_2 \cup \mathcal{G}(P'')P_1'P_2 \cup \mathcal{G}(P_1'P_2)$
  $= \{P_1\}P_2 \cup \mathcal{G}(P')P_1'P_2 \cup \mathcal{G}(P'')P_1'P_2 \cup \mathcal{G}(P_1')P_2 \cup \mathcal{G}(P_2)$
  (by induction hypothesis)
  $= (\{P_1\} \cup \mathcal{G}(P')P_1' \cup \mathcal{G}(P'')P_1' \cup \mathcal{G}(P_1'))P_2 \cup \mathcal{G}(P_2)$
  $= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$
- $P_1 = \texttt{while } F\ \{P'\}\ P_1'$
  $\mathcal{G}(P_1P_2)$
  $= \mathcal{G}(\texttt{while } F\ \{P'\}\ P_1'P_2)$
  $= \mathcal{G}(P')\texttt{while } F\ \{P'\}\ P_1'P_2 \cup \mathcal{G}(P_1'P_2)$
  $= \mathcal{G}(P')\texttt{while } F\ \{P'\}\ P_1'P_2 \cup \mathcal{G}(P_1')P_2 \cup \mathcal{G}(P_2)$ (by induction hypothesis)
  $= (\mathcal{G}(P')\texttt{while } F\ \{P'\}\ P_1' \cup \mathcal{G}(P_1'))P_2 \cup \mathcal{G}(P_2)$
  $= \mathcal{G}(\texttt{while } F\ \{P'\}\ P_1')P_2 \cup \mathcal{G}(P_2)$
  $= \mathcal{G}(P_1)P_2 \cup \mathcal{G}(P_2)$

□

Lemma A.6. (Subgraph of Execution)
$Q \in \mathcal{G}(P) \Rightarrow \mathcal{G}(Q) \subseteq \mathcal{G}(P)$

Proof:
By induction on P:

- $P = \{\}$
  $Q \in \mathcal{G}(P) = \{\{\}\}$
  So $Q = \{\} = P$ and $\mathcal{G}(Q) = \mathcal{G}(P)$
- $P = uP'$
  $Q \in \mathcal{G}(P) = \{uP'\} \cup \mathcal{G}(P')$
  So $Q = uP' = P$ or $Q \in \mathcal{G}(P')$
  By cases:
  - If $Q = P$ then $\mathcal{G}(Q) = \mathcal{G}(P)$
  - If $Q \in \mathcal{G}(P')$ then by induction hypothesis $\mathcal{G}(Q) \subseteq \mathcal{G}(P')$
    So $\mathcal{G}(Q) \subseteq \mathcal{G}(P') \subseteq \{uP'\} \cup \mathcal{G}(P') = \mathcal{G}(P)$
- $P = \texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'$
  $Q \in \mathcal{G}(P) = \{\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'\} \cup \mathcal{G}(P_1)P' \cup \mathcal{G}(P_2)P' \cup \mathcal{G}(P')$
  So $Q = \texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P' = P$
  or $Q \in \mathcal{G}(P_1)P'$ or $Q \in \mathcal{G}(P_2)P'$ or $Q \in \mathcal{G}(P')$
  By cases:
  - If $Q = P$ then $\mathcal{G}(Q) = \mathcal{G}(P)$
  - If $Q \in \mathcal{G}(P_1)P'$ then there exists $Q' \in \mathcal{G}(P_1)$ such that $Q = Q'P'$
    By induction hypothesis $\mathcal{G}(Q') \subseteq \mathcal{G}(P_1)$
    So $\mathcal{G}(Q')P' \subseteq \mathcal{G}(P_1)P'$
    By lemma A.5 $\mathcal{G}(Q'P') = \mathcal{G}(Q')P' \cup \mathcal{G}(P')$
    So $\mathcal{G}(Q)$
    $= \mathcal{G}(Q')P' \cup \mathcal{G}(P')$
    $\subseteq \mathcal{G}(P_1)P' \cup \mathcal{G}(P')$
    $\subseteq \{\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'\} \cup \mathcal{G}(P_1)P' \cup \mathcal{G}(P_2)P' \cup \mathcal{G}(P')$
    $= \mathcal{G}(P)$
  - Likewise if $Q \in \mathcal{G}(P_2)P'$
  - If $Q \in \mathcal{G}(P')$ then by induction hypothesis $\mathcal{G}(Q) \subseteq \mathcal{G}(P')$
    So $\mathcal{G}(Q)$
    $\subseteq \mathcal{G}(P')$
    $\subseteq \{\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'\} \cup \mathcal{G}(P_1)P' \cup \mathcal{G}(P_2)P' \cup \mathcal{G}(P')$
    $= \mathcal{G}(P)$
- $P = \texttt{while}\ F\ \{P_1\}\ P'$
  $Q \in \mathcal{G}(P) = \mathcal{G}(P_1)\texttt{while}\ \{P_1\}\ P' \cup \mathcal{G}(P') = \mathcal{G}(P_1)P \cup \mathcal{G}(P')$
  So $Q \in \mathcal{G}(P_1)P$ or $Q \in \mathcal{G}(P')$
  By cases:

