

# Semantics and Applications to Verification

Xavier Rival and Jérôme Feret

Notes by Antoine Groudiev

16th February 2024

## Contents

<b>1</b>	<b>Introduction to Semantics</b>	<b>1</b>
1.1	Case studies . . . . .	1
1.1.1	Ariane 5 – Flight 501 . . . . .	2
1.1.2	Lufthansa Flight 2904 . . . . .	2
1.1.3	Patriot missile (anti-missile system), Dahran . . . . .	2
1.1.4	General remarks . . . . .	3
1.2	Approaches to verification . . . . .	3
1.2.1	The termination problem . . . . .	3
1.2.2	A summary of common verification techniques . . . . .	5
1.3	Orderings, lattices and fixpoints . . . . .	5
1.3.1	Basic definitions on orderings . . . . .	5
1.3.2	Complete lattice . . . . .	5
1.3.3	How to prove semantic properties . . . . .	6
1.3.4	Operators over a poset . . . . .	6
1.3.5	Fixpoints theorems . . . . .	6
<b>2</b>	<b>Operational Semantics</b>	<b>7</b>
2.1	Definition and properties . . . . .	7
2.2	Examples . . . . .	8
2.2.1	Word recognition . . . . .	8
2.2.2	Pure $\lambda$ -calculus . . . . .	8
2.3	A MIPS like assembly language . . . . .	8
2.4	Traces semantics . . . . .	9

## Introduction

This document is Antoine Groudiev’s class notes while following the class *Sémantique et applications à la vérification de programmes* (Semantics and Applications to Verification) at the Computer Science Department of ENS Ulm. It is freely inspired by the class notes of Xavier Rival.

## 1 Introduction to Semantics

### 1.1 Case studies

We will study some examples of software errors: what are the causes of these, what kind of properties do we want to verify in order to prevent such failures?

### 1.1.1 Ariane 5 – Flight 501

Ariane 5 was a satellite launcher, aimed at replacing Ariane 4. Its first flight, June, 4th, 1996, was a failure, with more than \$370 000 000 of damages. 37 seconds after the launch, the rocket exploded.

The system contained sensors, two calculators (SRI, OBC), actuators, and redundant systems (failure tolerant system). The failure was due to an unhandled arithmetic error. Each register of the SRI has a size of 16, 32, or 64 bits. The error was due to a conversion of a 64-bit float to a 16-bit integer. The value was too large to be represented in 16 bits, and the conversion failed. The software was not able to handle this error, and the system crashed.

Several solutions would have prevented this mishapening:

- Desactivate interruptions on overflows
- Fix the SRI code, so that no overflow can happen. All conversions must be *guarded against overflows*:

```
double x = /* ... */ ;
short i = /* ... */ ;
if ( -32768. <= x && x <= 32767. )
    i = ( short ) x ;
else
    i = /* default value */ ;
```

This may be costly, but redundant tests can be removed.

- Handle conversion errors (not trivial): identify the problem and fix it at run time.

The piece of code that generated the error was used to do a useless task, the re-calibration process is not usefull after take-off. Furthermore, the code was already used in Ariane 4; initially protected by a safety guard, many conversions and tests were removed for the sake of performance after being tested on Ariane 4.

The crash was not prevented by redundant systems: the two calculators were running the same code, and the same error was made on both. Redundancy can prevent hardware errors, but is not enough to prevent software errors.

### 1.1.2 Lufthansa Flight 2904

On November 22, 2003, a Lufthansa Airbus A320-200 crashed at the airport of Warsaw, Poland. The plane was landing, and the weather was bad. The plane was not able to stop before the end of the runway, and crashed into a building. The cause of the crash was a software error in the plane's computer. The plane was not able to compute the correct deceleration, and the pilot was not able to stop the plane in time.

### 1.1.3 Patriot missile (anti-missile system), Dahrhan

The purpose of the Patriot system was to destroy foe missiles before they reach their target, and was used in the first Gulf War with a success rate around 50%. The system was used to destroy Scud missiles, and the system was not able to destroy one of them, which hit a barrack, killing 28 people. The cause of the failure was a software error in the system's clock. The system was not able to compute the time correctly due to fixed precision arithmetic error, and the missile was not destroyed.

