Semantics and Applications to Verification

Xavier Rival, Sylvain Conchon, Jérôme Feret, Antoine Miné

Class notes by Antoine Groudiev

Last modified 15th April 2024

Contents

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 and Jérôme Feret.

1 Introduction to Semantics

In this first chapter, we will introduce verification: we will see real-life uses of verification, different approaches to it, and will start building mathematical tools to define semantics and walk our first steps towards program verification.

The first chapter is purely an explanation of the practical need for verification, and can be skipped without loss of comprenhension.

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 mishappening:

- 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 converson 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), Dahran

The purporse 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 sounds 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 indecidability; several approaches make various compromises around indecidability.

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 Pt that takes as argument a program P and data X and that returns `True` if P terminates on X, and `False` otherwise?

Property 1.1. The termination problem is not computable.

Proof. We assum there exists a program Pa such that Pa 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. □

Property 1.2. 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 $[\![P]\!]$ 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 $[\![P]\!] \subseteq \mathcal{S}$

$[\![P]\!]$ 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:

$$[\![P]\!] \subseteq [\![P]\!]_{\text{upper}}$$

We let an automatic static analyser attempt to prove that:

$$[\![P]\!]_{\text{upper}} \subseteq \mathcal{S}_{\text{under}}$$

If it succedds, then we have proven that $[\![P]\!] \subseteq \mathcal{S}$. It is automated, sound, but incomplete.

1.2.2 A summary of common verification techniques

	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

1.3 Orderings, lattices and fixpoints

1.3.1 Basic definitions on orderings

Orderings are very useful in semantics and verification. Logical ordering expresses implication of logical facts, and computational ordering is useful to establish well-foundness of fixpoint definitions, and for proving termination.

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, antisymettric. 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.

Example (Hasse diagram)**.** The following order (given by the extensive definition) can simply be represented in a diagram:

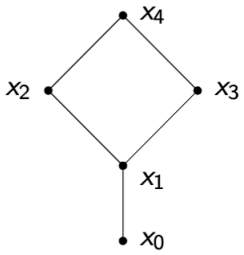
Extensive definition:

- $\mathcal{S} = \{x_0, x_1, x_2, x_3, x_4\}$
- $\sqsubset$ defined by:

$$x_0 \sqsubset x_1$$
$$x_1 \sqsubset x_2$$
$$x_1 \sqsubset x_3$$
$$x_2 \sqsubset x_4$$
$$x_3 \sqsubset x_4$$

- By reflexivity, we have, e.g., $x_1 \sqsubseteq x_1$
- By transitivity, we have, e.g., $x_1 \sqsubseteq x_4$

Diagram:



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

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

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

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

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

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

Definition ($\bot$, $\top$)**.** When they exist, we denote by infimum $\bot$ 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 (Complet lattice)**.** A complete lattice is a tuple $(\mathcal{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $(S, \sqsubseteq)$ 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 $(\mathcal{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $(S, \sqsubseteq)$, 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

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

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

Property 1.3. 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 $(\mathcal{P}(\mathbb{N}), \subseteq)$,

$$\mathcal{C} := \left\{ \{2^0, \ldots, 2^i\} \,\middle|\, i \in \mathbb{N} \right\}$$

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 $\bot$.

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 $\varphi : \mathcal{S} \to \mathcal{S}$ be an operator over $\mathcal{S}$. Then, φ 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 1.4 (Continuity $\implies$ monotonicity)**.** If φ is continuous, then it is monotone.

Property 1.5 ($\sqcup$-preserving $\implies$ monotonicity)**.** If φ preserves $\sqcup$, then it is monotone.

1.3.5 Fixpoints theorems

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

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

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

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

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

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

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

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

$$\operatorname{lfp} \varphi = \bigsqcup_{n \in \mathbb{N}} \varphi^n(\bot)$$

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 $(\mathbb{S}, \to)$ where $\mathbb{S}$ is the *set of states of the system*, and $\to \subseteq \mathbb{S} \times \mathbb{S}$ 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.

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

Otherwise, a transition system is non-deterministic.

The transition relation $\to$ 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 $\mathbb{S}_I \subseteq \mathbb{S}$ denots 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 $(S, \to, \mathbb{S}_I, \mathbb{S}_F)$.

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

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

As an example, we often introduce an error state (usually noted Ω)

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 $\mathcal{A} = (Q, q_i, q_f, \to)$. A state is defined by the remaining of the word to

recognize, and the automaton state that has been reached so far. Thus,

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

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

$$(q_O, aw) \rightarrow (q_1, w) \iff q_0 \rightarrow^a q_1$$

The initial and final states are defined by:

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

2.2.2 Pure λ-calculus

A bare-bones model of functional programming:

Definition (λ-terms)**.** The set of λ-terms is defined by:

$$\begin{array}{rr} t, u, \cdots ::= x & \textit{variable} \\ \mid \lambda x \cdot t & \textit{abstraction} \\ \mid tu & \textit{application} \end{array}$$

Definition (β-reduction)**.**

The λ-calculus defines a transition system. $\mathbb{S}$ is the set of λ-terms and $\rightarrow_\beta$ the transition relation. $\rightarrow_\beta$ is non-deterministic, since multiple β-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 $r_0, \ldots, r_{31}$, and the main memory (RAM):

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

Definition (State)**.** A state is a tuple (π, ρ, μ) which comprises a program counter value $\pi \in \mathbb{B}^{32}$, a function mapping each general purpose register to its value $\rho : \{0, \ldots, 32\} \rightarrow \mathbb{B}^{32}$, 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 $s_0, \ldots, s_n$ noted $\langle s_0, \ldots, s_n \rangle$. An infinite trace is an infinite sequence of states $\langle s_0, \ldots \rangle$. Besides, we write $\mathbb{S}^\star$ for the set of finite traces, $\mathbb{S}^\omega$ for the set of infinite traces, and $S^\infty = \mathbb{S}^\star \cup \mathbb{S}^\omega$.

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

3 Traces Properties

3.1 A high level overview

3.1.1 Introduction

The goal of verification is to prove that $[\![P]\!] \subseteq \mathcal{S}$ (i.e. that all behaviors of P satisfy specifications $\mathcal{S}$) where $[\![P]\!]$ is the program semantics and $\mathcal{S}$ the desired specification. Today, we will mostly focus on program's properties, $\mathcal{S}$. We will see different families of properties, proof techniques, and specification of properties (are there languages to describe properties?).

A property is a set of traces, defining the admissible executions. There are *safety properties*: something will never happen, which is often proven by invariance; *liveness properties*: something will eventually happen, proven by variance; and beyond safety and liveness, there are *hyperproperties*.

As usual, we consider $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_{\mathcal{I}})$.

Definition (Properties as sets of states)**.** A property $\mathcal{P} \subseteq \mathbb{S}$ is a set of states $\mathcal{P} \subseteq \mathbb{S}$. $\mathcal{P}$ is satisfied if and only if all reacheable states belong to $\mathcal{P}$.

This is the case of the absence of runtime errors, and non-termination.

Definition (Properties as sets of traces)**.** A property $\mathcal{T}$ is a set of traces $\mathcal{T} \subseteq \mathbb{S}^{\infty}$. $\mathcal{T}$ is satisfied if and only if all traces belong to $\mathcal{T}$, i.e. $[\![\mathbb{S}^{\infty}]\!] \subseteq \mathcal{T}$

State properties are trace properties. Functional properties (such as "program P takes one integer input x and returns its absolute value") and termination ($\mathcal{T} = \mathbb{S}^{\star}$) are trace properties.

3.1.2 A few monotonicity properties

Property 3.1. Let $\mathcal{P}_0, \mathcal{P}_1 \subseteq \mathbb{S}$ be two state properties. When $\mathcal{P}_0 \subseteq \mathcal{P}_1$, we say that $\mathcal{P}_0$ is *stronger* than $\mathcal{P}_1$, i.e. if a program $\mathcal{S}$ satisfies $\mathcal{P}_0$ then it also satisfies $\mathcal{P}_1$.

Property 3.2. Let $\mathcal{T}_0, \mathcal{T}_1 \subseteq \mathbb{S}$ be two trace properties. When $\mathcal{T}_0 \subseteq \mathcal{T}_1$, we say that $\mathcal{T}_0$ is *stronger* than $\mathcal{T}_1$, i.e. if a program $\mathcal{S}$ satisfies $\mathcal{T}_0$ then it also satisfies $\mathcal{T}_1$.

Property 3.3. Let $\mathcal{S}_0, \mathcal{S}_1$ be two transition systems. When $[\![\mathcal{S}_0]\!] \subseteq [\![\mathcal{P}_1]\!]$, we say that $\mathcal{S}_1$ *has more behaviors than* $\mathcal{S}_0$, i.e. if a property $\mathcal{P}$ is satisfied by $\mathcal{S}_1$ then it is also satisfied by $\mathcal{S}_0$.

3.2 Safety properties

3.2.1 Informal and formal definitions

Definition (Informal definition of safety properties)**.** A safety property is a property which specifies that some (bad) behavior defined by a finite, irrecoverable observation will never occur, at any time.

Example. The following properties are safety properties:

- Absence of runtime errors
- State properties (the "bad thing" is reaching $\mathbb{S} \setminus \mathcal{T}$)
- Non-termination
- "Not reaching state b after visiting state a"

Termination is **not** a safety property, since no finite execution is a counter-example of its termination.

We now intend to provide a formal definition of safety. How to refute a safety property? We assume $\mathcal{S}$ does not satisfy safety property $\mathcal{P}$. Thus, there exists a counter-example trace $\sigma = \langle s_0, \ldots, s_n, \ldots \rangle \in [\![S]\!] \setminus \mathcal{P}$. At this point of our study, the trace may be finite or infinite. The intuitive definition says this trace *eventually exhibits some bad behavior*, that is *irrecoverable* at some *observed at some given time*, thus the observation corresponds to some index i. Therefore, trace $\sigma' = \langle s_0, \ldots, s_i \rangle$ violates $\mathcal{P}$, i.e. $\sigma' \notin \mathcal{P}$. Due to the irrecoverability of the observation, the same goes for any trace with the same prefix. We remark that σ' is finite.

A safety property that does not hold can always be refuted with a finite, irrecoverable counter-example.

3.2.2 A few operators on traces

Definition (Prefix)**.** We write $\sigma_{\lceil i}$ for the prefix of length i of trace σ.

Definition (Suffix or tail)**.** We write $\sigma_{i\rceil}$ for the suffix of length i of trace σ.

Definition (Upper closure operators (PCl))**.** In a preorder $(\mathcal{S}, \sqsubseteq)$, a function $\varphi : \mathcal{S} \to \mathcal{S}$ is an upper closure operatof if and only if it is monotone, extensive ($\forall x \in \mathcal{S}, x \sqsubseteq \varphi(x)$) and idempotent.

Definition (Prefix closure)**.** The prefix closure operator is defined by:

$$\begin{aligned} \mathrm{PCl} : \mathcal{P}(\mathbb{S}^\infty) &\to \mathcal{P}(\mathbb{S}^\star) \\ X &\mapsto \left\{ \sigma_{\lceil i} \;\middle|\; \sigma \in X, i \in \mathbb{N} \right\} \end{aligned}$$

PCl is monotone, idempotent, but not extensive on $\mathcal{P}(\mathbb{S}^\infty)$ (infinite traces do not appear anymore). Its restriction to $\mathcal{P}(\mathbb{S}^\star)$ is extensive.

Definition (The Lim operator)**.** The limit operator is defined by:

$$\begin{aligned} \mathrm{Lim} : \mathcal{P}(\mathbb{S}^\infty) &\to \mathcal{P}(\mathbb{S}^\infty) \\ X &\mapsto X \cup \left\{ \sigma \in \mathbb{S}^\infty \;\middle|\; \forall i \in \mathbb{N}, \sigma_{\lceil i} \in X \right\} \end{aligned}$$

Note that the operator Lim is an upper-closure operator.

Proof. Left as an exercise! □

Example. Assume that:

$$\mathcal{S} = \{\varepsilon, \langle a \rangle, \langle a, b \rangle, \langle a, b, a \rangle, \langle a, b, a, b \rangle, \langle a, b, a, b, a \rangle, \ldots\}$$

then,

$$\mathrm{Lim}(\mathcal{S}) = \mathcal{S} \uplus \{\langle a, b, a, b, a, b, \ldots \rangle\}$$

3.2.3 Formal definition of safety

Definition (The Safe operator)**.** Operator Safe is defined by

$$\mathrm{Safe} = \mathrm{Lim} \circ \mathrm{PCl}$$

Note that Safe is an upper closure operator over $\mathcal{P}(\mathbb{S}^\infty)$.

Definition (Safety property)**.** A trace property $\mathcal{T}$ is a safety property if and only if it is a fixpoint of the Safe operator, that is

$$\mathrm{Safe}(\mathcal{T}) = \mathcal{T}$$

Furthermore, if $\mathcal{T}$ is a trace property, then $\mathrm{Safe}(\mathcal{T})$ is a safety property, since Safe is idempotent.

Theorem. Any state property is also a safety property.

Proof. Consider a state property $\mathcal{P}$. It is equivalent to trace property $\mathcal{T} = \mathbb{P}^{\propto}$:

$$\begin{aligned}\mathrm{Safe}(\mathcal{T}) &= \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{P}^{\propto}) \\ &= \mathrm{Lim}(\mathcal{P}^{\star}) \\ &= \mathcal{P}^{\star} \cup \mathcal{P}^{\omega} \\ &= \mathcal{P}^{\propto} \\ &= \mathcal{T}\end{aligned}$$

Therefore, $\mathcal{T}$ is indeed a safety property. □

3.2.4 Intuition of the formal definition

Operator Safe saturates a set of traces S with prefixes and infinite traces of all finite prefixes of which can be observed in S. Thus, if $\mathrm{Safe}(S) = S$ and σ is a trace, to establish that $\sigma \notin S$, it is sufficient to discover a *finite prefix* of σ that cannot be observed in S.