– If $Q \in \mathcal{G}(P_1)P$
  then there exists $Q' \in \mathcal{G}(P_1)$ such that $Q = Q'P$
  By induction hypothesis $\mathcal{G}(Q') \subseteq \mathcal{G}(P_1)$
  So $\mathcal{G}(Q')P \subseteq \mathcal{G}(P_1)P$
  By lemma A.5 $\mathcal{G}(Q'P) = \mathcal{G}(Q')P \cup \mathcal{G}(P)$
  So $\mathcal{G}(Q)$
  $= \mathcal{G}(Q')P \cup \mathcal{G}(P)$
  $\subseteq \mathcal{G}(P_1)P \cup \mathcal{G}(P_1)P \cup \mathcal{G}(P')$
  $= \mathcal{G}(P_1)P \cup \mathcal{G}(P')$
  $= \mathcal{G}(P)$
– If $Q \in \mathcal{G}(P')$ then by induction hypothesis $\mathcal{G}(Q) \subseteq \mathcal{G}(P')$
  So $\mathcal{G}(Q)$
  $\subseteq \mathcal{G}(P')$
  $\subseteq \mathcal{G}(P_1)P \cup \mathcal{G}(P')$
  $= \mathcal{G}(P)$

□

Proposition 3. (3.6 p.22)
(Operational Closure of Graph of Execution)

$$\text{If } uP' \in \mathcal{G}(P) \quad \text{then } P' \in \mathcal{G}(P)$$
$$\text{If } \texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\} \; P' \in \mathcal{G}(P) \quad \text{then } P_1P', P_2P' \in \mathcal{G}(P)$$
$$\text{If } \texttt{while } F \; \{P_1\} \; P' \in \mathcal{G}(P) \quad \text{then } P_1 \texttt{ while } F \; \{P_1\} \; P', P' \in \mathcal{G}(P)$$

Proof:
By cases:

- If $uP' \in \mathcal{G}(P)$
  Then $\{uP'\} \cup \mathcal{G}(P')$
  $= \mathcal{G}(uP')$
  $\subseteq \mathcal{G}(P)$ by lemma A.6
  But $P' \in \mathcal{G}(P')$
  So $P' \in \mathcal{G}(P)$
- If $\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\} \; P' \in \mathcal{G}(P)$
  Then $\{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\} \; P'\} \cup \mathcal{G}(P_1)P' \cup \mathcal{G}(P_2)P' \cup \mathcal{G}(P')$
  $= \mathcal{G}(\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\} \; P')$
  $\subseteq \mathcal{G}(P)$ by lemma A.6
  But $P_1P' \in \mathcal{G}(P_1)P'$ and $P_2P' \in \mathcal{G}(P_2)P'$
  So $P_1P', P_2P' \in \mathcal{G}(P)$

