Operational Semantics
Semantics and applications to verification

Xavier Rival

École Normale Supérieure

February 18, 2022

Program of this first lecture

Operational semantics

Mathematical description of the executions of a program

1. A **model** of programs: **transition systems**
   - definition, a **small step semantics**
   - a few common examples

2. **Trace semantics:** a kind of **big step** semantics
   - **finite** and **infinite** executions
   - **fixpoint**-based definitions
   - notion of **compositional semantics**

Outline

Definition

We will characterize a program by:

- **states:**
  photography of the program status at an instant of the execution
- **execution steps:** how do we move from one state to the next one

Definition: transition systems (TS)

A **transition system** is a tuple $(\mathbb{S}, \rightarrow)$ where:

- $\mathbb{S}$ is the **set of states** of the system
- $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$ is the **transition relation** of the system

Note:

- the set of states **may be infinite**

Transition systems: properties of the transition relation

A **deterministic** system is such that a state fully determines the next state

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

Otherwise, a transition system is **non deterministic**, i.e.:

$$\exists s_0, s_1, s_1' \in \mathbb{S}, \ s_0 \to s_1 \wedge s_0 \to s_1' \wedge s_1 \neq s_1'$$

Notes:

- the transition relation $\to$ defines atomic execution steps;
  it is often called **small-step semantics** or **structured operational semantics**

- steps are **discrete** (not continuous)
  to describe both discrete and continuous behaviors, we would need to look at *hybrid systems* (beyond the scope of this lecture)

Transition systems: initial and final states

Initial / final states:
we often consider transition systems with a set of initial and final states:

- a set of **initial states** $\mathbb{S}_{\mathcal{I}} \subseteq \mathbb{S}$ denotes states where the execution should start
- a set of **final states** $\mathbb{S}_{\mathcal{F}} \subseteq \mathbb{S}$ denotes states where the execution should reach the end of the program

When needed, we add these to the definition of the transition systems $(\mathbb{S}, \rightarrow, \mathbb{S}_{\mathcal{I}}, \mathbb{S}_{\mathcal{F}})$.

Blocking state (not the same as final state):

- a state $s_0 \in \mathbb{S}$ is **blocking** when it is the origin of no transition: $\forall s_1 \in \mathbb{S}, \ \neg(s_0 \rightarrow s_1)$
- example: we often introduce an **error state** (usually noted Ω to denote the erroneous, blocking configuration)

Outline

1. **Transition systems and small step semantics**
   - Definition and properties
   - Examples

2. Traces semantics

3. Summary

Finite automata as transition systems

We can formalize the **word recognition** by a finite automaton using a transition system:

- We consider **automaton** $\mathcal{A} = (Q, q_\mathrm{i}, q_\mathrm{f}, \rightarrow)$
- A **"state"** is defined by:
    - ▶ the **remaining of the word to recognize**
    - ▶ the **automaton state** that has been reached so far

  thus, $\mathbb{S} = Q \times L^*$

- The **transition relation** $\rightarrow$ of the transition system is defined by:

$$(q_0, aw) \rightarrow (q_1, w) \iff q_0 \xrightarrow{a} q_1$$

- The **initial** and **final states** are defined by:

$$\mathbb{S}_\mathcal{I} = \{(q_\mathrm{i}, w) \mid w \in L^*\} \qquad \mathbb{S}_\mathcal{F} = \{(q_\mathrm{f}, \epsilon)\}$$

Pure λ-calculus

A **bare bones model of functional programing**:

λ-terms

The set of λ-terms is defined by:

$$t, u, \ldots ::= \begin{array}{ll} x & \text{variable} \\ | \quad \lambda x \cdot t & \text{abstraction} \\ | \quad t\, u & \text{application} \end{array}$$

β-reduction

- $(\lambda x \cdot t)\, u \rightarrow_\beta t[x \leftarrow u]$
- if $u \rightarrow_\beta v$ then $\lambda x \cdot u \rightarrow_\beta \lambda x \cdot v$
- if $u \rightarrow_\beta v$ then $u\, t \rightarrow_\beta v\, t$
- if $u \rightarrow_\beta v$ then $t\, u \rightarrow_\beta t\, v$