### 1.1.4 General remarks

The examples given so far are not isolated cases. The typical causes of software errors are:

- Improper specification
- Incorrect implementation of a specification (the code should be free of runtime errors, and should produce a result that meets some property)
- Incorrect understanding of the execution model (generation of too imprecise results)

This creates new challenges to ensure embedded systems do not fail. The main techniques to ensure software safety are software development techniques:

- software engineering
- programming rules
- make software cleaner

In this class, we will instead dive into formal methods:

- should have sound mathematical foundations
- should allow guaranteeing software meet some complex properties
- should be trustable
- increasingly used in real life applications

This course will contain two main parts. The first part will be about Semantics, which allow describing precisely the behavior of programs, express the properties to verify, and to transform and compile programs. The second part, Verification, aims at proving semantic properties of programs. A very strong limitation of verification is undecidability; several approaches make various compromises around undecidability.

## 1.2 Approaches to verification

### 1.2.1 The termination problem

**Definition** (Termination). A program  $P$  terminates on input  $X$  if and only if any execution of  $P$  with input  $X$  eventually reaches a final state. A final state is a final point in the program (i.e., not an error).

**Definition** (Termination problem). Can we find a program  $P_t$  that takes as argument a program  $P$  and data  $X$  and that returns **True** if  $P$  terminates on  $X$ , and **False** otherwise?

**Property.** *The termination problem is not computable.*

*Proof.* We assume there exists a program  $P_a$  such that  $P_a$  always terminates, and returns 1 if and only if  $P$  terminates on input  $X$ . We consider the following program:

```
void P0 ( P ) {  
    if ( Pa ( P , P ) == 1 ) {  
        while ( 1 ) {  
            // loop forever  
        }  
    } else {  
        return ; // do nothing  
    }  
}
```

and we consider the return value of  $Pa(P0, P0)$ . If  $Pa(P0, P0) == 1$ , then  $P0$  loops forever, and if  $Pa(P0, P0) == 0$ , then  $P0$  terminates. This is a contradiction, and the termination problem is not computable.  $\square$

**Property.** *The absence of runtime errors is not computable. We cannot find a program  $Pc$  that takes a program  $P$  and input  $X$  as arguments, always terminates, and returns 1 if and only if  $P$  runs on input  $X$  without a runtime error.*

**Theorem** (Rice theorem). *Considering a Turing complete language, any non-trivial semantic specification is not computable. Therefore, all interesting properties are not computable (termination, absence of runtime/arithmetic errors, etc.).*

The initial verification problem is not computable. Several compromises can be made: simulation, testing, assisted theorem proving, model checking, bug-finding, static analysis with abstraction.

**Definition** (Safety verification problem). The Semantics  $\llbracket P \rrbracket$  of a program  $P$  is the set of behaviors of  $P$  (e.g. states). A property to verify  $\mathcal{S}$  is the set of admissible behaviors (e.g. safe states). Our goal is to establish  $\llbracket P \rrbracket \subseteq \mathcal{S}$

$\llbracket P \rrbracket$  can be sound (identify any wrong program), complete (accept all correct programs), and automated, but not all three at the same time.

**Testing by simulation** The principle of testing by simulation is to run the program on finitely many finite inputs, to maximize coverage and inspect erroneous traces to fix bugs. It is very widely used, through unit testing, integration testing, etc. It is both automated and complete, but is unsound and costly.

**Machine assisted proof** The principle of machine assisted proof is to have a machine check proof that is partly human written: tactics or solvers may help in the inference, and the hardest invariants have to be user-supplied. It is sound and quasi-complete, but not fully automated and costly.

**Model checking** We consider finite systems only, using algorithms for exhaustive exploration, symmetry reduction, ... It is automated, sound, and complete *with respect to the model*.

**Bug finding** The principle of bug finding is to identify "likely" issues, patterns known to often indicate an error: it uses bounded symbolic execution, model exploration, and rank "defect" reports using heuristics. It is neither sound nor complete, but is fully automated.

**Static analysis with abstraction** The principle of static analysis with abstraction is to use some approximation, but always in a conservative manner. We can use under-approximation of the property to verify:

$$\mathcal{S}_{\text{under}} \subseteq \mathcal{S}$$

and over-approximation of the semantics:

$$\llbracket P \rrbracket \subseteq \llbracket P \rrbracket_{\text{upper}}$$

We let an automatic static analyser attempt to prove that:

$$\llbracket P \rrbracket_{\text{upper}} \subseteq \mathcal{S}_{\text{under}}$$

If it succeeds, then we have proven that  $\llbracket P \rrbracket \subseteq \mathcal{S}$ . It is automated, sound, but incomplete.