- If $\texttt{while}\ F\ \{P_1\}\ P' \in \mathcal{G}(P)$
  Then $\mathcal{G}(P_1)\ \texttt{while}\ F\ \{P_1\}\ P' \cup \mathcal{G}(P')$
  $= \mathcal{G}(\texttt{while}\ F\ \{P_1\}\ P')$
  $\subseteq \mathcal{G}(P)$ by lemma A.6
  But $P_1\ \texttt{while}\ F\ \{P_1\}\ P' \in \mathcal{G}(P_1)\ \texttt{while}\ F\ \{P_1\}\ P'$
  And $P' \in \mathcal{G}(P')$
  So $P_1\ \texttt{while}\ F\ \{P_1\}\ P', P' \in \mathcal{G}(P)$

□

Proposition 4. (3.8 p.22)
(Step by Step Simulation)
For every $i < time(P, X)$: $\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}]) = \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

Proof:
$i < time(P, X)$ so $\tau_X^i(P) \neq \{\}$
By cases on $\tau_X^i(P)$:

- If $\tau_X^i(P) = uP'$
  Then $\tau_P^{i+1}(X) = \tau_P^i(X) + \{\overline{u}^{\tau_P^i(X)}\}$ and $\tau_X^{i+1}(P) = P'$
  $\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \Delta((uP')^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \Delta(\texttt{par}\ b_{uP'} := false \parallel u \parallel b_{P'} := true\ \texttt{endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \{(b_{uP'}, false), \overline{u}^{\tau_P^i(X)[b_{\tau_X^i(P)}]}, (b_{P'}, true)\}$
  $= \{(b_{uP'}, false), \overline{u}^{\tau_P^i(X)}, (b_{P'}, true)\}$ (because $b_{\tau_X^i(P)}$ is fresh)
  So $\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \tau_P^i(X)[b_{\tau_X^i(P)}] + \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \tau_P^i(X)[b_{uP'}] + \{(b_{uP'}, false), \overline{u}^{\tau_P^i(X)}, (b_{P'}, true)\}$
  $= (\tau_P^i(X) + \{\overline{u}^{\tau_P^i(X)}\})[b_{P'}]$
  $= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

- If $\tau_X^i(P) = \texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P'$
  Then $\tau_P^{i+1}(X) = \tau_P^i(X)$ and $\tau_X^{i+1}(P) = P_i P'$
  where $i = 1$ if $\overline{F}^{\tau_P^i(X)} = true$, and else $i = 2$
  $\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \Delta((\texttt{if}\ F\ \texttt{then}\ \{P_1\}\ \texttt{else}\ \{P_2\}\ P')^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$

$= \Delta(\texttt{par } b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P'} := false$
$\| \texttt{ if } F \texttt{ then } b_{P_1P'} := true \texttt{ else } b_{P_2P'} := true \texttt{ endif endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P'}, false), (b_{P_jP'}, true)\}$
where $j = 1$ if $\overline{F}^{\tau_P^i(X)[b_{\tau_X^i(P)}]} = true$, and else $i = 2$
But $b_{\tau_X^i(P)}$ is fresh, so $\overline{F}^{\tau_P^i(X)} = \overline{F}^{\tau_P^i(X)[b_{\tau_X^i(P)}]}$, so $i = j$
So $\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \{(b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P'}, false), (b_{P_iP'}, true)\}$
So $\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\tau_X^i(P)}] + \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
$= \tau_P^i(X)[b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P'}] + \{(b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\}\ P'}, false), (b_{P_iP'}, true)\}$
$= \tau_P^i(X)[b_{P_iP'}]$
$= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