Alternatively, if all finite prefixes of σ belong to S or can be observed as a prefix of another trace in S, by definition of the limit operator, σ belongs to S, *even if it is infinite.* Therefore, the definition captures properties that *can be disproved with a finite counter-example.*

3.2.5 Proof method for safety properties

We consider transition system $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_{\mathcal{I}})$ and a safety property $\mathcal{T}$. Finite traces semantics is the least fixpoint of F_*. We seek a way of verifying that $\mathcal{S}$ satisfies $\mathcal{T}$, i.e. that $[\![\mathcal{S}]\!]^{\propto} \subseteq \mathcal{T}$.

Definition (Invariance proofs)**.** Let $\mathbb{I}$ be a set of finite traces; it is said to be an *invariant* if and only if:

$$\begin{cases}\forall s \in \mathbb{S}_{\mathcal{I}}, \langle s \rangle \in \mathbb{I} \\ F_*(\mathbb{I}) \subseteq \mathbb{I}\end{cases}$$

Where F_* is the semantic function, defined previously, that computes the traces of length $i+1$ from the traces of length i, and adds the traces of length 1.

$\mathbb{I}$ is *stronger* than $\mathcal{T}$ if and only if $\mathbb{I} \subseteq \mathcal{T}$. The proof methode *by invariance* is based on finding an invariant that is stronger than $\mathcal{T}$.

Theorem (Soundness of the invariance proof method)**.** The invariance proof methode is *sound*: if we can find an invariant for $\mathcal{S}$, that is stronger than safety property $\mathcal{T}$, then $\mathcal{S}$ satisfies $\mathcal{T}$.

Proof. Assume that $\mathbb{I}$ is an invariant of $\mathcal{S}$ and that it is stronger than $\mathcal{T}$. Let's show that $\mathcal{S}$ satisfies $\mathcal{T}$.

By induction over n, we can prove that $F_*^n(\{\, \langle s \rangle \mid s \in \mathbb{S}_{\mathcal{I}} \,\}) \subseteq F_*^n(\mathbb{I}) \subseteq \mathbb{I}$. Therefore, $[\![\mathcal{S}]\!]^{\star} \subseteq \mathbb{I}$ and thus, $\mathrm{Safe}([\![\mathcal{S}]\!]^{\star}) \subseteq \mathrm{Safe}(\mathbb{I}) \subseteq \mathrm{Safe}(\mathcal{T})$ since Safe is monotone.

We remark that $[\![\mathcal{S}]\!]^{\propto} = \mathrm{Safe}([\![\mathcal{S}]\!]^{\star})$, hence $[\![\mathcal{S}]\!]^{\propto} = \mathrm{Safe}([\![\mathcal{S}]\!]^{\star}) \subseteq \mathrm{Safe}\,\mathcal{T} = \mathcal{T}$. We conclude $[\![\mathcal{S}]\!]^{\propto} \subseteq \mathcal{T}$, i.e. $\mathcal{S}$ statisfies property $\mathcal{T}$. □

Theorem (Completeness of the invariance proof method)**.** The invariance proof method is *complete*: if $\mathcal{S}$ satisfies safety property $\mathcal{T}$, then we can find an invariant $\mathbb{I}$ for $\mathcal{S}$, that is stronger than $\mathcal{T}$.

Proof. We choose $\mathbb{I} = [\![\mathcal{S}]\!]^\star$, which is both an invariant of $\mathcal{S}$ and is stronger than $\mathcal{T}$ since $\mathcal{S}$ satisfies $\mathcal{T}$. □

Note that $[\![\mathbb{S}]\!]^\propto$ is most likely not a very easy to express invariant, but it is just a convenient completeness argument. Therefore, completeness does not mean that the proof is easy.

Example. Consider the following program which computed the sum of the elements of an array: We want to proove that when the exit is reached, $\mathtt{s} = \sum_{k=0}^{n-1} \mathtt{t}[k]$. For each program point l, we

```
s = 0;
i = 0;
while (i < n) {
    s = s + t[i];
    i = i + 1;
}
```

Figure 1: A simple program which computes the sum of the elements of `t`

have a local invariant $\mathbb{I}_l$, denoted below by a logical formula instead of a set of states:

```
1 s = 0;
2 i = 0;
3 while (i < n) {
4     s = s + t[i];
5     i = i + 1;
6 }
```

$$
\begin{aligned}
l_0 &: (\!|\mathtt{true}|\!) \\
l_1 &: (\!|s = 0|\!) \\
l_2 &: (\!|i = 1 \wedge s = 0|\!) \\
l_3 &: (\!|0 \leqslant i < n \wedge s = \textstyle\sum_{k=0}^{i-1} t[k]|\!) \\
l_4 &: (\!|0 \leqslant i < n \wedge s = \textstyle\sum_{k=0}^{i} t[k]|\!) \\
l_5 &: (\!|0 \leqslant i \leqslant n \wedge s = \textstyle\sum_{k=0}^{i-1} t[k]|\!) \\
l_6 &: (\!|i = n \wedge s = \textstyle\sum_{k=0}^{n-1} t[k]|\!)
\end{aligned}
$$

Figure 2: The program and the local invariants

The global invariant $\mathbb{I}$ is defined by:

$$\mathbb{I} := \{\, \langle (l_0, m_0), \ldots, (l_n, m_n) \rangle \mid \forall n, m_n \in \mathbb{I}_{l_n} \,\}$$

which is the set of traces satisfying all the local invariants.

3.3 Liveness properties

Similarly, we will start by sketching an informal definition of a *liveness property*, before introducing an operator-based formal definition and a proof mechanism.

3.3.1 Informal definition

Definition (Informal definition of liveness properties)**.** A liveness property is a property which specifies that some (good) behavior *will eventually occur*, and that this behavior *may still occur after any finite observation.*

- *Termination* is a liveness property, in which the "good behavior" is reaching a blocking state (no more transition are available).
- "State a will enventually be reached by any execution" is also a liveness property, in which the "good behavior" is reaching the state a.
- The absence of runtime errors is **not** a liveness property.

We can also ask ourselves how to refute a lineveness property. Considering a liveness property $\mathcal{T}$ (i.e. $\mathcal{T}$ is termination), we assume that $\mathcal{S}$ does **not** satisfy liveness property $\mathcal{T}$. Thus, there exists a counter-example trace $\sigma \in [\![\mathcal{S}]\!] \setminus \mathcal{T}$. The informal definition says "... may still occur after any finite observation", thus each finite trace σ' can be extended into a good trace. Therefore, σ is necessarily *infinite*.

To prove that a liveness property does not hold, we need to look for an infinite counter-example: no finite trace is a counter-example.

For example, when $\mathcal{T}$ is the termination property, no finite execution can guarantee us that the program will not eventually stop; to prove that a program will not halt, we need to exhibit an infinite execution.

3.3.2 Formal definition

Definition (Operator Live)**.** The operator Live is defined by:

$$\mathrm{Live}(\mathcal{T}) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T}))$$

Definition (Liveness property)**.** $\mathcal{T}$ is a liveness property if and only if it satisfies one of the three following equivalend statements:

(i) $\mathrm{Live}(\mathcal{T}) = \mathcal{T}$
(ii) $\mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\star$ (any finite trace is the prefix of a trace in $\mathcal{T}$)
(iii) $\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\infty$

Proof: $(i) \implies (ii)$. We assume that $\mathrm{Live}(\mathcal{T}) = \mathcal{T}$, i.e. that $\mathcal{T} \cup (\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T})) = \mathcal{T}$. Therefore, $\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T}) \subseteq \mathcal{T}$.

Let $\sigma \in \mathbb{S}^\star$ and let us show that $\sigma \in \mathrm{PCl}(\mathcal{T})$. Since we must have $\sigma \in \mathbb{S}^\infty$:

- either $\sigma \in \mathrm{Safe}(\mathcal{T}) = \mathrm{Lim}(\mathrm{PCl}(\mathcal{T}))$, so all its prefixes are in $\mathrm{PCl}(\mathcal{T})$ and $\sigma \in \mathrm{PCl}(\mathcal{T})$;
- or $\sigma \in \mathcal{T}$ which implies that $\sigma \in \mathrm{PCl}(\mathcal{T})$.

□

Proof: $(ii) \implies (iii)$. If $\mathrm{PCl}\,\mathcal{T} = \mathbb{S}^\star$, then $\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\infty$. □

Proof: $(iii) \implies (i)$. If $\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\infty$, then

$$\mathrm{Live}(\mathcal{T}) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus (\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}))) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus \mathbb{S}^\infty) = \mathcal{T}$$

□

Example (Termination)**.** The property $\mathcal{T} = \mathbb{S}^\star$ corresponds to termination: there should be no infinite execution. $\mathcal{T}$ clearly satisfies

$$\mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\star$$

thus termination is indeed a liveness property according to the formal definition.

Example. Assume that $\mathbb{S} = \{a, b, c\}$ and that $\mathcal{T}$ is the set of states in which b has been visited after a has been visited. $\mathcal{T}$ can be described by $\mathcal{T} = \mathrm{PCl}(\mathbb{S}^\star \cdot a \cdot \mathbb{S}^\star \cdot b \cdot \mathbb{S}^\star)$.

Then, $\mathcal{T}$ is a liveness property. Let $\sigma \in \mathbb{S}^\star$; then $\sigma \cdot a \cdot b \in \mathcal{T}$, so $\sigma \in \mathrm{PCl}\, T$. Thus, $\mathrm{PCl}\, \mathcal{T} = \mathbb{S}^\star$, which corresponds to property (ii).

Theorem (Live is idempotent)**.** If $\mathcal{T}$ is a trace property, then $\mathrm{Live}(\mathcal{T})$ is a liveness property.

Proof. Let's show that $\mathrm{PCl} \circ \mathrm{Live}(\mathcal{T}) = \mathbb{S}^\star$. Consider $\sigma \in \mathbb{S}^\star$, we will prove that $\sigma \in \mathrm{PCl} \circ \mathrm{Live}(\mathcal{T})$. Note that:

$$\begin{aligned}\mathrm{PCl} \circ \mathrm{Live}(\mathcal{T}) &= \mathrm{PCl}(\mathcal{T}) \cup \mathrm{PCl}(\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T})) \\ &= \mathrm{PCl}(\mathcal{T}) \cup \mathrm{PCl}(\mathbb{S}^\infty \setminus \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}))\end{aligned}$$

If $\sigma \in \mathrm{PCl}(\mathcal{T})$, the result is obvious. If $\sigma \notin \mathrm{PCl}(\mathcal{T})$, then $\sigma \notin \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T})$ by definition of the limit, thus $\sigma \in \mathbb{S}^\infty \setminus \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T})$. Therefore, $\sigma \in \mathrm{PCl}(\mathbb{S}^\infty \setminus \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}))$ as PCl is extensive when applied to sets of finite traces, which proves the above result. □

3.3.3 Termination proof with ranking function

In this section, we will only consider termination. Given a transition system $\mathcal{S} = (\mathbb{S}, \to, \mathbb{S}_\mathbb{I})$, and a liveness property $\mathcal{T}$, we look for a way of verifying that $\mathcal{S}$ satisfies termination, i.e. that $[\![\mathcal{S}]\!]^\infty \subseteq \mathbb{S}^\star$. To do so, we introduce the notion of ranking function.

Definition (Ranking function)**.** A *ranking function* is a function $\phi : \mathbb{S} \to E$ where $(E, \sqsubseteq)$ is a *well-founded* ordering[1], and

$$\forall s_1, s_2 \in \mathbb{S}, s_0 \to s_1 \implies \phi(s_1) \sqsubset \phi(s_0)$$

Intuitively, following a transition can only make $\phi(s)$ striclty decrease, and $\phi(s)$ cannot keep decreasing forever, since the order is well-founded.

Theorem. If $\mathcal{S}$ has a ranking function ϕ, it satisfies termination.

Example. We consider the termination of the array sum program. As seen on Figure **??**, ϕ is a ranking function, hence the transition system of our program satistifes termination.

```
1  s = 0;
2  i = 0;
3  while (i < n) {
4      s = s + t[i];
5      i = i + 1;
6  }
```

$$\begin{aligned}\phi :\ \mathbb{S} &\longrightarrow \mathbb{N} \\ (l_1, m) &\longmapsto 3n + 6 \\ (l_2, m) &\longmapsto 3n + 5 \\ (l_3, m) &\longmapsto 3n + 4 \\ (l_4, m) &\longmapsto 3(n - m(i)) + 3 \\ (l_5, m) &\longmapsto 3(n - m(i)) + 2 \\ (l_6, m) &\longmapsto 3(n - m(i)) + 4 \\ (l_7, m) &\longmapsto 0\end{aligned}$$

Figure 3: The program and the ranking function

[1] An ordering is well-founded when every non-empty subset of E has a minimal element for $\sqsubseteq$.

3.3.4 Proof by variance

We consider a transition system $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_\mathcal{I})$, and liveness property $\mathcal{T}$. Infinite traces semantics is the greatest fixpoint of F_ω. We seek a way of verifying that $\mathcal{S}$ satisfies $\mathcal{T}$, i.e. that $[\![\mathcal{S}]\!]^\infty \subseteq \mathcal{T}$.