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**; example ?
  though, ML fixes an execution order
- given a lambda term t_0, we may consider $(\mathbb{S}, \rightarrow_\beta, \mathbb{S}_\mathcal{I})$ where $\mathbb{S}_\mathcal{I} = \{t_0\}$
- **blocking states** are terms with no redex $(\lambda x \cdot u)\, v$

A MIPS like assembly language: syntax

We now consider a (very simplified) **assembly language**

- machine integers: sequences of 32-bits (set: $\mathbb{B}^{32}$)
- instructions are encoded over 32-bits (set: $\mathbb{I}_{\mathrm{MIPS}}$)
  and stored into the same space as data (i.e., $\mathbb{I}_{\mathrm{MIPS}} \subseteq \mathbb{B}^{32}$)
- we assume a fixed set of addresses $\mathbb{A}$

Memory configurations

- **Program counter pc**
  current instruction
- **General purpose registers**
  $\mathbf{r}_0 \ldots \mathbf{r}_{31}$
- **Main memory** (RAM)
  mem : $\mathbb{A} \to \mathbb{B}^{32}$
  where $\mathbb{A} \subseteq \mathbb{B}^{32}$

Instructions

$$
\begin{aligned}
i ::= \ & (\in \mathbb{I}_{\mathrm{MIPS}}) \\
| \ & \mathbf{add} \ \mathbf{r}_d, \mathbf{r}_s, \mathbf{r}_{s'} && \text{addition} \\
| \ & \mathbf{addi} \ \mathbf{r}_d, \mathbf{r}_s, v && \text{add. } v \in \mathbb{B}^{32} \\
| \ & \mathbf{sub} \ \mathbf{r}_d, \mathbf{r}_s, \mathbf{r}_{s'} && \text{subtraction} \\
| \ & \mathbf{b} \ t && \text{branch} \\
| \ & \mathbf{blt} \ \mathbf{r}_s, \mathbf{r}_{s'}, t && \text{cond. branch} \\
| \ & \mathbf{ld} \ \mathbf{r}_d, o, \mathbf{r}_x && \text{relative load} \\
| \ & \mathbf{st} \ \mathbf{r}_d, o, \mathbf{r}_x && \text{relative store} \\
\end{aligned}
$$
$v, t, o \in \mathbb{B}^{32}, \ d, s, s', x \in [0, 31]$

A MIPS like assembly language: states

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
  $\rho : \{0, \ldots, 31\} \rightarrow \mathbb{B}^{32}$
- A function mapping each **memory cell** to its value $\mu : \mathbb{A} \rightarrow \mathbb{B}^{32}$

What would a **dangerous state** be ?

- writing **over an instruction**
- reading or writing **outside the program's memory**
- we cannot fully formalize these yet...
  as we need to formalize the behavior of each instruction first

A MIPS like assembly language: transition relation

We assume a state $s = (\pi, \rho, \mu)$ and that $\mu(\pi) = i$; then:

- if $i = \mathbf{add}\ \mathbf{r}_d, \mathbf{r}_s, \mathbf{r}_{s'}$, then:
$$s \rightarrow (\pi + 4, \rho[d \leftarrow \rho(s) + \rho(s')], \mu)$$

- if $i = \mathbf{addi}\ \mathbf{r}_d, \mathbf{r}_s, v$, then:
$$s \rightarrow (\pi + 4, \rho[d \leftarrow \rho(s) + v], \mu)$$

- if $i = \mathbf{sub}\ \mathbf{r}_d, \mathbf{r}_s, \mathbf{r}_{s'}$, then:
$$s \rightarrow (\pi + 4, \rho[d \leftarrow \rho(s) - \rho(s')], \mu)$$