- If $\tau_X^i(P) = \texttt{while } F\ \{P_1\}\ P'$
  Then $\tau_P^{i+1}(X) = \tau_P^i(X)$ and $\tau_X^{i+1}(P) = QP'$
  where $Q = P_1 \texttt{ while } F\ \{P_1\}$ if $\overline{F}^{\tau_P^i(X)} = true$, and else $Q = \{\}$
  $\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \Delta((\texttt{while } F\ \{P_1\}\ P')^{tr}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \Delta(\texttt{par } b_{\texttt{while } F\ \{P_1\}\ P'} := false$
  $\| \texttt{ if } F \texttt{ then } b_{P_1 \texttt{ while } F\ \{P_1\}\ P'} := true \texttt{ else } b_{P'} := true \texttt{ endif endpar}, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \{(b_{\texttt{while } F\ \{P_1\}\ P'}, false), (b_{Q'P'}, true)\}$
  where $Q' = P_1 \texttt{ while } F\ \{P_1\}$ if $\overline{F}^{\tau_P^i(X)[b_{\tau_X^i(P)}]} = true$, and else $Q' = \{\}$
  But $b_{\tau_X^i(P)}$ is fresh, so $\overline{F}^{\tau_P^i(X)} = \overline{F}^{\tau_P^i(X)[b_{\tau_X^i(P)}]}$, so $Q = Q'$
  So $\Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \{(b_{\texttt{while } F\ \{P_1\}\ P'}, false), (b_{QP'}, true)\}$
  So $\tau_{\Pi_P}(\tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \tau_P^i(X)[b_{\tau_X^i(P)}] + \Delta(\Pi_P, \tau_P^i(X)[b_{\tau_X^i(P)}])$
  $= \tau_P^i(X)[b_{\texttt{while } F\ \{P_1\}\ P'}] + \{(b_{\texttt{while } F\ \{P_1\}\ P'}, false), (b_{QP'}, true)\}$
  $= \tau_P^i(X)[b_{QP'}]$
  $= \tau_P^{i+1}(X)[b_{\tau_X^{i+1}(P)}]$

□

Theorem 2. (3.9 p.23)
`ASM` simulates `While`

Proof:
$\mathcal{L}_{\Pi_P} = \mathcal{L}_P \cup \{b_{P_\mathcal{G}} \ ; \ P_\mathcal{G} \in \mathcal{G}(P)\}$
where $card(\{b_{P_\mathcal{G}} \ ; \ P_\mathcal{G} \in \mathcal{G}(P)\}) \leq length(P) + 1$ by lemma 3.5.

I prove that for every $i \leq time(P, X)$: $\tau^i_{\Pi_P}(X[b_P]) = \tau^i_P(X)[b_{\tau^i_X(P)}]$
By induction on i:

- $i = 0$
  $\tau^0_{\Pi_P}(X[b_P]) = X[b_P] = \tau^0_P(X)[b_{\tau^0_X(P)}]$
- $i \to i + 1$
  $\tau^{i+1}_{\Pi_P}(X[b_P])$
  $= \tau_{\Pi_P}(\tau^i_{\Pi_P}(X[b_P]))$
  $= \tau_{\Pi_P}(\tau^i_P(X)[b_{\tau^i_X(P)}])$ by induction hypothesis
  $= \tau^{i+1}_P(X)[b_{\tau^{i+1}_X(P)}]$ by proposition 3.8
  Because $i < i + 1 \leq time(P, X)$

So for every $i \in \mathbb{N}$:
$\tau^i_{\Pi_P}(X[b_P])|_{\mathcal{L}_P}$
$= \tau^i_P(X)[b_{\tau^i_X(P)}]|_{\mathcal{L}_P}$
$= \tau^i_P(X)$
And the temporal dilation is $d = 1$.