Definition (Variance proofs)**.** Let $(\mathbb{I}_n)_{n\in\mathbb{N}}$ be elements of $\mathbb{S}^\infty$; these are said to form a *variance proof* if and only if:

- $\mathbb{S}^\omega \subseteq \mathbb{I}_0$
- for all $k \in 1, 2, \ldots, \omega$, $\forall s \in \mathbb{S}, \langle s \rangle \in \mathbb{I}_k$
- for all $k \in 1, 2, \ldots, \omega$, there exists $l < k$ such that $F_\omega(\mathbb{I}_l) \subseteq \mathbb{I}_k$
- $\mathbb{I}_\omega \subseteq \mathcal{T}$

Theorem. The variance proof method is both sound and complete.

3.4 Decomposition of trace properties

Theorem. Let $\mathcal{T} \subseteq \mathbb{S}^\infty$; it can be decomposed into the conjunction of safety property $\text{Safe}(\mathcal{T})$ and liveness property $\text{Live}(\mathcal{T})$:

$$\mathcal{T} = \text{Safe}(\mathcal{T}) \cap \text{Live}(\mathcal{T})$$

Example. Considering the same program: we try to prove its total correctness. We need to

```
s = 0;
i = 0;
while (i < n) {
    s = s + t[i];
    i = i + 1;
}
```

Figure 4: A simple program which computes the sum of the elements of `t`

show that the program terminates, does not crash, and computes the sum of the elements in the array. We can apply the decomposition principle and express total correctness as the *conjunction of two proofs*: termination can be proved with a ranking function, and the absence of errors and result correctness can be proved with local invariants.

Example. Likewise, we consider a very simple greatest common divider code function:

```
int f(int a, int b) {
    while (a > 0) {
        int d = b/a;
        int r = r - a * d;
        b = a;
        a = r;
    }
    return b;
}
```

Figure 5: A simple program which computes the sum of the elements of `t`

The specification of this function – that we want to prove – is: "When applied to positive integers, function f should always return their GCD". The safety part is the conjunction of

two properties: no runtime errors, and the final value of b is the GCD. The liveness part is termination on all traces starting with positive inputs.

3.5 A specification language: temporal logic

3.5.1 Notion of specification language

Ultimately, we would like to *verify* or *compute* properties, but so far, we simply describe properties with sets of executions or worse, natural language statements. Ideally, we would prefer to use a mathematical language for that, to gain in concision and avoid ambiguity.

Definition (Specification language)**.** A specification language is a set of terms $\mathbb{L}$ with an interpretation function (or semantics):

$$[\![\cdot]\!] : \mathbb{L} \to \mathcal{P}(\mathbb{S}^\infty)$$

3.5.2 A state specification language

We will consider the following example of a simple specification language.

Definition. Syntax: we let terms of $\mathcal{L}_\mathbb{S}$ be defined by:

$$p \in \mathcal{L}_\mathbb{S} ::= @l | x < x' | x < n | \neg p' | p' \wedge p'' | \Omega$$

Semantics: $[\![p]\!]_s \subseteq \mathbb{S}_\Omega$ is defined by:

$$\begin{aligned}
[\![@l]\!]_s &= \{l\} \times \mathbb{M} \\
[\![x \leqslant x']\!]_s &= \{\, (l,m) \in \mathbb{S} \mid m(x) \leqslant m(x') \,\} \\
[\![x \leqslant n]\!]_s &= \{\, (l,m) \in \mathbb{S} \mid m(x) \leqslant n \,\} \\
[\![\neg p]\!]_s &= \mathbb{S}_\Omega \setminus [\![p]\!]_s \\
[\![p \wedge p']\!]_S &= [\![p]\!]_s \cap [\![p']\!]_s \\
[\![\Omega]\!]_s &= \{\Omega\}
\end{aligned}$$

This specification language allows to formally express state properties.

Example (Unreachability of control state l_0)**.**

$$[\![\neg @l_0]\!]_s = \mathbb{S}_\Omega \setminus \{\, (l_0, m) \mid m \in \mathbb{M} \,\}$$

Example (Absence of runtime errors)**.**

$$[\![\neg\Omega]\!]_s = \mathbb{S}_\Omega \setminus \{\Omega\} = \mathbb{S}$$

Example (Intermittent invariant)**.** The principle is to attach a local invariant to each control state.

3.5.3 Propositional temporal logic

We now consider the specification of trace properties, using temporal logic. Temporal logic specifies the properties in terms of events that occur at distinct times in the execution (hence the name "temporal"). There are many instances of temporal logic, but we will study a simple one: Pnueli's Propositional Temporal Logic.

```
1 if (x >= 0) {
2     y = x;
3 } else {
4     y = -x;
5 }
```

$$
\begin{aligned}
@l_1 &\implies x \geqslant 0 \\
@l_2 &\implies x \geqslant 0 \wedge y \geqslant 0 \\
@l_2 &\implies x < 0 \\
@l_4 &\implies x < 0 \wedge y > 0 \\
@l_5 &\implies y \geqslant 0
\end{aligned}
$$

Figure 6: The program and the local invariants

Definition (Syntax of PTL (Propositional Temporal Logic))**.** Properties over traces are defined as terms of the form:

$$
\begin{array}{rcll}
t \in \mathbb{L}_{\text{PTL}} & ::= & p & \textit{(state property, i. } p \in \mathbb{L}_{\mathbb{S}}\textit{)} \\
& | & t' \vee t'' & \textit{(disjunction)} \\
& | & \neg t' & \textit{(negation)} \\
& | & \bigcirc t' & \textit{(next)} \\
& | & t' \mathfrak{U} t'' & \textit{(}t' \textit{ until } t''\textit{)}
\end{array}
$$

Figure 7: Propositional Temporal Logical syntax

The semantics of a temporal property is a set of traces, and it is defined by induction over the syntax:

$$
\begin{aligned}
[\![p]\!]_t &= \{\, s \cdot \sigma \mid s \in [\![p]\!]_s \wedge \sigma \in \mathbb{S}^\infty \,\} \\
[\![t_0 \vee t_1]\!]_t &= [\![t_0]\!]_t \cup [\![t_1]\!]_t \\
[\![\neg t_0]\!]_t &= \mathbb{S}^\infty \setminus [\![t_0]\!]_t \\
[\![\bigcirc t_0]\!]_t &= \{\, s \cdot \sigma \mid s \in \mathbb{S} \wedge \sigma \in [\![t_0]\!]_t \,\} \\
[\![t_0 \mathfrak{U} t_1]\!]_t &= \left\{ \sigma \in \mathbb{S}^\infty \;\middle|\; \exists n \in \mathbb{N}, \forall i < n, \sigma_{i\rceil} \in [\![t_0]\!]_t \wedge \sigma_{n\rceil} \in [\![t_1]\!]_t \right\}
\end{aligned}
$$

Many useful operators can be added:

- Boolean constants:

$$
\begin{aligned}
\text{true} &::= (x < 0) \wedge \neg(x < 0) \\
\text{false} &::= \neg\text{true}
\end{aligned}
$$

- Sometime (there exists a rank n at which t holds):

$$\Diamond t ::= \text{true} \mathfrak{U} t$$

- Always (there is no rank at which the negation of t holds):

$$\Box t ::= \neg(\Diamond(\neg t))$$

One can combine these operators to construct interesting properties: $\Diamond\Box t$ means that t is true starting from a certain rank, and $\Box \Diamond t$ means that t happens an infinite number of times.

3.6 Beyond safety and liveness

We now consider other interesting properties of programs, and show that they do not all reduce to trace properties.

For example, one can think about security properties. We can consider just one: "An unauthorized observer should not be able to guess anything about private information by looking at public information".

3.6.1 Assumptions

We let $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_\mathcal{I})$ be a transition system. Its states are of the form $(l, m) \in \mathbb{L} \times \mathbb{M}$. Memory states are of the form $\mathbb{X} \rightarrow \mathbb{V}$. We let $l, l' \in \mathbb{L}$ be the program entry and exit, and $x, x' \in \mathbb{X}$ be the private and public variables. We look at the following security property:

"Observing the value x' at l' gives no information on the value of x at l."

3.6.2 Examples

- A secure program – no information flow, no way to guess x:
$$x' = 84;$$
- An insecure program – explicit information flow, x' gives a lot of information about x, so that we can simply recompute it:
$$x' = x - 2;$$
- An insecure program – implicit information flow, through a test:
$$\text{if}(x < 0)\{x' = 0;\}$$

How can we characterize information flow in the semantic level?

3.6.3 Non-interference

We consider the transformer Φ defined by:

$$\begin{aligned} \Phi : \mathbb{M} &\longrightarrow \mathcal{P}(\mathbb{M}) \\ m &\longmapsto \{\, m' \in \mathbb{M} \mid \exists \sigma = \langle (l, m), \ldots, (l', m') \rangle \in [\![\mathcal{S}]\!] \,\} \end{aligned}$$

Definition (Non-interference)**.** There is no interference between (l, x) and (l', x'), and we write $(l', x') \not\rightsquigarrow (l, x)$, if and only if the following property holds:

$$\forall m \in \mathbb{M}, \forall v_0, v_1 \in \mathbb{V}, \quad \{\, m'(x') \mid m' \in \Phi(m[x \leftarrow v_0]) \,\} = \{\, m'(x') \mid m' \in \Phi(m[x \leftarrow v_1]) \,\}$$

Intuitively, if two observations at point l differ only in the value of x, there is no difference in observation x' at l'. In other words, observing x' at l' (even on many executions) gives no information about the value of x at point l.

Property 3.4. Non-interference is not a trace property.

3.6.4 Dependence properties

Many notions of dependences can be defined, but we will consider just one:

what inputs may have an impact on the observation of a given output

This as many applications in reverse engineering, slicing, and corresponds to the negation of non-interference.

Definition (Interference)**.** There is interference between (l, x) and (l', x'), and we write $(l', x') \rightsquigarrow (l, x)$ if and only if the following property holds:

$$\exists x \in \mathbb{M}, \exists v_0, v_1 \in \mathbb{V}, \quad \{\, m'(x') \mid m' \in \Phi(m[x \leftarrow v_0]) \,\} \neq \{\, m'(x') \mid m' \in \Phi(m[x \leftarrow v_1]) \,\}$$

This expresses that there is at least one case, where the value of x at l has an impact on that of x' at l'. It may not hold even if the computation of x' reads x:

$$x' = 0 \times x;$$

Property 3.5. Interference property is not a trace property.

3.6.5 Hyperproperties

In conclusion, the absence of interference between (l, x) and (l', x') is not a trace property; one cannot describe as the set of programs the semantics of which is included into a given set of traces. It can however be described by a set of sets of traces: we simply collect the set of program semantics that satisfy the property. This is what we call a hyperproperty: trace hyperproperties are described by sets of sets of executions, while trace properties are described by sets of executions.

To disprove the absence of interference (i.e. to show there exists an interference), we simply need to exhibit two finite traces: this is called *2-safety*.

3.7 Conclusion

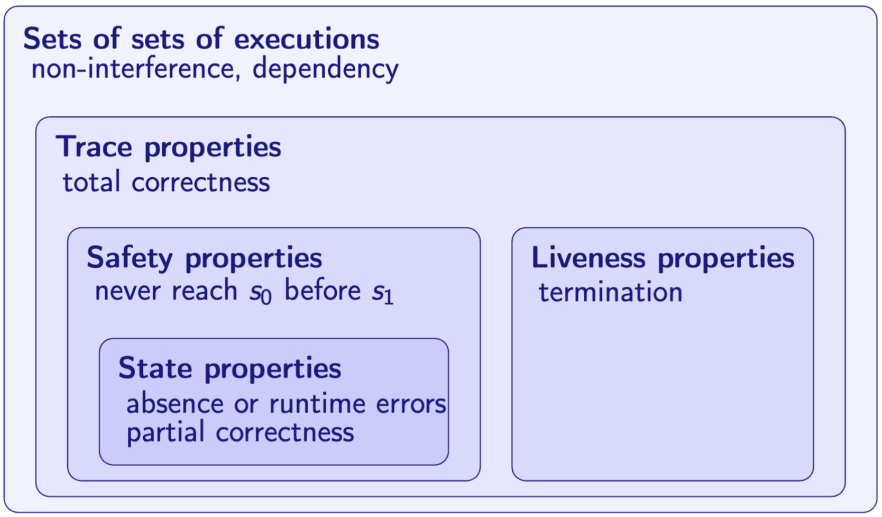


Figure 8: The Zoo of semantic properties

Trace properties allow to express a large range of program properties. Safety represents the absence of bad bahaviors, while liveness expresses the existence of good behaviors. Trace properties can be decomposed as conjuctions of safety and liveness properties, with dedicated proof methods. Nevertheless, some interesting properties are not trace properties: we saw that security properties are sets of sets of executions. Finally, the notion of specification languages is usefull to describe program properties.

4 Denotational semantics

The first chapters of this class focused on *operational semantics*, through state and trace properties. Such semantics are defined as small execution steps, over low-level internal configurations, with transitions chianed to define maximal traces.

In this chapter, we will delve into denotational semantics. We will study direct function from programs to mathematical objects (denotations), defined by induction on the program syntax (compositional), ignoring intermediate steps and execution details. This is a higher-level, more abstract and modular approach, trying to decouple a program meaning from its execution, and focused on the mathematical structures that represent programs.