- if $i = \mathbf{b}\ t$, then:

$$s \rightarrow (t, \rho, \mu)$$

A MIPS like assembly language: transition relation

We assume a state $s = (\pi, \rho, \mu)$ and that $\mu(\pi) = i$; then:

- if $i = $ **blt** $r_s, r_{s'}, t$, then:
$$s \rightarrow \begin{cases} (t, \rho, \mu) & \text{if } \rho(s) < \rho(s') \\ (\pi + 4, \rho, \mu) & \text{otherwise} \end{cases}$$

- if $i = $ **ld** r_d, o, r_x, then:
$$s \rightarrow \begin{cases} (\pi + 4, \rho[d \leftarrow \mu(\rho(x) + o)], \mu) & \text{if } \rho(x) + o \in \mathbb{A} \\ \Omega & \text{otherwise} \end{cases}$$

- if $i = $ **st** r_d, o, r_x, then:
$$s \rightarrow \begin{cases} (\pi + 4, \rho, \mu[\rho(x) + o \leftarrow \rho(d)]) & \text{if } \rho(x) + o \in \mathbb{A} \\ \Omega & \text{otherwise} \end{cases}$$

A simple imperative language: syntax

We now look at a more classical **imperative language** (intuitively, a bare-bone subset of C):

- **variables** $\mathbb{X}$: finite, predefined set of variables
- **labels** $\mathbb{L}$: before and after each statement
- **values** $\mathbb{V}$: $\mathbb{V}_{\text{int}} \cup \mathbb{V}_{\text{float}} \cup \ldots$

Syntax

$$
\begin{array}{llll}
\text{e} & ::= & v \ (v \in \mathbb{V}) \mid \text{x} \ (\text{x} \in \mathbb{X}) \mid \text{e} + \text{e} \mid \text{e} * \text{e} \mid \ldots & \text{expressions} \\
\text{c} & ::= & \text{TRUE} \mid \text{FALSE} \mid \text{e} < \text{e} \mid \text{e} = \text{e} & \text{conditions} \\
\text{i} & ::= & \text{x} := \text{e}; & \text{assignment} \\
& \mid & \text{if}(\text{c}) \ \text{b else b} & \text{condition} \\
& \mid & \text{while}(\text{c}) \ \text{b} & \text{loop} \\
\text{b} & ::= & \{\text{i}; \ldots; \text{i}; \} & \text{block, program}(\mathbb{P})
\end{array}
$$

A simple imperative language: states

A **non-error state** should fully describe the configuration at one instant of the program execution, including memory and control

The **memory state** defines the current contents of the memory

$$m \in \mathbb{M} = \mathbb{X} \longrightarrow \mathbb{V}$$

The **control state** defines *where* the program currently is

- analoguous to the **program counter**
- can be defined by adding **labels** $\mathbb{L} = \{l_0, l_1, \ldots\}$ between each pair of consecutive statements; then:

$$\mathbb{S} = \mathbb{L} \times \mathbb{M} \uplus \{\Omega\}$$

- or by the **program remaining to be executed**; then:

$$\mathbb{S} = \mathbb{P} \times \mathbb{M} \uplus \{\Omega\}$$

A simple imperative language: semantics of expressions

- The **semantics** $\llbracket e \rrbracket$ **of expression** e should evaluate each expression into a value, given a memory state
- **Evaluation errors** may occur: division by zero...
  error value is also noted Ω

Thus: $\llbracket e \rrbracket : \mathbb{M} \longrightarrow \mathbb{V} \uplus \{\Omega\}$

Definition, by **induction over the syntax**:

$$
\begin{aligned}
\llbracket v \rrbracket(m) &= v \\
\llbracket \mathrm{x} \rrbracket(m) &= m(\mathrm{x}) \\
\llbracket e_0 + e_1 \rrbracket(m) &= \llbracket e_0 \rrbracket(m) \pm \llbracket e_1 \rrbracket(m) \\
\llbracket e_0 / e_1 \rrbracket(m) &= \begin{cases} \Omega & \text{if } \llbracket e_1 \rrbracket(m) = 0 \\ \llbracket e_0 \rrbracket(m) \underline{/} \llbracket e_1 \rrbracket(m) & \text{otherwise} \end{cases}
\end{aligned}
$$

where $\underline{\oplus}$ is the machine implementation of operator $\oplus$, and is Ω-strict, i.e.,
$\forall v \in \overline{\mathbb{V}}, \ v \underline{\oplus} \Omega = \Omega \underline{\oplus} v = \Omega$.