	Automatic	Sound	Complete	Source level
Simulation	Yes	No	Yes	Yes
Assisted Proving	No	Yes	Almost	Partially
Model-checking	Yes	Yes	Partially	No
Bug-finding	Yes	No	No	Yes
Static analysis	Yes	Yes	No	Yes

### 1.2.2 A summary of common verification techniques

## 1.3 Orderings, lattices and fixpoints

### 1.3.1 Basic definitions on orderings

**Definition** (Partially ordered set (poset)). Let a set  $\mathcal{S}$  and a binary relation  $(\sqsubseteq) \subseteq \mathcal{S} \times \mathcal{S}$  over  $\mathcal{S}$ . Then,  $\sqsubseteq$  is an order relation if and only if it is reflexive, transitive, antisymmetric. Furthermore, we define  $x \sqsubset y ::= (x \sqsubseteq y \wedge x \neq y)$ . Most orders in this class won't be total.

We often use Hasse diagrams to represent posets.

In the following, we illustrate order relations and their usefulness in semantics using the standard definition of *word automata*. The semantics of an automaton is the set of words recognized by it.

We can already define a few semantic properties:

- $\mathcal{P}_0$ : no recognized word contains two consecutive  $b$

$$\mathcal{L}[\mathcal{A}] \subseteq L^* \setminus L^*bbL^*$$

- $\mathcal{P}_1$ : all recognized words contain at least one occurrence of  $a$

$$\mathcal{L}[\mathcal{A}] \subseteq L^*aL^*$$

- $\mathcal{P}_2$ : recognized words do not contain  $b$

$$\mathcal{L}[\mathcal{A}] \subseteq (L \setminus \{b\})^*$$

**Definition** ( $\perp, \top$ ). When they exist, we denote by infimum  $\perp$  and supremum  $\top$  the smallest and largest elements of the poset.

**Definition** ( $\sqcup, \sqcap$ ). We denote  $\sqcap \mathcal{S}$  the glb (greatest lower bound) of  $\mathcal{S}$ , and  $\sqcup$  the lub (lowest upper bound) of  $\mathcal{S}$ .

### 1.3.2 Complete lattice

**Definition** (Complete lattice). A complete lattice is a tuple  $(\mathcal{S}, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$  where  $(\mathcal{S}, \sqsubseteq)$  is a poset, and any subset  $\mathcal{S}'$  of  $\mathcal{S}$  has a glb  $\sqcap \mathcal{S}'$  and a lub  $\sqcup \mathcal{S}$ .

**Definition** (Lattice). A lattice is a tuple  $(\mathcal{S}, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$  where  $(\mathcal{S}, \sqsubseteq)$ , and any pair  $\{x, y\}$  of  $\mathcal{S}$  has a glb  $x \sqcap y$  and a lub  $x \sqcup y$ .

**Example.**  $\mathbb{Q} \cap [0, 1]$  is a lattice but not a complete lattice, since

$$\left\{ q \in \mathbb{Q} \cap [0, 1] \mid q \leq \frac{\sqrt{2}}{2} \right\} \subseteq \mathbb{Q} \cap [0, 1]$$

has no lowest upper bound in  $\mathbb{Q} \cap [0, 1]$ .

**Property.** *A finite lattice is also a complete lattice.*

**Definition** (Increasing chain). Let  $(\mathcal{S}, \sqsubseteq)$  be a poset and  $\mathcal{C} \subseteq \mathcal{S}$ . It is an increasing chain if and only if it is not empty, and  $(\mathcal{C}, \sqsubseteq)$  is total.

**Example.** *In the powerset  $(\mathcal{P}(\mathbb{N}), \subseteq)$ ,*

$$\mathcal{C} := \left\{ \{2^0, \dots, 2^i\} \mid i \in \mathbb{N} \right\}$$

*is an increasing chain.*

**Definition** (Increasing chain condition). The poset  $(\mathcal{S}, \sqsubseteq)$  satisfies the increasing chain condition if and only if any increasing chain  $\mathcal{C} \subseteq \mathcal{S}$  is finite.

**Definition** (Complete partial order). A complete partial order (cpo) is a poset  $(\mathcal{S}, \sqsubseteq)$  such that any increasing chain  $\mathcal{C}$  of  $\mathcal{S}$  has at least an upper bound. A pointed cpo is a cpo with a bottom element  $\perp$ .