If $i = time(P, X)$ then $\tau^i_X(P) = \{\}$
So $\Delta(\Pi_P, \tau^i_P(X)[b_{\tau^i_X(P)}])$
$= \Delta(\{\}^{tr}, \tau^i_P(X)[b_{\tau^i_X(P)}])$
$= \Delta(\texttt{skip}, \tau^i_P(X)[b_{\tau^i_X(P)}])$
$= \varnothing$
So $\tau^{i+1}_{\Pi_P}(X[b_P])$
$= \tau_{\Pi_P}(\tau^i_{\Pi_P}(X[b_P]))$
$= \tau_{\Pi_P}(\tau^i_P(X)[b_{\tau^i_X(P)}])$
$= \tau^i_P(X)[b_{\tau^i_X(P)}] + \Delta(\Pi_P, \tau^i_P(X)[b_{\tau^i_X(P)}])$
$= \tau^i_P(X)[b_{\tau^i_X(P)}]$
$= \tau^i_{\Pi_P}(X[b_P])$
So $time(\Pi_P, X) \leq time(P, X)$ (1)
But remember p.16: if $P_1 \star X_1 \succ P_2 \star X_2$ then $P_1 \neq P_2$.
So for every $i < time(P, X)$ $b_{\tau^i_X(P)}$ is updated, so $\tau^{i+1}_{\Pi_P}(X[b_P]) \neq \tau^i_{\Pi_P}(X[b_P])$
So $time(\Pi_P, X) \geq time(P, X)$ (2)
By (1) and (2) $time(\Pi_P, X) = time(P, X)$, and $e = 0$. □

Translation of a Normal Form Program

Proposition 5. (3.13 p.26)
(Semantical Translation of the ASMs)
There exists t_Π depending only of 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:

Remind that:
$P_\Pi =_{def}$ {
$v_1 := t_1$;
$v_2 := t_2$;
$\vdots$
$v_r := t_r$;
`if` v_{F_1} `then` {
$f_1^1(\vec{v_1^1}) := v_1^1$;
$f_2^1(\vec{v_2^1}) := v_2^1$;
$\vdots$
$f_{m_1}^1(\vec{v_{m_1}^1}) := v_{m_1}^1$;
`skip`;
$\vdots$ ($m - m_1$ times)
`skip`;
};
`if` v_{F_2} `then` {
$f_1^2(\vec{v_1^2}) := v_1^2$;
$f_2^2(\vec{v_2^2}) := v_2^2$;
$\vdots$
$f_{m_2}^2(\vec{v_{m_2}^2}) := v_{m_2}^2$;
`skip`;
$\vdots$ ($m - m_2$ times)
`skip`;
};
$\vdots$
`if` v_{F_c} `then` {
$f_1^c(\vec{v_1^c}) := v_1^c$;
$f_2^c(\vec{v_2^c}) := v_2^c$;
$\vdots$
$f_{m_c}^c(\vec{v_{m_c}^c}) := v_{m_c}^c$;

```
        skip;
        ⋮ (m − m_c times)
        skip;
    };
}
```

The $\vec{v}$ are not occurring in the $\vec{t}$, so after the initialization the state is $X' = X + \{(\vec{v}, \vec{t}^X)\}$, and for every $k \in \{1, ... r\}$: $\overline{v_k}^{X'} = \overline{t_k}^X$.

The remaining program is $\Pi^{tr}[\vec{v}/\vec{t}]$ with `skip` commands, where the updated symbols are the same that in the `ASM` program.

So, because the $\vec{v}$ are fresh they are not updated in the remaining program, and for every following state Y: $\overline{v_k}^Y = \overline{v_k}^{X'}$

So for every $k \in \{1, ... r)\}$ and for every following state Y: $\overline{v_k}^Y = \overline{t_k}^X$

The remaining program is a sequence of `if` commands, where the F_j are guards: for every state Y one and only one F_j is $true$.

Let F_i be the formula $true$ in X.

But for every following state Y and for every conditional F_j: $\overline{v_{F_j}}^Y = \overline{F_j}^X$.

So, during the following states v_{F_i} is $true$ and the other v_{F_j} are $false$, which means that during the execution the block of `if` v_{F_i} is executed, but the other conditional are erased.

These commands are: $f_1^i(\vec{v_1^i}) := v_1^i$; $f_2^i(\vec{v_2^i}) := v_2^i$; ... $f_{m_i}^i(\vec{v_{m_i}^i}) := v_{m_i}^i$; and $m - m_i$ `skip` commands.