Consider the following two programs:

```c
int swapped;
do {
    swapped = 0;
    for (int i=1; i<n; i++) {
        if (a[i-1]>a[i]) {
            swap(&a[i-1], &a[i]);
            swapped = 1;
        }
    }
} while (swapped);
```

```ocaml
let rec sort = function
| [] -> []
| a::rest ->
    let lo, hi = List.partition
        (fun y -> y < x) rest
    in
    (sort lo) @ [x] @ (sort hi)
```

Figure 9: Bubble sort in `C` (left) and quicksort in `OCaml` (right)

These programs use different languages, algorithms, programming principles and data types; nevertheless, can we give them the same semantics?

We will study two denotation worlds:

- Imperative programs, which mutate a memory state. The natural denotation of such a program is an input/output function, over a domain $\mathcal{D} \simeq \text{memory} \to \text{memory}$.
- Functional programs, returning a value without any side effect. One can model a program of type `a -> b` as a function $\mathcal{D}_a \to \mathcal{D}_b$.

4.1 Deterministic imperative programs

4.1.1 Syntax

We now consider a simple imperative language: IMP. Variables are in a fixed set: $X \in \mathbb{V}$. Constants are defined as a set $\mathbb{I} := \mathbb{B} \cup \mathbb{Z}$, containing booleans $\mathbb{B} := \{\text{true}, \text{false}\}$ and integers $\mathbb{Z}$. Furthermore, we include unary and binary operators, represented by $\diamond$. Such operators can be integer operations $+, -, \times, /, <, \leqslant$, boolean operations $\neg, \wedge, \vee$, or polymorphic operations $=, \neq$.

expr	::=	X	*(variable)*
	|	c	*(constant)*
	|	$\diamond$ expr	*(unary op.)*
	|	expr $\diamond$ expr	*(binary op.)*

stat	::=	**skip**	*(do nothing)*
	|	$X \leftarrow$ expr	*(assignment)*
	|	stat; stat	*(sequence)*
	|	**if** e **then** s **else** s	*(conditional)*
	|	**while** expr **do** stat	*(loop)*

Figure 10: IMP expressions and statements

4.1.2 Semantics

Expression semantics For expressions, we define a function E such that:

$$E[\![\text{ expr }]\!] : \mathcal{E} \rightharpoonup \mathbb{I}$$

where $\mathcal{E} := \mathbb{V} \to \mathbb{I}$ is an environment mapping variables in $\mathbb{V}$ to values in $\mathbb{I}$. The symbol $\rightharpoonup$ denotes a partial function: it is necessary because some operations are undefined for typing reasons ($1 + \text{true}$, $1 \wedge 2$), or are undefined ($3/0$, ...). We will define E by structural induction on the abstract syntax tree.

$$
\begin{array}{lcll}
\mathsf{E}[\![\, c \,]\!]\,\rho & \stackrel{\text{def}}{=} & c & \in \mathbb{I} \\
\mathsf{E}[\![\, V \,]\!]\,\rho & \stackrel{\text{def}}{=} & \rho(V) & \in \mathbb{I} \\
\mathsf{E}[\![\, -e \,]\!]\,\rho & \stackrel{\text{def}}{=} & -v & \in \mathbb{Z} \quad \text{if } v = \mathsf{E}[\![\, e \,]\!]\,\rho \in \mathbb{Z} \\
\mathsf{E}[\![\, \neg e \,]\!]\,\rho & \stackrel{\text{def}}{=} & \neg v & \in \mathbb{B} \quad \text{if } v = \mathsf{E}[\![\, e \,]\!]\,\rho \in \mathbb{B} \\
\mathsf{E}[\![\, e_1 + e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 + v_2 & \in \mathbb{Z} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{Z}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{Z} \\
\mathsf{E}[\![\, e_1 - e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 - v_2 & \in \mathbb{Z} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{Z}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{Z} \\
\mathsf{E}[\![\, e_1 \times e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 \times v_2 & \in \mathbb{Z} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{Z}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{Z} \\
\mathsf{E}[\![\, e_1 / e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 / v_2 & \in \mathbb{Z} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{Z}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{Z} \setminus \{0\} \\
\mathsf{E}[\![\, e_1 \wedge e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 \wedge v_2 & \in \mathbb{B} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{B}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{B} \\
\mathsf{E}[\![\, e_1 \vee e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 \vee v_2 & \in \mathbb{B} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{B}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{B} \\
\mathsf{E}[\![\, e_1 < e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 < v_2 & \in \mathbb{B} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{Z}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{Z} \\
\mathsf{E}[\![\, e_1 \leq e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 \leq v_2 & \in \mathbb{B} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{Z}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{Z} \\
\mathsf{E}[\![\, e_1 = e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 = v_2 & \in \mathbb{B} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{I}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{I} \\
\mathsf{E}[\![\, e_1 \neq e_2 \,]\!]\,\rho & \stackrel{\text{def}}{=} & v_1 \neq v_2 & \in \mathbb{B} \quad \text{if } v_1 = \mathsf{E}[\![\, e_1 \,]\!]\,\rho \in \mathbb{I}, v_2 = \mathsf{E}[\![\, e_2 \,]\!]\,\rho \in \mathbb{I}
\end{array}
$$

Figure 11: Expression semantics

Statement semantics Similarly, we define for statements a function S of the form:

$$S[\![\ \text{stat}\ ]\!] : \mathcal{E} \rightharpoonup \mathcal{E}$$

It maps an environment before the statement to an environment after the statement. Likewise, it is a partial function due to errors in expressions or non-termination. We also define it by strucutral induction as described in what follows:

- **Skip** – do nothing:
$$S[\![\mathbf{skip}]\!]\rho := \rho$$
- **Assignment** – evaluate expression and mutate environment:
$$S[\![X \leftarrow e]\!]\rho := \rho[X \mapsto v] \quad \text{if } E[\![e]\!]\rho = v$$
where $f[x \mapsto y]$ denotes the function that maps x to y, and any $z \neq x$ to $f(z)$.
- **Sequence** – function composition:
$$S[\![s_1; s_2]\!] := S[\![s_2]\!] \circ S[\![s_1]\!]$$
- **Conditional**:
$$S[\![\mathbf{if}\ \text{e}\ \mathbf{then}\ s_2\ \mathbf{else}\ s_2]\!]\rho := \begin{cases} S[\![s_1]\!]\rho & \text{if } E[\![e]\!]\rho = \text{true} \\ S[\![s_2]\!]\rho & \text{if } E[\![e]\!]\rho = \text{false} \\ \text{undefined} & \text{otherwise} \end{cases}$$
- **Loops** – the semantics of loops must satisfy:
$$S[\![\mathbf{while}\ \text{e}\ \mathbf{do}\ \text{s}]\!] = \begin{cases} \rho & \text{if } E[\![e]\!]\rho = \text{false} \\ S[\![\mathbf{while}\ \text{e}\ \mathbf{do}\ \text{s}]\!](S[\![s]\!]\rho) & \text{if } E[\![e]\!]\rho = \text{true} \\ \text{undefined} & \text{otherwise} \end{cases}$$
This is a recursive definition, and we must therefore prove that the equation has solution(s); in case there are several solutions, there is a single "right" one. To do so, we will use fixpoints of operators over partially ordered sets.

4.1.3 Fixpoint semantics of loops

Definition (Flat ordering)**.** We introduce the *flat ordering* $(\mathbb{I}_\bot, \sqsubseteq)$ on $\mathbb{I}$. We let $\mathbb{I}_\bot := \mathbb{I} \cup \{\bot\}$, which is a pointed set, and define:

$$a \sqsubseteq b \iff a = \bot \vee a = b$$

Every chain is finite, and therefore has a lub $\sqcup$. Thus, $(\mathbb{I}_\bot, \sqsubseteq)$ is a pointed complete partial order (cpo). $\bot$ denotes the value *undefined*: $\sqsubseteq$ is an information order. We can similarly introduce $\mathcal{E}_\bot := \mathcal{E} \cup \{\bot\}$, and note that $(\mathcal{E} \rightharpoonup \mathcal{E}) \simeq (\mathcal{E} \to \mathcal{E}_\bot)$. This allows us to now use total functions only: if the value is undefined, we define its result to $\bot$.

Definition (Poset of continuous partial functions)**.** We introduce a partial order structure on partial function $(\mathcal{E}_\bot \xrightarrow{c} \mathcal{E}_\bot, \dot\sqsubseteq)$.

- $\mathcal{E}_\bot \to \mathcal{E}_\bot$ extends $\mathcal{E} \to \mathcal{E}_\bot$: since the domain of functions is now equal to their co-domain, it allows composition $\circ$. To do this extension, we extend $f \in \mathcal{E} \to \mathcal{E}_\bot$ with $f(\bot) := \bot$. That way, if $S[\![s]\!]x$ is undefined, so is $(S[\![s']\!] \circ S[\![s]\!])x$. Such functions are therefore monotonic[2] and continuous[3].
- We restrict $\mathcal{E}_\bot \to \mathcal{E}_\bot$ to continuous functions $\mathcal{E}_\bot \xrightarrow{c} \mathcal{E}_\bot$.
- $\dot\sqsubseteq$ is the point-wise order on functions, defined by:

$$f \dot\sqsubseteq g \iff \forall x,\ f(x) \sqsubseteq g(x)$$

- $\mathcal{E}_\bot \xrightarrow{c} \mathcal{E}_\bot$ has a least element $\dot\bot := \lambda x.\bot = x \mapsto \bot$.
- Finally, it is also complete for point-wise lub $\dot\sqcup$ of chains, which makes the order structure on partial functions a *cpo*:

$$\dot\sqcup F := x \longmapsto \sqcup \{\, f(x) \mid f \in F \,\}$$

To solve the semantic equation, we use a *fixpoint* of a functional – actually, we will even use the *least fixpoint*, which is more precise for the information order. To do so, we introduce the following operator F:

$$F : (\mathcal{E}_\bot \to \mathcal{E}_\bot) \to (\mathcal{E}_\bot \to \mathcal{E}_\bot)$$

$$F(f)(\rho) = \begin{cases} \rho & \text{if } \mathbb{E}[\![e]\!]\rho = \text{false} \\ f(S[\![s]\!]\rho) & \text{if } E[\![e]\!]\rho = \text{true} \\ \bot & \text{otherwise} \end{cases}$$

Using this operator, we define:

$$S[\![\textbf{while } e \textbf{ do } s]\!] := \operatorname{lfp} F$$

Theorem. $\operatorname{lfp} F$ is well-defined, and we have $\operatorname{lfp} F = \dot\bigsqcup_{n\in\mathbb{N}} F^n(\dot\bot)$.

Proof sketch. We prove that $S[\![\text{stat}]\!]$ is continuous by induction on the syntax, and use Kleene's theorem. The base cases $S[\![\textbf{skip}]\!]$ and $S[X \leftarrow e]$ are continuous, and we use the induction hypotheses and the fact that $\circ$ respects continuity to handle the other cases. □

We have the following:

- $F^0(\dot\bot) = \dot\bot$ is completely undefined, giving no information

[2] $a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$
[3] $f(\sqcup X) = \sqcup \{\, f(x) \mid x \in \mathcal{X} \,\}$

- If the loop is never entered (providing partial information), we have the following environment:
$$F^1(\dot{\bot})(\rho) = \begin{cases} \rho & \text{if } E[\![e]\!]\rho = \text{false} \\ \bot & \text{otherwise} \end{cases}$$
- If the loop is iterated at most once:
$$F^2(\dot{\bot})(\rho) = \begin{cases} \rho & \text{if } E[\![e]\!]\rho = \text{false} \\ S[\![s]\!]\rho & \text{else if } E[\![e]\!](S[\![s]\!]\rho) = \text{false} \\ \bot & \text{otherwise} \end{cases}$$
- If the loop is iterated at most $n-1$ times, the environment is $F^n(\dot{\bot})(\rho)$.
- The environment when exiting the loop, whatever the number of iterations, is $\dot{\bigsqcup}_{n\in\mathbb{N}} F^n(\dot{\bot})$, providing total information.

Remark. *In our semantics, $S[\![\text{stat}]\!]\rho = \bot$ can either mean that the statement starting on input ρ loops for ever, or that it stops prematurely with an error. We could distinguish between the two cases by adding an error value Ω distinct from $\bot$, and propagating it in the semantics, bypassing coputations.*

4.1.4 Error vs. non-termination

4.2 Non-determinism

4.3 Link between operational and denotational semantics

4.4 Higher-order programs

4.5 Recursive domain equations

5 Axiomatic semantics

We previously studied *operational semantics*, which models precisely program execution as low-level transitions between internal states. This was done using transition systems, execution traces, and big-step semantics. We then analysed *denotational semantics*, which maps programs into objects in a mathematical domain; it is higher-level, compositional, and domain oriented.

In this chapter, we will study axiomatic semantics, used to prove properties about programs. We will consider programs annotated with logical assertions, and introduce a rule system to define the validity of assertion. This approach clearly separates programs from specifications, which are user-provided abstractions of the behavior. This enables the use of logic tools and partial automation with increased confidence.