### 1.3.3 How to prove semantic properties

#### 1.3.4 Operators over a poset

**Definition** (Operators and orderings). Let  $(\mathcal{S}, \sqsubseteq)$  be a poset and  $\varphi : \mathcal{S} \rightarrow \mathcal{S}$  be an operator over  $\mathcal{S}$ . Then,  $\varphi$  is:

- *monotone* if and only if  $x \sqsubseteq y \Rightarrow \varphi(x) \sqsubseteq \varphi(y)$
- *continuous* if and only if, for any **chain**  $\mathcal{S}' \subseteq \mathcal{S}$ , then:

$$\begin{cases} \text{if } \sqcup \mathcal{S}' \text{ exists, so does } \sqcup \varphi(\mathcal{S}') \\ \text{and } \sqcup \varphi(\mathcal{S}') = \varphi(\sqcup \mathcal{S}') \end{cases}$$

- $\sqcup$ -preserving if and only if:

$$\forall \mathcal{S}' \subseteq \mathcal{S}, \begin{cases} \text{if } \sqcup \mathcal{S}' \text{ exists, so does } \sqcup \varphi(\mathcal{S}') \\ \text{and } \sqcup \varphi(\mathcal{S}') = \varphi(\sqcup \mathcal{S}') \end{cases}$$

**Property** (Continuity  $\implies$  monotonicity). *If  $\varphi$  is continuous, then it is monotone.*

**Property** ( $\sqcup$ -preserving  $\implies$  monotonicity). *If  $\varphi$  preserves  $\sqcup$ , then it is monotone.*

#### 1.3.5 Fixpoints theorems

**Definition** (Fixpoints). Let  $(\mathcal{S}, \sqsubseteq)$  be a poset and  $\varphi : \mathcal{S} \rightarrow \mathcal{S}$  be an operator over  $\mathcal{S}$ .

- A fixpoint of  $\varphi$  is an element  $x$  such that  $\varphi(x) = x$ .
- A pre-fixpoint of  $\varphi$  is an element  $x$  such that  $x \sqsubseteq \varphi(x)$ .
- A post-fixpoint of  $\varphi$  is an element  $x$  such that  $\varphi(x) \sqsubseteq x$ .
- The least fixpoint of  $\varphi$  is a fixpoint  $x$  such that, for any fixpoint  $y$ ,  $x \sqsubseteq y$ .
- The greatest fixpoint of  $\varphi$  is a fixpoint  $x$  such that, for any fixpoint  $y$ ,  $y \sqsubseteq x$ .

**Theorem** (Tarski's). *Let  $(\mathcal{S}, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$  be a complete lattice and  $\varphi : \mathcal{S} \rightarrow \mathcal{S}$  be a monotone operator. Then:*

- $\varphi$  has a least fixpoint  $\text{lfp } \varphi$  and  $\text{lfp } \varphi = \bigcap \{x \in \mathcal{S} \mid \varphi(x) \sqsubseteq x\}$
- $\varphi$  has a greatest fixpoint  $\text{gfp } \varphi$  and  $\text{gfp } \varphi = \bigcup \{x \in \mathcal{S} \mid x \sqsubseteq \varphi(x)\}$
- the set of fixpoints of  $\varphi$  is a complete lattice

**Example.** We consider a set  $\mathcal{E}$ , and a subset  $\mathcal{A} \subseteq \mathcal{E}$ . We let:

$$\begin{aligned} f : \mathcal{P}(\mathcal{E}) &\rightarrow \mathcal{P}(\mathcal{E}) \\ X &\mapsto X \cup \mathcal{A} \end{aligned}$$

According to Tarski's theorem, the smallest fixpoint of  $f$  is  $\mathcal{A}$ , and the greatest is  $\mathcal{E}$ .

**Theorem** (Kleene's). Let  $(\mathcal{S}, \sqsubseteq, \perp)$  be a pointed cpo and  $\varphi : \mathcal{S} \rightarrow \mathcal{S}$  be a continuous operator over  $\mathcal{S}$ . The  $\varphi$  has a least fixpoint, and

$$\text{lfp } \varphi = \bigsqcup_{n \in \mathbb{N}} \varphi^n(\perp)$$

## 2 Operational Semantics

Operational semantics are mathematical descriptions of the executions of a program. It is based on a model of programs, called transition systems.

### 2.1 Definition and properties

**Definition** (Transition systems (TS)). A transition system is a tuple  $(\mathbb{S}, \rightarrow)$  where  $\mathbb{S}$  is the *set of states of the system*, and  $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$  is the transition relation of the system.

Note that the set of states may be infinite. The majority of interesting examples come from the cases where  $\mathbb{S}$  is infinite.

**Definition** (Deterministic system). A deterministic system is such that a state fully determines the next state.

$$\forall s_0, s_1, s'_1 \in \mathbb{S}, (s_0 \rightarrow s_1 \wedge s_0 \rightarrow s'_1) \implies s_1 = s'_1$$

Otherwise, a transition system is non-deterministic.

