Semantic Equivalence
Semantics and applications to verification

Xavier Rival

École Normale Supérieure
Reasoning on program equivalence

Properties considered in the previous lectures:

- Sets of **states**: absence of runtime errors...
- Sets of **traces**: termination
- Sets of **sets or traces**: security, dependences

In all these cases, only one program is considered.

**Today: reasoning about several programs**

Kinds of questions we will consider:

- Does \( P \) and \( Q \) have the same behaviors ?
- Does \( P \) have more behaviors than \( Q \) ?

**A short introduction** to these properties and **the verification of a micro-compiler** with Coq
Informal definition: program transformation

- A **program transformation** is a **partial function**, mapping a program $P$ into another program $T(P)$.
- It should **preserve some semantic properties** of programs.

- **Compilation**: the target code behaviors should match those of the source code.

- **Optimization**: the target code may differ strongly from the source, yet “produce the same observation”.

- **Slicing**: the target code should perform the same actions over the “slicing criterion”.
A first (and naive) definition of correctness

Correctness by semantic equivalence

**Correctness** of transformation $\mathcal{T}$ writes down as an equivalence of semantics:

$$\llbracket \mathcal{T}(P) \rrbracket = \llbracket P \rrbracket$$

Why is it naïve?

- $P$ and $\mathcal{T}(P)$ may not be expressed in the same language, and thus have “comparable” semantics
- e.g., if we consider compilation, $\mathcal{T}(P)$ is much lower level (machine language, with registers, etc) than $P$
Limitations (1)

Example: compilation of a simple imperative language

- Variables
- Syntax:
  
  \[
  e ::= \nu | e + e | \ldots \\
  i ::= x := e; \\
  | \text{if}(e \leq 0) \text{ b else } \text{ b} \\
  | \text{while}(e \leq 0) \text{ b} \\
  b ::= \{i;\ldots;i;\} \\
  \]

- Variables + registers
- Program counter
- Instructions:
  
  \[
  i ::= \text{add } r_d, r_{s0}, r_{s1} | \text{li } r_d, r_{s0} v \\
  | b \text{ dst} | \text{blt } r_{s0}, r_{s1}, dst \\
  | \text{ld } [s], r_d | \text{st } [d], r_s \\
  \]

Translation of a simple code fragment

\[
\begin{align*}
\ell_0 &: x = 7; \\
\ell_1 &: y = 8 + x; \\
\ell_2 &: \ldots \\
\end{align*}
\xrightarrow{\mathcal{T}}
\begin{align*}
0 &: \text{li } r_0, 7 \\
1 &: \text{st } [x], r_0 \\
2 &: \text{li } r_0, 8 \\
3 &: \text{ld } [x], r_1 \\
4 &: \text{add } r_1, r_0, r_1 \\
5 &: \text{st } [y], r_1 \\
\end{align*}
\]
Limitations (1)

Translation of a simple code fragment:

If we attempt at comparing traces point by point:

- **intermediate assembly points** 1, 3, 4 have **no counterpart in the source**
- **registers** $r_0, r_1$ have **no counterpart in the source**

A semantic equality is **too tight**.
A second definition of correctness

Fix: apply an observation function to traces

Correctness up to observation

Correctness up to observation $\mathcal{O}$ of transformation $\mathcal{T}$ writes down as an equivalence of semantics, after applying $\mathcal{O}$ to the semantics:

$$\mathcal{O}[\mathcal{T}(P)] = [P]$$

Example:

- $\mathcal{O}$ ignores 1, 3, 4 and registers
- $\mathcal{O}$ maps 0 to $l_0$; 2 to $l_1$ and 5 to $l_2$
Limitations (2)

Floating point computations:
- source semantics: allows any IEEE-754 compliant rounding mode
- target machine semantics: may choose a specific rounding mode (e.g., before running the program)
- all target program behavior are admissible in the source
- but not all source behavior occur in the target program

Execution order:
- unspecified in the C semantics
- chosen by the compiler (different compilers may make different choices)

A semantic equality is too strong
A third definition of correctness

Fix: weaken the previous statement to an inclusion

Correctness as an inclusion

Correctness up to observation $\mathcal{O}$ of transformation $\mathcal{T}$ writes down as an inclusion of semantics, after applying $\mathcal{O}$ to the semantics:

$$\mathcal{O}[[\mathcal{T}(P)]] \subseteq [[P]]$$

In both examples, only an inclusion holds
Summary

- **Correctness** relies on **comparing executions**

- This comparison is usually **not tight**:  
  - **up-to observation** (abstraction)  
    intricate aspects of the execution of initial and transformed programs typically do not match  
  - **inclusion** (one-way only)  
    transformed programs often **refine** the initial one