A simple imperative language: semantics of conditions

- The **semantics $\llbracket c \rrbracket$ of condition** c should return a *boolean value*
- It follows a similar definition to that of the semantics of expressions:
  $\llbracket c \rrbracket : \mathbb{M} \longrightarrow \mathbb{V}_{\text{bool}} \uplus \{\Omega\}$

Definition, by **induction over the syntax**:

$$
\begin{aligned}
\llbracket \text{TRUE} \rrbracket(m) &= \text{TRUE} \\
\llbracket \text{FALSE} \rrbracket(m) &= \text{FALSE} \\
\llbracket e_0 < e_1 \rrbracket(m) &= \begin{cases} \text{TRUE} & \text{if } \llbracket e_0 \rrbracket(m) < \llbracket e_1 \rrbracket(m) \\ \text{FALSE} & \text{if } \llbracket e_0 \rrbracket(m) \geq \llbracket e_1 \rrbracket(m) \\ \Omega & \text{if } \llbracket e_0 \rrbracket(m) = \Omega \text{ or } \llbracket e_1 \rrbracket(m) = \Omega \end{cases} \\
\llbracket e_0 = e_1 \rrbracket(m) &= \begin{cases} \text{TRUE} & \text{if } \llbracket e_0 \rrbracket(m) = \llbracket e_1 \rrbracket(m) \\ \text{FALSE} & \text{if } \llbracket e_0 \rrbracket(m) \neq \llbracket e_1 \rrbracket(m) \\ \Omega & \text{if } \llbracket e_0 \rrbracket(m) = \Omega \text{ or } \llbracket e_1 \rrbracket(m) = \Omega \end{cases}
\end{aligned}
$$

A simple imperative language: transitions

Transitions describe **local program execution steps**, thus are defined by case analysis on the program statements

Case of **assignment** $l_0 : x = e;\ l_1$

- if $[\![e]\!](m) \neq \Omega$, then $(l_0, m) \rightarrow (l_1, m[x \leftarrow [\![e]\!](m)])$
- if $[\![e]\!](m) = \Omega$, then $(l_0, m) \rightarrow \Omega$

Case of **condition** $l_0 : \text{if}(c)\{l_1 : b_t\ l_2\}\ \text{else}\{l_3 : b_f\ l_4\}\ l_5$

- if $[\![c]\!](m) = \text{TRUE}$, then $(l_0, m) \rightarrow (l_1, m)$
- if $[\![c]\!](m) = \text{FALSE}$, then $(l_0, m) \rightarrow (l_3, m)$
- if $[\![c]\!](m) = \Omega$, then $(l_0, m) \rightarrow \Omega$
- $(l_2, m) \rightarrow (l_5, m)$
- $(l_4, m) \rightarrow (l_5, m)$

A simple imperative language: transitions

Case of **loop** $l_0 : \text{while}(c)\{l_1 : b_t \ l_2\} \ l_3$

- if $[\![c]\!](m) = \text{TRUE}$, then $\begin{cases} (l_0, m) \rightarrow (l_1, m) \\ (l_2, m) \rightarrow (l_1, m) \end{cases}$

- if $[\![c]\!](m) = \text{FALSE}$, then $\begin{cases} (l_0, m) \rightarrow (l_3, m) \\ (l_2, m) \rightarrow (l_3, m) \end{cases}$

- if $[\![c]\!](m) = \Omega$, then $\begin{cases} (l_0, m) \rightarrow \Omega \\ (l_2, m) \rightarrow \Omega \end{cases}$

Case of $\{l_0 : i_0; \ l_1 : \ldots; l_{n-1} i_{n-1}; \ l_n\}$

- the transition relation is defined by the individual instructions

Extending the language with non-determinism

The language we have considered so far is a bit **limited**:

- it is **deterministic**: at most one transition possible from any state
- it does not support the **input of values**

Changes if we model non deterministic inputs...