The transition relation  $\rightarrow$  defines atomic execution steps. It is often called *small-step semantics* or *structured operational semantics*. Steps are *discrete*, and we do not consider non-deterministic systems with probability on transitions (probabilistic transition systems).

**Definition** (Initial and final states). We often consider transition systems with a set of initial and final states:

1. a set of initial states  $\mathbb{S}_I \subseteq \mathbb{S}$  denotes states where the execution should start
2. a set of final states  $\mathbb{S}_F \subseteq \mathbb{S}$  denotes states where the execution should reach the end of the program

When needed, we add these to the definition of the transition systems  $(\mathbb{S}, \rightarrow, \mathbb{S}_I, \mathbb{S}_F)$ .

**Definition** (Blocking state  $\neq$  final state). A state  $s_0 \in \mathbb{S}$  is blocking when it is the origin of no transition:

$$\forall s_1 \in \mathbb{S}, \neg(s_0 \rightarrow s_1)$$

As an example, we often introduce an error state (usually noted  $\Omega$ )

## 2.2 Examples

### 2.2.1 Word recognition

We can formalize the *word recognition* by a finite automaton using a transition system. We consider an automaton  $\mathcal{A} = (Q, q_i, q_f, \rightarrow)$ . A state is defined by the remaining of the word to recognize, and the automaton state that has been reached so far. Thus,

$$\mathbb{S} = Q \times \Sigma^*$$

We define the transition relation  $\rightarrow$  to be:

$$(q_O, aw) \rightarrow (q_1, w) \iff q_O \rightarrow^a q_1$$

The initial and final states are defined by:

$$\begin{cases} \mathbb{S}_I = \{(q_i, w) \mid w \in \Sigma^*\} \\ \mathbb{S}_F = \{(q_f, \varepsilon)\} \end{cases}$$

### 2.2.2 Pure $\lambda$ -calculus

A bare-bones model of functional programming:

**Definition** ( $\lambda$ -terms). The set of  $\lambda$ -terms is defined by:

$$\begin{array}{ll} t, u, \dots ::= x & \text{variable} \\ \mid \lambda x \cdot t & \text{abstraction} \\ \mid tu & \text{application} \end{array}$$

**Definition** ( $\beta$ -reduction).

The  $\lambda$ -calculus defines a transition system.  $\mathbb{S}$  is the set of  $\lambda$ -terms and  $\rightarrow_\beta$  the transition relation.  $\rightarrow_\beta$  is non-deterministic, since multiple  $\beta$ -reduction are sometimes possible. Given a lambda term  $t_0$ , we may consider  $(\mathbb{S}, \rightarrow_\beta, \mathbb{S}_I)$  where  $\mathbb{S}_I = \{t_0\}$ . Blocking states are terms with no redex  $(\lambda x \cdot u)v$ .

## 2.3 A MIPS like assembly language

We now consider a very simplified assembly language, containing machine integers  $\mathbb{B}^{32}$ . Instruction are encoded over 32 bits and stored in the same space as data,  $\mathbb{B}^{32}$ . We assume a fixed set of addresses  $A$ .

The memory configuration contains the program counter **pc**, the general purpose register  $r_0, \dots, r_{31}$ , and the main memory (RAM):

$$\text{mem} : A \subseteq \mathbb{B}^{32} \rightarrow \mathbb{B}^{32}$$

**Definition** (State). A state is a tuple  $(\pi, \rho, \mu)$  which comprises a program counter value  $\pi \in \mathbb{B}^{32}$ , a function mapping each general purpose register to its value  $\rho : \{0, \dots, 32\} \rightarrow \mathbb{B}^{32}$ , and a function mapping each memory cell to its value,  $\mu : A \rightarrow \mathbb{B}^{32}$ .

We can define transition relations.

We now look at a more classical imperative language (a bare-bone subset of C), containing variables  $X$ , labels  $L$ , and values  $V$ . A syntax can be defined.



## 2.4 Traces semantics

**Definition** (Traces). A *finite trace* is a finite sequence of states  $s_0, \dots, s_n$  noted  $\langle s_0, \dots, s_n \rangle$ . An infinite trace is an infinite sequence of states  $\langle s_0, \dots \rangle$ . Besides, we write  $\mathbb{S}^*$  for the set of finite traces,  $\mathbb{S}^\omega$  for the set of infinite traces, and  $\mathbb{S}^\alpha = \mathbb{S}^* \cup \mathbb{S}^\omega$ .

**Definition** (Concatenation operator  $\cdot$ ).