5.1 Specifications

5.1.1 An example of function specification

Consider the following program:

```
int mod(int A, int B) {
    int Q = 0;
    int R = A;
    while (R >= B) {
        R = R - B;
        Q = Q + 1;
    }
    return R;
}
```

Figure 12: An example of `mod` function

We will progressively add assertions in ACSL[4] to give details about the program goal, internal computations, and why and how contracts are fulfilled. We can start by expressing the intended behavior of the function, using the line:

```
//@ ensures \result == A mod B;
```

We can add requirements for the function to actually behave as intended. We say that a requires/ensures pair is a *function contract*. This give:

```
//@ requires A >= 0 && B >= 0;
```

We can even strengthen this requirement to ensure termination:

```
//@ requires A >= 0 && B>0;
```

Finally, we add details about the internal computations:

```
//@ requires A>=0 && B>0;
//@ ensures \result == A mod B;
int mod(int A, int B) {
    int Q = 0;
    int R = A;
    //@ assert A>=0 && B>0 && Q=0 && R==A;
    while (R >= B) {
        //@ assert A>=0 && B>0 && R>=B && A==Q*B+R;
        R = R - B;
        Q = Q + 1;
    }
    //@ assert A>=0 && B>0 && R>=0 && R<B && A==Q*B+R;
    return R;
}
```

Figure 13: The original function with full assertions

Remark. *The variable `Q` is actually not usefull in the program. We could simply remove it without change of behavior. This would nevertheless make the verification harder, as we would*

[4] ANSI/ISO C Specification Language

need to find the condition:

$$\exists Q,\ A = QB + R \wedge 0 \leqslant R < B$$

Such variables are caled ghost variables*, and can be freely added by the assertions to simplify and add expressiveness to proofs. This shows that annotations can be more complex than the program itself.*

Contracts and class invariants are build in a few languages (mostly Eiffel), but are available as a library or external tool in most languages such as C, Java, C#, ... Contracts can be checked dynamically, statically, or inferred statically.

In this course, we will use deductive methods using logic to check statically and partiallyautomatically whether contracts hold.

5.2 Floyd-Hoare logic

5.2.1 Hoare triples and a small language

Definition (Hoare triple)**.** A Hoare triple is of the form:

$$\{P\}\ \text{prog}\ \{Q\}$$

where P (the *precondition*) and Q (the *postcondition*) are *logical assertions* over program variables, and `prog` is a program fragment. As an example, we could have:

$$P := (X \geqslant 0 \wedge Y \geqslant 0) \vee (X < 0 \wedge Y < 0)$$

A triple means that if P holds before `prog` is executed, then Q holds after the execution of `prog`, unless `prog` does not terminate or encounters an error. It is important to note that $\{P\}$ prog $\{Q\}$ expresses *partial correctness*, but does not rule out errors and non-termination. Hoare triples serve as judgments in a proof system.

Language In what follow, we will consider the language defined by the following grammar:

stat	::=	X ← expr	*(assignment)*
	|	**skip**	*(do nothing)*
	|	**fail**	*(error)*
	|	stat; stat	*(sequence)*
	|	**if** expr **then** stat **else** stat	*(conditional)*
	|	**while** expr **do** stat	*(loop)*

Figure 14: The formal grammar of the considered language statements

where $X \in \mathbb{V}$ are integer-valued variables, `expr` are integer arithmetic expressions, which we assume to be deterministic and causing no error. For instance, this can be achieved by assuming that all divisions are explicitly guarded by an **if** construction.

5.2.2 Hoare rules

We now introduce Hoare rules to infer Hoare triples.

Axioms Intuitively, any property true before **skip** is true afterwards. We formalize this using the following axiom:

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}\ \text{Ax.}$$

Similarly, any property is true after fail:

$$\frac{}{\{P\}\ \textbf{fail}\ \{Q\}}\ \text{Ax.}$$

Finally, we can introduce the *assignment axiom*: for P over X to be true after $X \leftarrow e$, P must be true over e before the assignment:

$$\frac{}{\{P[e/X]\}\ X \leftarrow e\ \{P\}}\ \text{Ax.}$$

where $P[e/X]$ is P in which all free occurrences of X are replaced with e. Note that e must be deterministic, and that the rule is *backwards*: P appears as a postcondition.

Example (Assignment examples)**.**

$$\begin{aligned}
\{\text{true}\}\ X \leftarrow 5\ \{X = 5\} \\
\{Y = 5\}\ X \leftarrow Y\ \{X = 5\} \\
\{X + 1 \geqslant 0\}\ X \leftarrow X + 1\ \{X \geqslant 0\} \\
\{\text{false}\}\ X \leftarrow Y + 3\ \{Y = 0 \wedge X = 12\} \\
\{Y \in [0, 10]\}\ X \leftarrow Y + 3\ \{X + Y + 2 \wedge Y \in [0, 10]\}
\end{aligned}$$

Consequence (weaknening) The rule of consequence expresses that we can weaken a Hoare triple by both weakening its postcondition $Q \impliedby Q'$ and strengthening its precondition $P \implies P'$. We assume a logic system to be available to prove formulas on assertions, such as $P \implies P'$.

$$\frac{P \implies P' \qquad Q' \implies Q \qquad \{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}\ \text{Weak.}$$

Example (Consequence rule)**.** The axiom for **fail** can be replaced with:

$$\frac{}{\{\text{true}\}\ \textbf{fail}\ \{\text{false}\}}\ \text{Ax.}$$

as $P \implies$ true and false $\implies Q$ always hold. Another example is:

$$\{X = 99 \wedge Y \in [1, 10]\}\ X \leftarrow Y + 10\ \{X = Y + 10 \wedge Y \in [1, 10]\}$$

Tests To prove that Q holds after a test, we prove that it holds after each branch (s, t) under the assumption that the branch is executed, i.e. e or $\neg e$:

$$\frac{\{P \wedge e\}\ s\ \{Q\} \qquad \{P \wedge \neg e\}\ t\ \{Q\}}{\{P\}\ \textbf{if}\ \mathrm{e}\ \textbf{then}\ \mathrm{s}\ \textbf{else}\ \mathrm{t}\ \{Q\}}\ \text{Test}$$

Example (Test rule)**.**

$$\dfrac{\dfrac{\dfrac{}{\{X < 0\}\ X \leftarrow -X\ \{X > 0\}}\ \text{Ax.}}{\{(X \neq 0)\}\ X \leftarrow -X\ \{X > 0\}}\ \text{Weak.} \qquad \dfrac{\dfrac{}{\{X > 0\}\ \textbf{skip}\ \{X > 0\}}\ \text{Ax.}}{\{(X \neq 0) \wedge (X \geqslant 0)\}\ \textbf{skip}\ \{X > 0\}}\ \text{Weak.}}{\{X \neq 0\}\ \textbf{if}\ X < 0\ \textbf{then}\ X \leftarrow -X\ \textbf{else skip}\ \{X > 0\}}\ \text{Test}$$

Sequences To prove a sequence $s; t$, we must invent an intermediate assertion R implied by P after s, and implying Q after t:

$$\frac{\{P\}\ s\ \{R\} \qquad \{R\}\ t\ \{Q\}}{\{P\}\ s; t\ \{Q\}}\ \text{Seq.}$$

This is also often denoted by $\{P\}\, s\, \{R\}\, t\, \{Q\}$.

Example (Sequence rule)**.**

$$\{X = 1 \wedge Y = 1\}\, X \leftarrow X + 1\, \{X = 2 \wedge Y = 1\}\, Y \leftarrow Y - 1\, \{X + 1 \wedge Y = 0\}$$

Loops To prove that P – true before a loop – still holds after the loop, we can prove that P is a loop invariant: P holds before each loop iteration, before even testing e. This is expressed with the following rule:

$$\frac{\{P \wedge e\}\ s\ \{P\}}{\{P\}\ \textbf{while}\ \mathrm{e}\ \textbf{do}\ \mathrm{s}\ \{P \wedge \neg e\}}\ \text{Loop}$$

In practice, we would rather prove the triple $\{P\}$ **while** e **do** s $\{Q\}$. To do so, it is sufficient to invent an assertion I that:

- holds when the loop starts: $P \implies I$
- is invariant by the body s: $\{I \wedge e\}\ s\ \{I\}$
- implies the assertion when the loop stops: $(I \wedge \neg e) \implies Q$

We can therefore derive the rule:

$$\frac{P \implies I \qquad I \wedge \neg e \implies Q \qquad \dfrac{\{I \wedge e\}\ s\ \{I\}}{\{I\}\ \textbf{while}\ \mathrm{e}\ \textbf{do}\ \mathrm{s}\ \{I \wedge \neg e\}}}{\{P\}\ \textbf{while}\ \mathrm{e}\ \textbf{do}\ \mathrm{s}\ \{Q\}}\ \text{Inv.}$$

Remark. *Hoare logic is parametrized by the choice of logical theory of assertions. The logical theory is used both to prove properties of the form $P \implies Q$, and to simply formulas by replacing a formula with a simpler one, equivalent in a logical sense ($\iff$).*

In practice, we often use first order theories, such as booleans $(\mathbb{B}, \neg, \wedge, \vee)$, Peano arithmetic $(\mathbb{N}, +, \times)$, ZF set theory or theory of arrays. Theories have different expressiveness, decidability and complexity results. This is an important factor when trying to automate program verification.

5.2.3 Summary of Hoare rules

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}\ \text{Ax.} \qquad \frac{}{\{P\}\ \textbf{fail}\ \{Q\}}\ \text{Ax.} \qquad \frac{}{\{P[e/X]\}\ X \leftarrow e\ \{P\}}\ \text{Ax.}$$

$$\frac{\{P \wedge e\}\ s\ \{Q\} \quad \{P \wedge \neg e\}\ t\ \{Q\}}{\{P\}\ \textbf{if}\ \text{e}\ \textbf{then}\ \text{s}\ \textbf{else}\ \text{t}\ \{Q\}}\ \text{Test} \qquad \frac{\{P\}\ s\ \{R\} \quad \{R\}\ t\ \{Q\}}{\{P\}\ s;t\ \{Q\}}\ \text{Seq.}$$

$$\frac{P \implies P' \quad Q' \implies Q \quad \{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}\ \text{Weak.} \qquad \frac{\{P \wedge e\}\ s\ \{P\}}{\{P\}\ \textbf{while}\ \text{e}\ \textbf{do}\ \text{s}\ \{P \wedge \neg e\}}\ \text{Loop}$$

5.2.4 Proof tree example

5.2.5 Invariants and inductive invariants

5.2.6 Auxiliary variables

Auxiliary variables are mathematical variables that do not appear in the program. They are constant during the program execution. Such objects are widely used to simplify proofs, achieve more properties and relative completeness on extended languages (concurrency, recursive procedures).

Example. We consider the simple program:

$$\textbf{if}\ X < Y\ \textbf{then}\ Y \leftarrow X\ \textbf{else skip}$$

and we want to express the fact that at the end of the execution, Y is equal to the minimum value between the initial values of X and Y. We might use the following auxiliary-variables-free Hoare triple to express such a behavior:

$$\{\text{true}\}\ \textbf{if}\ X < Y\ \textbf{then}\ Y \leftarrow X\ \textbf{else skip}\ \{Y = \min(X, Y)\}$$

Nevertheless, this is much less useful as a specification of a min function; indeed, even though the expected triple holds, the following also holds:

$$\{\text{true}\}\ Y \leftarrow X + 1\ \{Y = \min(X, Y)\}$$

Any program ending with $Y < X$ satisfies the conditions. To express a more interesting property – the fact that Y ends with a value that was present initially – we introduce auxiliary variables:

$$\{X = x \wedge Y = y\}\ \textbf{if}\ X < Y\ \textbf{then}\ Y \leftarrow X\ \textbf{else skip}\ \{Y = \min(x, y)\}$$

5.2.7 Link with denotational semantics

5.3 Dijkstra's predicate calculus

We will now study *predicate calculus*, which is calculus to derive preconditions from postconditions. It orders and mechanizes the search for intermediate assertions. Note that similarly to manual techniques, it is easier to go backwards, mainly due to assignments.

5.3.1 Dijkstra's weakest liberal preconditions

Definition (Weakest liberal preconditions)**.** We denote by *wlp* the function such that:

$$\text{wlp} : (\text{prog} \times \text{Prop}) \longrightarrow \text{Prop}$$

and that $\text{wlp}(c, P)$ is the weakest, i.e. most general, precondition ensuring that $\{\text{wlp}(c, P)\}\ c\ \{P\}$ holds. Said otherwise, it is the greatest set that ensures that the computation ends up in P. Formally:

$$\{P\}\ c\ \{Q\} \iff (P \implies \text{wlp}(c, Q))$$

"Liberal" means that we do not care about termination and errors.

Example.

$$\begin{aligned} \text{wlp}(X \leftarrow X + 1, X = 1) &= (X = 0) \\ \text{wlp}(\textbf{while } X < 0 \textbf{ do } X \leftarrow X + 1, X \geqslant 0) &= \text{true} \\ \text{wlp}(X \leftarrow X + 1, X \geqslant 0) &= \text{true} \end{aligned}$$

Note how in cases 2 and 3 we did not take into account non-terminating cases.

Definition (Calculus definition for *wlp*)**.**

$$\begin{aligned} \text{wlp}(\textbf{skip}, P) &:= P \\ \text{wlp}(\textbf{fail}, P) &:= \text{true} \\ \text{wlp}(X \leftarrow e, P) &:= P[e/X] \\ \text{wlp}(s; t, P) &:= \text{wlp}(s, \text{wlp}(t, P)) \\ \text{wlp}(\textbf{if } e \textbf{ then } s \textbf{ else } t, P) &:= (e \implies \text{wlp}(s, P)) \wedge (\neg e \implies \text{wlp}(t, P)) \\ \text{wlp}(\textbf{while } e \textbf{ do } s, P) &:= I \wedge ((e \wedge I) \implies \text{wlp}(s, I)) \wedge ((\neg e \wedge I) \implies P) \end{aligned}$$

Note that **while** loops require providing an invariant predicate I. Intuitively, *wlp* checks that I is an inductive invariant implying P. If so, it returns I, otherwise it returns false. *wlp* is the weakest precondition only if I is well-chosen.