Because for every following state Y: $\overline{v_k}^Y = \overline{t_k}^X$, the set of updates induced by the block is $\{(f_1^i, \vec{t_1^i}^X, \overline{t_1^i}^X), (f_2^i, \vec{t_2^i}^X, \overline{t_2^i}^X), ... (f_{m_i}^i, \vec{t_{m_i}^i}^X, \overline{t_{m_i}^i}^X)\} = \Delta(\Pi, X|_{\mathcal{L}_\Pi})$.

So: $\Delta(P_\Pi, X) = \{(\vec{v}, \vec{t}^X)\} \cup \Delta(\Pi, X|_{\mathcal{L}_\Pi})$

$\Delta(\Pi, X|_{\mathcal{L}_\Pi}) = \tau_\Pi(X|_{\mathcal{L}_\Pi}) - X|_{\mathcal{L}_\Pi}$ is consistent.

The $\vec{v}$ are fresh distinct variables, so $\Delta(P_\Pi, X)$ is consistent too.

But by lemma 2.12 P_Π is terminal, so $P_\Pi \downarrow X$ without overwrite.

So by proposition 2.14: $\Delta(P_\Pi, X) = P_\Pi(X) - X$, and:

$(P_\Pi(X) - X)|_{\mathcal{L}_\Pi}$
$= \Delta(P_\Pi, X)|_{\mathcal{L}_\Pi}$
$= (\{(\vec{v}, \vec{t}^X)\} \cup \Delta(\Pi, X|_{\mathcal{L}_\Pi}))|_{\mathcal{L}_\Pi}$
$= \{(\vec{v}, \vec{t}^X)\}|_{\mathcal{L}_\Pi} \cup \Delta(\Pi, X|_{\mathcal{L}_\Pi})|_{\mathcal{L}_\Pi}$
$= \varnothing \cup \Delta(\Pi, X|_{\mathcal{L}_\Pi})$
$= \Delta(\Pi, X|_{\mathcal{L}_\Pi})$

For the execution time:

- The initialization of the $\vec{v}$ requires r steps.
- The conditional before F_i are erased in $i - 1$ steps.
- One step is required to enter on the block of v_{F_i}.
- The updates of the block require m_i steps.
- The `skip` commands require $m - m_i$ steps.

- The conditional after F_i are erased in $c - i$ steps.

So:
$time(P_\Pi, X)$
$= r + (i-1) + 1 + m_i + (m - m_i) + (c - i)$
$= r + c + m$
which depends only of Π. □

Corollary 2. (3.14 p.27)
$P^i_\Pi(X)|_{\mathcal{L}_\Pi} = \tau^i_\Pi(X|_{\mathcal{L}_\Pi})$

Proof:
By induction on i:

- $i = 0$
  $P^0_\Pi(X)|_{\mathcal{L}_\Pi} = X|_{\mathcal{L}_\Pi} = \tau^0_\Pi(X|_{\mathcal{L}_\Pi})$
- $i \to i+1$
  $P^{i+1}_\Pi(X)|_{\mathcal{L}_\Pi}$
  $= P_\Pi(P^i_\Pi(X))|_{\mathcal{L}_\Pi}$
  $= (P^i_\Pi(X) + (P_\Pi(P^i_\Pi(X)) - P^i_\Pi(X)))|_{\mathcal{L}_\Pi}$
  $= P^i_\Pi(X)|_{\mathcal{L}_\Pi} + (P_\Pi(P^i_\Pi(X)) - P^i_\Pi(X))|_{\mathcal{L}_\Pi}$
  $= P^i_\Pi(X)|_{\mathcal{L}_\Pi} + \Delta(\Pi, P^i_\Pi(X)|_{\mathcal{L}_\Pi})$ (proposition 3.13)
  $= \tau^i_\Pi(X|_{\mathcal{L}_\Pi}) + \Delta(\Pi, \tau^i_\Pi(X|_{\mathcal{L}_\Pi}))$ (induction hypothesis)
  $= \tau^{i+1}_\Pi(X|_{\mathcal{L}_\Pi})$