Example. We will compute the weakest liberal precondition of a simple program using the inductive definition of *wlp*:

$$\begin{aligned} &\text{wlp}(\textbf{if } X < 0 \textbf{ then } Y \leftarrow -X \textbf{ else } Y \leftarrow X, Y \geqslant 10) \\ =&(X < 0 \implies \text{wlp}(Y \leftarrow -X, Y \geqslant 10)) \wedge (X \geqslant 0 \implies \text{wlp}(Y \leftarrow X, Y \geqslant 10)) \\ =&(X < 0 \implies -X \geqslant 10) \wedge (X \geqslant 0 \implies X \geqslant 10) \\ =&(X \geqslant 0 \wedge -X \geqslant 10) \wedge (X < 0 \vee X \geqslant 10) \\ =&X \geqslant 10 \vee X \leqslant -10 \end{aligned}$$

As seen in this example, this definition generates complex formulas. It is important to simplify them from time to time to avoid huge computational costs.

In what follows, $A \equiv B$ means that formulas A and B are equivalent. It is stronger that syntactic equality. Formally,

$$\forall \rho, \quad \rho \vDash A \iff \rho \vDash B$$

Property 5.1 (Excluded miracle)**.**

$$\text{wlp}(c, \text{false}) \equiv \text{false}$$

Property 5.2 ($\wedge$-distributivity)**.**

$$\text{wlp}(c, P) \wedge \text{wlp}(d, Q) \equiv \text{wlp}(c, P \wedge Q)$$

Property 5.3 ($\vee$-distributivity)**.**

$$\text{wlp}(c, P) \vee \text{wlp}(d, Q) \equiv \text{wlp}(c, P \vee Q)$$

Note that $\implies$ is always true, while $\impliedby$ is only true for deterministic, error-free programs.

Property 5.4 (Monotonicity)**.**

$$(P \implies Q) \implies (\text{wlp}(c, P) \implies \text{wlp}(c, Q))$$

5.3.2 Strongest liberal postconditions

Similarly, we can define the strongest liberal postcondition.

Definition. We denote by *slp* the function such that:

$$\text{slp} : (\text{prog} \times \text{Prop}) \longrightarrow \text{Prop}$$

and that $\text{slp}(c, P)$ is the strongest, i.e. most general, postcondition ensuring that $\{P\}\ c\ \{\text{slp}(P, c)\}$ holds. Said otherwise, it is the smallest state set. Formally:

$$\{P\}\ c\ \{Q\} \iff (\text{slp}(P, c) \implies Q)$$

Like *wlp*, *slp* does not care about non-termination. It allows forward reasoning.

Property 5.5 (Duality)**.**

$$(P \implies \text{wlp}(c, Q)) \iff (\text{slp}(P, c) \implies Q)$$

Definition (Calculus definition for *slp*)**.**

$$\begin{aligned}
\text{slp}(P, \textbf{skip}) &:= P \\
\text{slp}(P, \textbf{fail}) &:= \text{false} \\
\text{slp}(P, X \leftarrow e) &:= \exists v,\ P[v/X] \wedge X = e[v/X] \\
\text{slp}(P, s; t) &:= \text{slp}(\text{slp}(s, P), t) \\
\text{slp}(P, \textbf{if } e \textbf{ then } s \textbf{ else } t) &:= \text{slp}(P \wedge e, s) \vee \text{slp}(P \wedge \neg e, t) \\
\text{slp}(P, \textbf{while } e \textbf{ do } s) &:= (P \implies I) \wedge (\text{slp}(I \wedge e, s) \implies I) \wedge (\neg e \wedge I)
\end{aligned}$$

Similarly, **while** loops require providing an invariant predicate I.

5.4 Verification conditions

5.4.1 Verification condition generation

The previously introduced notions define a framework to verify programs. How can we automate program verification using logic? Hoare logic is a deductive system and can therefore only automate the checking of proofs. Predicate transformers, such as *wlp* and *slp* calculus, allow constructing (big) formulas mechanically, but invention is still needed, for example in loops.

Such tools can nevertheless be used in verification condition generation: a verificator takes as input a program with annotations, such as contracts and loop invariants, and generates mechanically logic formlas ensuring the correctness. It reduces the input to a mathematical problem, which contains no reference to a program. Finally, an automatic SAT/SMT solver or an interactive theorem prover is used to discharge (prove) the formulas.

To do so, we slightly modifiy the grammar of statements to introduce optional assertions at any point:

$$
\begin{array}{rcll}
\text{stat} & ::= & \text{X} \leftarrow \text{expr} & \\
& | & \textbf{skip} & \\
& | & \textbf{fail} & \\
& | & \text{stat; stat} & \\
& | & \textbf{if } \text{expr } \textbf{then } \text{stat } \textbf{else } \text{stat} & \\
& | & \textbf{while } \text{expr } \textbf{do } \text{stat} & \\
& | & \textbf{assert } \text{expr} & \textit{(assertion)} \\
\\
\text{prog} & ::= & \{\text{Prop}\}\ \text{stat}\ \{\text{Prop}\} &
\end{array}
$$

Figure 15: Statements with assertions and programs annotated with a contract

5.4.2 Verification algorithm

We define two OCaml-like functions, vcg_p and vcg_s, which inductively define an algorithm for verification condition generation.

Program verification vcg_p is of the form:

$$\text{vcg}_\text{p} : \text{prog} \to \mathcal{P}(\text{Prop})$$

Since prog contains only one construction, its definition is simply:

$$\text{vcg}_\text{p}(\{P\}\ c\ \{Q\}) := \text{let } (R, C) = \text{vcg}_\text{s}(c, Q) \text{ in } C \cup \{P \implies R\}$$

Statement verification vcg_s is of the form: $\text{vcg}_\text{s} : (\text{stat} \times \text{Prop}) \to (\text{Prop} \times \mathcal{P}(\text{Prop}))$, and is defined inductively by:

$$
\begin{aligned}
\text{vcg}_\text{s}(\textbf{skip}, Q) &:= (Q, \varnothing) \\
\text{vcg}_\text{s}(X \leftarrow e, Q) &:= (Q[e/X], \varnothing) \\
\text{vcg}_\text{s}(s; t, Q) &:= \text{let } (R, C) = \text{vcg}_\text{s}(t, Q) \text{ in} \\
&\qquad \text{let } (P, D) = \text{vcg}_\text{s}(s, R) \text{ in } (P, C \cup D) \\
\text{vcg}_\text{s}(\textbf{if } e \textbf{ then } s \textbf{ else } t, Q) &:= \text{let } (S, C) = \text{vcg}_\text{s}(s, Q) \text{ in} \\
&\qquad \text{let } (T, D) = \text{vcg}_\text{s}(t, Q) \text{ in } ((e \Rightarrow S) \wedge (\neg e \Rightarrow T), C \cup D) \\
\text{vcg}_\text{s}(\textbf{while}_{\{I\}}\ e \textbf{ do } s, Q) &:= \text{let } (R, C) = \text{vcg}_\text{s}(s, I) \text{ in } (I, C \cup \{(I \wedge e) \Rightarrow R, (I \wedge \neg e)\ Q\}) \\
\text{vcg}_\text{s}(\textbf{assert } e, Q) &:= (e \Rightarrow Q, \varnothing)
\end{aligned}
$$

We use *wlp* to infer assertions automatically when possible. $\text{vcg}_\text{s}(c, P) = (P', C)$ propagates postconditions backwards and accumulates into C the verification conditions (from loops).

Example. Consider the following program:

$$X \leftarrow 1\,;\, I \leftarrow 0\,;\, \textbf{while } I < N \textbf{ do } (X \leftarrow 2X; I \leftarrow I + 1)$$

We want to prove that under the precondition $\{N \geqslant 0\}$, the postcondition $\{X = 2^N\}$ holds. To do so, we anotate the loop with the invariant:

$$\{X = 2^I \ \wedge 0 \leqslant I \leqslant N\}$$

Applying the foredescribed algorithm, we obtain three verification cconditions:

$$\begin{aligned}
C_1 &:= (X = 2^I) \wedge (0 \leqslant I \leqslant N) \wedge (I \geqslant N \implies X = 2^N) \\
C_2 &:= (X = 2^I) \wedge (0 \leqslant I \leqslant N) \wedge (I < N \implies 2X = 2^{I+1} \wedge 0 \leqslant I + 1 \leqslant N) \\
C_3 &:= N \geqslant 0 \implies 1 = 2^0 \wedge 0 \leqslant 0 \leqslant N
\end{aligned}$$

These three conditions can be checked independently using a SAT solver.

5.4.3 What about real languages?

We described verification rules in a minimal example; in a real language such as C, the rules are not so simple. Take as an example the assignement rule:

$$\frac{}{\{P[e/X]\} \ X \leftarrow e \ \{P\}} \text{ Ax.}$$

It requires that e has no effect, that there is no pointer aliasing, that e has no runtime error... Moreover, the operations in the program and in the logic may not match: for integers, logic uses $\mathbb{Z}$ while computers use $\mathbb{Z}/2^n\mathbb{Z}$. The same remark applies to continuous sets (logic uses $\mathbb{Q}$ or $\mathbb{R}$ while program use floating-point numbers) which might cause rounding-related differences. Finally, a logic for pointers and dynamic allocation is also required.

Some problems can be circumvented: you can see for instance how solutions to such differences have be implemented in the tool Why3.

5.5 Total correctness (termination)

5.5.1 Total correctness

We now want to take into account the termination of our program. To do so, we introduce a new type of Hoare triple:

$$[P] \ \text{prog} \ [Q]$$

It has the following meaning: if P holds before prog is executed, then prog always terminates and Q holds after its execution. To adapt our Hoare rules, we only need to change the rule for **while**, as it is the only source of non-termination:

$$\frac{\forall t \in W, \ [P \wedge e \wedge u = t] \ \ s \ \ [P \wedge u \prec t]}{[P] \ \ \textbf{while} \ \text{e} \ \textbf{do} \ \text{s} \ \ [P \wedge \neg e]} \text{ Loop}$$

in which we require $(W, \prec)$ to be a well-founded ordering. This ensures that we cannot decrease infinitely by $\prec$ in W. In general, we simply use $(\mathbb{N}, <)$, even though lexicographic orders and ordinals can also come handy. Intuitively, in addition to the loop invariant P, we invent an expression u that strictly decreases by s. Such a u is called a *ranking function*; often, $\neg e \implies u = 0$: u counts the number of steps until termination.

To simplify, we can decompose a proof of total correctness into two proofs. The first proof is a proof of partial correctness $\{P\} \ c \ \{Q\}$, ignoring termination. The second one is a proof

of termination, meaning $[P]\ c\ [\text{true}]$, ignoring the specification. We must still include the precondition P as the program may not terminate for all inputs. This decomposition can be justified by:

$$\frac{\{P\}\ c\ \{Q\} \qquad [P]\ c\ [\text{true}]}{[P]\ c\ [Q]}$$

Example. For this example, we will use a simpler rule for integer ranking function over $(\mathbb{N}, \leqslant)$ using an integer expression r over program variables:

$$\frac{\forall n \in \mathbb{N},\ [P \wedge e \wedge r = n]\ s\ [P \wedge r < n] \qquad (P \wedge e) \implies (r \geqslant 0)}{[P]\ \textbf{while}\ \text{e}\ \textbf{do}\ \text{s}\ [P \wedge \neg e]}\ \text{Loop}$$

We consider the following program:

$$p := \textbf{while}\ I < N\ \textbf{do}\ I \leftarrow I + 1; X \leftarrow 2X$$

Using $r := N - I$ and $P := \text{true}$ as ranking function and precondition, we have:

$$\frac{\forall n \in \mathbb{N},\ [I < N \wedge N - I = n]\ s\ [N - I = n - 1] \qquad (I < N) \implies (N - I \geqslant 0)}{[\text{true}]\ p\ [I \geqslant N]}\ \text{Loop}$$

5.5.2 Weakest precondition

Similarly to *wlp*, we can define the weakest precondition; it will additionally impose termination, conversely to the liberal precondition:

$$\text{wp} : \text{prog} \times \text{Prop} \to \text{Prop}$$

As before, it is defined as the weakest precondition such that:

$$[P]\ c\ [Q] \iff (P \implies \text{wp}(c, Q))$$

Only the definition for **while** needs to be modified:

$$\begin{aligned}\text{wp}(\textbf{while}\ \text{e}\ \textbf{do}\ \text{s}, P) := {} & I \wedge (I \implies v \geqslant 0) \\ & \wedge \forall n, ((e \wedge I \wedge v = n) \Rightarrow \text{wp}(s, I \wedge v < n)) \\ & \wedge ((\neg e \wedge I) \Rightarrow P)\end{aligned}$$

The invariant predicate I is combined with a variant expression v, which is positive and decreases at each loop iteration. We can define similarly a strongest postcondition.

5.6 Non-determinism

We model non-determinism with the statement $X \leftarrow ?$, meaning that X is assigned a random value. We can also model $X \leftarrow [a, b]$ (assigning to X a random value in the $[a, b]$ interval) by:

$$\textbf{if}\ X < A \vee X > b\ \textbf{then fail}$$

5.6.1 Non-detereminism in Hoare logic

To handle non-determinism in Hoare logic, we add the following axiom:

$$\frac{}{\{\forall X, P\}\ X \leftarrow ?\ \{P\}}\ \text{Ax.}$$

This translates the fact that if P holds whatever the value of X is, P is true after assigning a random value to X. Often, X does not appear in P and we simply write:

$$\frac{}{\{P\}\ X \leftarrow ?\ \{P\}}\ \text{Ax.}$$

Example. We consider a simple program in which we save the value of X before assigning a random value to it, and finally restoring its value. We prove that the value of X stays the same after the execution:

$$\frac{\dfrac{}{\{X = x\}\ Y \leftarrow X\ \{Y = x\}} \quad \dfrac{}{\{Y = x\}\ X \leftarrow ?\ \{Y = x\}} \quad \dfrac{}{\{Y = x\}\ X \leftarrow Y\ \{X = x\}}}{\{X = x\}\ X \leftarrow X; X \leftarrow ?; X \leftarrow Y\ \{X = x\}}\ \text{Seq.}$$

5.6.2 Non-determinism in predicate calculus

We complete the structural induction of the definition of *wlp* and *slp*, using the following definition:

$$\text{wlp}(X \leftarrow ?, P) := \forall X, P$$

translating the fact that P must hold whatever the value of X is before the assignment. For *slp*:

$$\text{slp}(P, X \leftarrow ?) := \exists X, P$$

translating the fact that if P held for one value of X, P holds for all values of X after the assignment. Said otherwise, if P held before $X \leftarrow ?$ and holds after, one can only say that there exists a certain value (the random value) for which P holds.

5.6.3 Link with operational semantics

We briefly analyse the link with operational semantics seen as transition systems. We see predicates P as sets of states: $P \subseteq \Sigma$, and commands c as transition relations: $c \subseteq \Sigma \times \Sigma$. We want to define *slp* and *wlp* as operational predicates, i.e. sets of states.

To do so, we define for a relation $\tau \subseteq \Sigma \times \Sigma$ and a predicate $P \subseteq \Sigma$:

$$\begin{aligned} \text{post}[\tau](P) &:= \{\, \sigma' \mid \exists \sigma \in P, (\sigma, \sigma') \in \tau \,\} \\ \text{pre}[\tau](P) &:= \{\, \sigma \mid \forall \sigma' \in \Sigma, (\sigma,\ \sigma') \in \tau \implies \sigma' \in P \,\} \end{aligned}$$

Using these definitions, we have:

$$\begin{aligned} \text{slp}(P, c) &= \text{post}[c](P) \\ \text{wlp}(c, P) &= \ \text{pre}[c](P) \end{aligned}$$

5.7 Arrays

We enrich our language with a set $\mathbb{A}$ of *array variables*, array access in expressions: $A(\text{expr})$, and array assignment: $A(\text{expr}) \leftarrow \text{expr}$, for some $A \in \mathbb{A}$. Note that arrays have unbouded size here, we do not care about overflow.

A natural idea is to generalize the assignment axiom:

$$\frac{}{\{P[f/A(e)]\}\ A(e) \leftarrow f\ \{P\}}\ \text{Ax.}$$

but this is not sound, due to aliasing.

Example. We would derive the invalid triple:

$$\{A(J) = 1 \wedge I = J\}\ A(I) \leftarrow 0\ \{A(J) = 1 \wedge I = J\}$$

as $(A(J) = 1)[0/A(I)] = (A(J) = 1)$.

The solution to this issue is to use a specific theory of arrays, enriching the assertion language with expressions $A\{e \mapsto f\}$, meaning the array equal to A except that index e maps to value f. This allows to add the axiom:

$$\frac{}{\{P[A\{e \mapsto f\}/A]\}\ A(e) \leftarrow f\ \{P\}}\ \text{Ax.}$$

Intuitively, we use functional arrays in the logic world. We also add logical axioms to reason about our arrays in assertions:

$$\frac{}{A\{e \mapsto f\}(e) = f}\ \text{Ax.} \qquad \frac{}{(e \neq e') \implies (A\{e \mapsto f\}(e') = A(e'))}\ \text{Ax.}$$

Example (Swap)**.** We consider the program:

$$p := T \leftarrow A(I); A(I) \leftarrow A(j); A(J) \leftarrow T$$

which swaps values of A at indices I and J. We wish to prove that the values are indeed swapped:

$$\{A(I) = x \wedge A(J) = y\}\ p\ \{A(I) = y \wedge A(J) = x\}$$

We propagate $A(I) = y$ backwards by the assignment rule, and we get:

$$\begin{aligned} &A\{J \mapsto T\}(I) = y \\ &A\{I \mapsto A(J)\}\{J \mapsto T\}(I) = y \\ &A\{I \mapsto A(J)\}\{J \mapsto A(I)\}(I) = y \end{aligned}$$

We now consider two cases.

- If $I = J$, then $A\{I \mapsto A(J)\}\{J \mapsto A(I)\} = A$ and therefore

$$A\{I \mapsto A(J)\}\{J \mapsto A(I)\}(I) = A(I) = A(J)$$

- If $I \neq J$, then

$$A\{I \mapsto A(J)\}\{J \mapsto A(I)\}(I) = A\{I \mapsto A(J)\}(I) = A(J)$$

In both cases, we get $A(J) = y$ in the precondition. Likewise, $A(I) = x$ in the precondition.

5.8 Concurrency

We want to add a syntax to build concurrent programs. We add a parallel composition statement:

$$\text{stat} \mid\mid \text{stat}$$

We want its semantics $s_1||s_2$ to express the execution of s_1 and s_2 in parallel, allowing an arbitrary interleaving of atomic statements. It terminates when both s_1 and s_2 terminate. The first idea to include it in Hoare logic is to add the following rule:

$$\frac{\{P_1\}\ s_1\ \{Q_1\} \qquad \{P_2\}\ s_2\ \{Q_2\}}{\{P_1 \wedge P_2\}\ s_1\ ||s_2\ \{Q_1 \wedge Q_2\}}\ \text{Conc.}$$

Nevertheless, this rule is unsound.

Example. We define $s_1 := X \leftarrow 1$ and $s_2 := \textbf{if } X = 0 \textbf{ then } Y = 1$, and we derive:

$$\frac{\dfrac{}{\{X = Y = 0\}\ s_1\ \{X = 1 \wedge Y = 0\}} \qquad \dfrac{}{\{X = Y = 0\}\ s_2\ \{X = 0 \wedge Y = 1\}}}{\{X = Y = 0\}\ s_1\ ||s_2\ \{\text{false}\}}\ \text{Conc.}$$

The solution to this issue is to require that the proofs of $\{P_1\}\ s_1\ \{Q_1\}$ and $\{P_2\}\ s_2\ \{Q_2\}$ must not interfere.

Definition (Interference freedom)**.** Given proofs Δ_1 and Δ_2 of $\{P_1\}\ s_1\ \{Q_1\}$ and $\{P_2\}\ s_2\ \{Q_2\}$, we say that Δ_1 do not interfere with Δ_2 if:

for any Φ appearing before a statement in Δ_1, for any $\{P_2'\}\ s_2'\ \{Q_2'\}$ appearing in Δ_2,
$\{\Phi \wedge P_2'\}\ s_2'\ \{\Phi\}$ and $\{Q_1 \wedge P_2'\}\ s_2'\ \{Q_1\}$ hold

Intuitively, the assertions used to prove $\{P_1\}\ s_1\ \{Q_1\}$ are stable by s_2.

Example. Given the same statements $s_1 := X \leftarrow 1$ and $s_2 := \textbf{if } X = 0 \textbf{ then } Y = 1$, we derive:

$$\frac{\dfrac{}{\{X = 0 \wedge Y \in [0,1]\}\ s_1\ \{X = 1 \wedge Y \in [0,1]\}} \qquad \dfrac{}{\{X \in [0,1] \wedge Y = 0\}\ s_2\ \{X \in [0,1] \wedge Y \in [0,1]\}}}{\{X = Y = 0\}\ s_1 \mid\mid s_2\ \{X = 1 \wedge Y \in [0,1]\}}$$

Another issue remains in the handling of concurrent programs, which is rule completeness. Take as an example:

$$\{X = 0\}\ X \leftarrow X + 1 \mid\mid X \leftarrow X + 1\ \{X = 2\}$$

This Hoare triple is valid, but no proof of it can be derived. A solution to this is to include auxiliary variables to introduce explicitly program points and program counters.

Example. We change our program into:

$$^{l_1}X \leftarrow X + 1^{l_2} \mid\mid {}^{l_3}X \leftarrow X + 1^{l_4}$$

with auxiliary variables $pc_1 \in \{1, 2\}$, $pc_2 \in \{3, 4\}$. We can now express that a process is at a given control point and distinguish assertions based on the location of other processes.

Auxiliary variables make the proof method complete.

Conclusion

Logic allows us to reason about program correctness; verification can be reduced to proofs of simple logic statements. The main issue of this approach to verification is automation: annotations are required – taking the form of loop invariants, contracts – and verification conditions must be proven. To scale up to realistic programs, we need to automate as much as possible.

Some solutions are provided by automatic logic solvers to discharge proof obligations. Another option is abstract interpretation to approximate the semantics: it is fully automatic and is able to infer invariants.

6 Types

6.1 Introduction

The purpose of typing is to avoid errors during the execution of programs, by restricting the programs accepted at compilation time. It helps the compiler produce efficient code, and document properties of programs (e.g. in OCaml, typing can give important hints about the content of a function).

In this course, we look at typing from a formal and semantic view: what semantics can we give to types and typing? What semantic information is guaranteed by types?

We won't discuss type systems in language design and implementation, relations between type theory and proof theory, and type theory as an alternative to set theory.

6.1.1 Classification

A type is a set of values with a specific machine representation. Often, distinct types denote non-overlapping value sets, but this is not always the case: `short/int/long` in C, or subtyping in Java and C++ show that some values might have different types. Variables are assigned a type that defines its possible values.

Static vs. dynamic typing The typing is said to be *static* when the type of each variable is known at compile time. This is the case of languages like C, Java, OCaml, ... Conversely, *dynamic* typing allows the type of each variable to be discovered during the execution, and may change, like in Python or Javascript.

Strongly vs. loosely typed languages *Loose* typing does not prevent invalid construction and use: in C, C++, or assembly, an integer can be view as a pointer without an error from the compiler. *Strong* typing detects all type errors – whether at compile time or at run time – this is the case in Java, OCaml, Python, and Javascript.

Theorem (Static strong typing)**.** Well-typed programs cannot go wrong.

Type checking vs. type inference Type *checking* checks the consistency of variable use according to user declarations (C, Java), while type *inference* discovers almost automatically a most general type, consistent with the use of the variable (OCaml, except modules...).

Example (Javascript, a dynamically typed safe language)**.** Dynamic type checking comes with multiple drawbacks: errors are detected late, during execution. This has the tendency to reduce the amount of type errors by adding implicit conversions, which creates complex and

unpredictable semantics. It is costly in time, as it requires many type checks, and an optimizing compiler or JIT generates type-specialized versions of the code. Finally, it is costly in memory, as we must tag each value wit type information at run-time.

6.1.2 Overview

Our goal will be to introduce strong static typing for imperative programs. We will follow a classic workflow to introduce types: we will design a type system, which is a set of logical rules stating whether a program is "well typed"; then, we will prove the soundness with respect to the operational semantics; finally, we will design algorithms to check typing from user-given type annotations, or to infer type annotations that make the program well typed. Finally, we will take a less classic view, typing design by abstraction of the semantics, which will be sound by construction: this is static analysis.

6.2 Type systems

6.2.1 Simple imperative language

We once again use the simple imperative language introduced previously:

$$
\begin{array}{rcll}
\text{expr} & ::= & \text{X} & \textit{(variable)} \\
& | & \text{c} & \textit{(constant)} \\
& | & \diamond\ \text{expr} & \textit{(unary operation)} \\
& | & \text{expr} \diamond \text{expr} & \textit{(binary operation)} \\
\\
\text{stat} & ::= & \text{X} \leftarrow \text{expr} & \textit{(assignment)} \\
& | & \textbf{skip} & \textit{(do nothing)} \\
& | & \text{stat; stat} & \textit{(sequence)} \\
& | & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stat}\ \textbf{else}\ \text{stat} & \textit{(conditional)} \\
& | & \textbf{while}\ \text{expr}\ \textbf{do}\ \text{stat} & \textit{(loop)} \\
& | & \textbf{local}\ \text{X}\ \textbf{in}\ \text{stat} & \textit{(loop)}
\end{array}
$$

where constants c are taken in the set $\mathbb{I} := \mathbb{Z} \cup \mathbb{B}$ (integers and booleans), operators $\diamond$ in $\{+, -, \times, /, <, \leqslant, \neg, \wedge, \vee, =, \neq\}$, and variables in the set $\mathbb{V}$ of all program variables. Note that variables are now local, with limited scope, and must be declared with no type information... yet!

6.2.2 Deductive systems

A deductive system is a set of axioms and logical rules to derive theorems, which define what is provable in a formal way. We introduce *judgments* $\Gamma \vdash \text{Prop}$, a fact meaning "under hypotheses Γ, we can prove Prop". Similarly to Hoare, logic, we structure rules (hoptheses-conclusion rules, and axioms) into proof trees, which complete application of rules from axioms to conclusion.

6.2.3 Typing judgments

We define simple types:

$$
\begin{array}{rcll}
\text{type} & ::= & \textbf{int} & \textit{(integers)} \\
& | & \textbf{bool} & \textit{(booleans)}
\end{array}
$$

To define hypotheses Γ for typing judgments, we introduce a set of type assignments $X : t$ with $X \in \mathbb{V}$, $t \in$ type, intuitively meaning that variable V has type t.