□

Lemma 3. (3.15 p.27)
(The μ-formula)
Let $F_\Pi =_{def} (\wedge \vec{v} = \vec{t})$.
$time(\Pi, X|_{\mathcal{L}_\Pi}) = min_{i \in \mathbb{N}}\{\overline{F_\Pi}^{P^{i+1}_\Pi(X)} = true\}$

Proof:
Remind that: $time(\Pi, X|_{\mathcal{L}_\Pi}) = min_{i \in \mathbb{N}}\{\tau^i_\Pi(X|_{\mathcal{L}_\Pi}) = \tau^{i+1}_\Pi(X|_{\mathcal{L}_\Pi})\}$
I prove that: $\tau^i_\Pi(X|_{\mathcal{L}_\Pi}) = \tau^{i+1}_\Pi(X|_{\mathcal{L}_\Pi})$ iff $\overline{F_\Pi}^{P^{i+1}_\Pi(X)} = true$
$\overline{F_\Pi}^{P^{i+1}_\Pi(X)} = true$
iff for every k $\overline{v_k}^{P^{i+1}_\Pi(X)} = \overline{t_k}^{P^{i+1}_\Pi(X)}$ (definition of F_Π)
iff for every k $\overline{t_k}^{P^i_\Pi(X)} = \overline{t_k}^{P^{i+1}_\Pi(X)}$ (remark p.27)
iff for every k $\overline{t_k}^{P^i_\Pi(X)|_{\mathcal{L}_\Pi}} = \overline{t_k}^{P^{i+1}_\Pi(X)|_{\mathcal{L}_\Pi}}$ ($t_k \in Read(\Pi)$)
iff for every k $\overline{t_k}^{\tau^i_\Pi(X|_{\mathcal{L}_\Pi})} = \overline{t_k}^{\tau^{i+1}_\Pi(X|_{\mathcal{L}_\Pi})}$ (corollary 3.14 p.27)

- If $\tau_\Pi^i(X|_{\mathcal{L}_\Pi}) = \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})$

  Then for every k $\overline{t_k}^{\tau_\Pi^i(X|_{\mathcal{L}_\Pi})} = \overline{t_k}^{\tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})}$

  So $\overline{\overline{F_\Pi}^{P_\Pi^{i+1}(X)}} = true$

- If $\overline{\overline{F_\Pi}^{P_\Pi^{i+1}(X)}} = true$

  Then for every k $\overline{t_k}^{\tau_\Pi^i(X|_{\mathcal{L}_\Pi})} = \overline{t_k}^{\tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})}$

  So $\tau_\Pi^i(X|_{\mathcal{L}_\Pi})$ and $\tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})$ coincides over $Read(\Pi)$.

  By the remark p.12: $\Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi})) = \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi}))$

  So $\tau_\Pi^{i+2}(X|_{\mathcal{L}_\Pi})$

  $= \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi}) + \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi}))$

  $= (\tau_\Pi^i(X|_{\mathcal{L}_\Pi}) + \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi}))) + \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi}))$

  $= (\tau_\Pi^i(X|_{\mathcal{L}_\Pi}) + \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi}))) + \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi}))$

  $= \tau_\Pi^i(X|_{\mathcal{L}_\Pi}) + \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi}))$

  $= \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})$

  So $\tau_\Pi^{i+2}(X|_{\mathcal{L}_\Pi}) - \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi}) = \varnothing$

  But $\Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})) = \tau_\Pi^{i+2}(X|_{\mathcal{L}_\Pi}) - \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})$

  So $\Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi})) = \varnothing$

  But $\Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi})) = \Delta(\Pi, \tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi}))$

  So $\Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi})) = \varnothing$

  And $\tau_\Pi^{i+1}(X|_{\mathcal{L}_\Pi}) = \tau_\Pi^i(X|_{\mathcal{L}_\Pi}) + \Delta(\Pi, \tau_\Pi^i(X|_{\mathcal{L}_\Pi})) = \tau_\Pi^i(X|_{\mathcal{L}_\Pi})$

□