Judgments can be of two types:

- $\Gamma \vdash$ stat, meaning that given the type assignments Γ, stat is well typed
- $\Gamma \vdash$ expr : type, meaning that given the type of variables γ, expr is well-typed and has type *type*

We add the following rules and axioms to derive theorems:

$$\frac{}{\Gamma \vdash c : \mathbf{int}}\ (c \in \mathbb{Z}) \qquad \frac{}{\Gamma \vdash c : \mathbf{bool}}\ (c \in \mathbb{B}) \qquad \frac{}{\Gamma \vdash X : t}\ ((X : t) \in \Gamma)$$

$$\frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash -e : \mathbf{int}} \qquad \frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \neg e : \mathbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \diamond e_2 : \mathbf{int}}\ (\diamond \in \{+, -, \times, /\})$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \diamond e_2 : \mathbf{bool}}\ (\diamond \in \{=, \neq, <, \leqslant\})$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool}}{\Gamma \vdash e_1 \diamond e_2 : \mathbf{bool}}\ (\diamond \in \{=, \neq, \wedge, \vee\})$$

Figure 16: Expression typing rules

Note that the syntax of an expression uniquely identifies a rule to apply, except for $=$ and $\neq$, where it depends upon the choice of a type for e_1 and e_2.

6.2.4 Statement typing

We introduce the following rules for statement typing:

$$\frac{}{\Gamma \vdash \mathbf{skip}} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{skip}}\ ((X : t) \in \Gamma)$$

$$\frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1; s_2} \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2 \quad \Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2}$$

$$\frac{\Gamma \vdash s \quad \Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ s} \qquad \frac{\Gamma \cup \{(X : t)\} \vdash s}{\Gamma \vdash \mathbf{local}\ X\ \mathbf{in}\ s}$$

Similarly, the syntax of a statement uniquely identifies a rule to apply, up to the choice of t in the rule for local variables.

Definition (Well-typed statement)**.** A statement s is well-typed if we can prove $\varnothing \vdash s$.

6.3 Soundness of typing

We want to prove the following result:

Theorem. Well-typed program "cannot go wrong".

The operational semantics has several kinds of errors: type mismatch in operators ($1 \vee 2$, $\text{true} + 2$), and value errors (divide or modulo by 0, use of uninitialized variables). Typing seeks only to prevent statically the first kind of errors.

Value errors can be prevented with static analysis; this is a much more complex and costly process, which we will discuss later in the course. Typing aims at a sweet point, to detect at compile-time all errors of a certain kind.

Property 6.1 (Soundness)**.** Well-typed programs have no type mismatch error. It is proved based on an operational semantic of the program.

6.3.1 Expression of denotational semantics with explicit errors

TODO.

6.3.2 Statement of operational semantics with explicit errors

TODO.

Definition (Operational semantics)**.** We consider maximal execution traces:

$$\begin{aligned} t[\![s]\!] := &\{\, (\sigma_0, \dots, \sigma_n) \mid \sigma_0 \in \mathbb{I}, \sigma_n \in B, \forall i < n, \sigma_i \to \sigma_{i+1} \,\} \\ &\cup \{\, (\sigma_0, \dots) \mid \sigma_0 \in \mathbb{I}, \forall i \in \mathbb{N}, \sigma_i \to \sigma_{i+1} \,\} \end{aligned}$$

Theorem (Type soundness)**.** Typing is sound, i.e. well-typed programs cannot produce errors. Formally:

$$s \text{ is well-typed } \implies \forall (\sigma_0, \dots, \sigma_n) \in t[\![s]\!], \sigma_n \neq \Omega_t$$

6.4 Typing checking

6.4.1 Type declarations

How do we prove that a program is well-typed?

An approach is bottom-up reasoning: we construct a proof tree ending in $\varnothing \vdash s$ by applying rules "in reverse". Given a conclusion, there is generally only one rule to apply. The only rule that requires imagination is

$$\frac{\Gamma \cup \{(X : t)\} \vdash s}{\Gamma \vdash \textbf{local } X \textbf{ in } s}$$

where t is assumed to be a free variable. We need to guess a good t, i.e. a type for X, that makes the proof work. Similarly, to type $\Gamma \vdash e_1 = e_2 : \textbf{bool}$, we also have to choose between $\Gamma \vdash e_1 : \textbf{bool}$ and $\Gamma \vdash e_1 : \textbf{int}$. In other words, we would need to know the types of the variables, which we will use to deduce the types of the expressions.

A common solution in programming languages is to ask the programmer to add type information to all variables declarations: we change the syntax of declaration statements into:

$$\begin{array}{rcl} \text{stat} & ::= & \textbf{local } X : \text{type } \textbf{in } \text{stat} \\ & | & \ldots \end{array}$$

Under these assumptions, the typing rule for local variable declarations becomes deterministic:

$$\frac{\Gamma \cup \{(X : t)\} \vdash s}{\Gamma \vdash \textbf{local } X : t \textbf{ in } s}$$

6.4.2 Type propagation in expressions

6.4.3 Type propagation in statements

6.5 Type inference

Can we avoid specifying types in the progam?

Specifying types is a handy solution from the semantics point of view, but it is unpleasant in practice – hopefully, automatic type inference allow us to guess the type of a variable.

6.5.1 Automatic type inference

To do so, each variable $X \in \mathbb{V}$ is assigned a *type variable* t_X, and we generate a set of *type constraints* ensuring that the program is well typed. Thus, we simply need to solve the constraint system to infer a type value for each type variable.

To express type constraints, we need equalities on types and type variables:

$$\begin{array}{rcll} \text{type constr} & ::= & \text{type expr} = \text{type expr} & \textit{(type equality)} \\ \\ \text{type expr} & ::= & \textbf{int} & \textit{(integers)} \\ & | & \textbf{bool} & \textit{(booleans)} \\ & | & t_X & \textit{(type variable for } X \in \mathbb{V}\textit{)} \end{array}$$

6.5.2 Generating type constraints

Expressions In a method similar to type propagation, we will inductively define a function

$$\tau_e : \text{expr} \to (\text{type expr} \times \mathcal{P}(\text{type constr}))$$

which maps to an expression the type of the expression (possibly a type variable) and a set of constraints to satisfy to ensure it is well typed. No type environment is need: a variable X has a symbolic type t_X. In what follows, $(t_i, C_i) := \tau_e(e_i)$.

$$\begin{array}{rl} & \tau_e(c \in \mathbb{Z}) := (\textbf{int}, \varnothing) \\ & \tau_e(c \in \mathbb{B}) := (\textbf{bool}, \varnothing) \\ & \tau_e(X) := (t_X, \varnothing) \\ & \tau_e(-e_1) := (\textbf{int}, C_1 \cup \{t_1 = \textbf{int}\}) \\ & \tau_e(\neg e_1) := (\textbf{bool}, C_1 \cup \{t_1 = \textbf{bool}\}) \\ \forall \diamond \in \{+, -, \times, /\}, & \tau_e(e_1 \diamond e_2) := (\textbf{int}, C_1 \cup C_2 \cup \{t_1 = \textbf{int}, t_2 = \textbf{int}\}) \\ \forall \diamond \in \{<, \leqslant\}, & \tau_e(e_1 \diamond e_2) := (\textbf{bool}, C_1 \cup C_2 \cup \{t_1 = \textbf{int}, t_2 = \textbf{int}\}) \\ \forall \diamond \in \{\wedge, \vee\}, & \tau_e(e_1 \diamond e_2) := (\textbf{bool}, C_1 \cup C_2 \cup \{t_1 = \textbf{bool}, t_2 = \textbf{bool}\}) \\ \forall \diamond \in \{=, \neq\}, & \tau_e(e_1 \diamond e_2) := (\textbf{bool}, C_1 \cup C_2 \cup \{t_1 = t_2\}) \end{array}$$

Typing $e_1 = e_2$ and $e_1 \neq e_2$ reduces to ensuring an equal type for e_1 and e_2.

Statements Similarly, we inductively define τ_s of the form:

$$\tau_s = \text{stat} \to \mathcal{P}(\text{type constr})$$

We return a set of constraints to satisfy to ensure it is well typed. For simplicity, scoping in **local** $X \in s$ is not handled: we assign a single type for all the local variables with the same name.

$$\begin{aligned}
\tau_s(\textbf{skip}) &:= \varnothing \\
\tau_s(s_1; s_2) &:= \tau_s(s_1) \cup \tau_s(s_2) \\
\tau_s(X \leftarrow e) &:= C \cup \{t_X : t\} \\
\tau_s(\textbf{while } e \textbf{ do } s) &:= \tau_s(s) \cup C \ \cup \{t = \textbf{bool}\} \\
\tau_s(\textbf{local } X \textbf{ in } s) &:= \tau_s(s)
\end{aligned}$$

where $(t, C) := \tau_e(e)$.

6.6 Object-oriented languages

6.7 Types as semantic abstraction

Conclusion

7 Complement: SAT and Satisfiability Modulo Theories (SMT)

Since 2000, SAT solvers have become extremely performant. SMT is the extension of SAT with clauses instead of simply variables.

7.1 Modern SAT solvers

7.1.1 The SAT problem

Given a boolean formula without quantifiers, e.g.

$$(p \vee q \vee \neg r) \wedge (r \vee \neg p)$$

we try to find if this formula is satisfiable. Techniques to do so include truth tables, resolution-based procedures, backtracking-based procedure. Since the 80s, there's been a focus on variable selection heuristics, associated with search-pruning techniques (non-chronological backtracking, learning clauses): these are the CDCL algorithms (Conflicts-Driven Clause Learning). New techniques such as indexing and scoring have since emerged.

7.1.2 Resolution

Resolution uses a proof-finder procedure, which tries to reduce the equation.

The state of the procedure is represented by a variable F, to which we can apply some rules (Resolve, Empty, Tauto, Subsume, Fail). This allows to construct a proof tree.

7.1.3 DPLL

DPLL is a model-finder procedure that builds incrementally a model M for a CNF formula F by

- deducing the truth value of a literal l from M and F by Boolean Constraint Propagations (BCP):
$$\text{If } C \vee l \in F \text{ and } M \vDash \neg C \text{ then } l \text{ must be true}$$
- guessing the truth value of an unassigned literal:
$$\text{If } M \cup \{l\} \text{ leads to a model for which } F \text{ is unsatisfiable}$$
$$\text{then } \textit{backtrack} \text{ and try } M \cup \{\neg l\}$$

Similarly, DPLL is a set of rules (Success, Unit, Decide, Backtrack, Fail).

7.1.4 Clause conflict, backjump

DPLL remains slow: instead of backtracking, we would prefer to *backjump* multiple steps before, to the node where the conflit has indeed been decided. Conflicts are reflected by backjump clauses. Given a backjump clause $C \vee l$, backjumping can undo several decisions at once: it goes back to the assignment M where $M \vDash \neg C$.

7.1.5 CDCL algorithm

Conflict-Driven Clause Learning SAT solvers (CDCL) add backjump clauses to M as learned clauses (or lemmas) to prevent future similar conflicts.

Lemmas can also be removed from M.

The algorithm now has two modes: search and resolution.

7.1.6 Heuristics, Two-watched literals

The Variable State Independent Decaying Sum (VSIDS) heuristic associates a score to each literal in order to select the literal with the hightest score when Decide is used.

CDCL performances are tightly related to their learning clause management. Keeping too many clauses decrease the BCP efficency, but cleaning out too many clauses break the overall learning benefit.

Quality measures for learning clauses are based on scores associated with learned clauses.

BCP represents 80% of SAT-solver runtime. The two watched literals technique assigns two non-false watched literals per clause. Only if one of the two watched literal becomes false, the clause is inspected. If the other watched literal is assigned to true, the do nothing. Otherwise, try to find another (unassigned) watched literal. If no such literal exists, then apply Backjump. If the only possible literal is the oother watched literal of the clause, then apply Unit.

Its main advantages are that clauses are inspected only when watched literal are assigned, and that no updating is required when backjumping.

7.2 Satisfiability Modulo Theories (SMT)

7.2.1 Introduction

The fundamental goal of research in SAT was to be able to solve first-order logic.

The SMT problem asks to decide of the satisfiability of logical formulae with some specific theory symbols.

In theory, SMT solvers can take as an input any first order logic formula. In pratcice, most SMT solvers only take as input boolean formulas with only one theory (e.g. linear arithmetic over the integers).

Example.

$$f(f(f(x))) = x \wedge f(f(f(f(f(x))))) = x \wedge f(x) \neq x$$

$$x + y = 19 \wedge x - y = 7 \wedge x \neq 13$$

Let F be the following formula using arithmetic symbols:

$$x + y \geqslant 0 \wedge (x = z \implies y + z = -1) \wedge z > 3t$$

We want to know whether F is satisfiable. We use the following algorithm:

1. Convertion in CNF
2. Replace arithmetical constraints with boolean variables
3. Call a SAT solver: a model M is returned if the formula is satisfiable
4. Convert M to the original theory
5. Check whether M is coherent with the theory
6. If not, add $\neg M$ to F and start over

Figure 17: Simple SMT procedure using a SAT solver

SMT encounters multiple difficulties:

- The size of formulas (up to multiple Go)
- The complexity of boolean structures (e.g. when the formulas encode logical circuits of microprocessors)
- The combination of Theories
- The efficiency of decision procedures
- Theories over mathematical objects more and more complex
- The handling of quantifiers

7.2.2 Small proof engines

7.2.3 SAT and decision procedures

7.2.4 Combination of decision procedures