... with an input instruction:

- i ::= ... | x := input()
- $l_0 : x := input(); l_1$ generates transitions

  $\forall v \in \mathbb{V}, \ (l_0, m) \rightarrow (l_1, m[x \leftarrow v])$

- one instruction induces non determinism

... with a random function:

- e ::= ... | rand()
- **expressions** have a **non-deterministic** semantics:

  $$[\![e]\!] : \mathbb{M} \rightarrow \mathcal{P}(\mathbb{V} \uplus \{\Omega\})$$
  $$[\![rand()]\!](m) = \mathbb{V}$$
  $$[\![v]\!](m) = \{v\}$$
  $$[\![c]\!] : \mathbb{M} \rightarrow \mathcal{P}(\mathbb{V}_{bool} \uplus \{\Omega\})$$

- all instructions induce non determinism

Semantics of real world programming languages

C language:

- several **norms**: ANSI C'99, ANSI C'11, K&R...
- not fully specified:
  - ► **undefined behavior**
  - ► **implementation dependent behavior**: architecture (ABI) or implementation (compiler...)
  - ► unspecified parts: leave room for implementation of compilers and optimizations
- **formalizations** in HOL (C'99), in Coq (CompCert C compiler)

OCaml language:

- more formal...
- ... but still with some unspecified parts, e.g., execution order

Outline

Execution traces

- So far, we considered only states and atomic transitions
- We now consider program **executions** as a whole

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:

- $\mathbb{S}^*$ for the **set of finite traces**
- $\mathbb{S}^\omega$ for the **set of infinite traces**
- $\mathbb{S}^\infty = \mathbb{S}^* \cup \mathbb{S}^\omega$ for the **set of finite or infinite traces**

Operations on traces: concatenation

Definition: concatenation

The **concatenation operator** $\cdot$ is defined by:

$$\begin{array}{rcl}
\langle s_0, \ldots, s_n \rangle \cdot \langle s'_0, \ldots, s'_{n'} \rangle &=& \langle s_0, \ldots, s_n, s'_0, \ldots, s'_{n'} \rangle \\
\langle s_0, \ldots, s_n \rangle \cdot \langle s'_0, \ldots \rangle &=& \langle s_0, \ldots, s_n, s'_0, \ldots \rangle \\
\langle s_0, \ldots, s_n, \ldots \rangle \cdot \sigma' &=& \langle s_0, \ldots, s_n, \ldots \rangle
\end{array}$$

We also define:

- the **empty trace** ϵ, neutral element for $\cdot$
- the **length** operator $|.|$:

$$\left\{ \begin{array}{rcl}
|\epsilon| &=& 0 \\
|\langle s_0, \ldots, s_n \rangle| &=& n + 1 \\
|\langle s_0, \ldots \rangle| &=& \omega
\end{array} \right.$$

Comparing traces: the prefix order relation

Definition: prefix order relation

Relation $\prec$ is defined by:

$$\langle s_0, \ldots, s_n \rangle \prec \langle s'_0, \ldots, s'_{n'} \rangle \iff \left\{ \begin{array}{l} n \leq n' \\ \forall i \in [\![0, n]\!], \ s_i = s'_i \end{array} \right.$$

$$\langle s_0, \ldots \rangle \prec \langle s'_0, \ldots \rangle \iff \forall i \in \mathbb{N}, \ s_i = s'_i$$

$$\langle s_0, \ldots, s_n \rangle \prec \langle s'_0, \ldots \rangle \iff \forall i \in [\![0, n]\!], \ s_i = s'_i$$

Proof: straightforward application of the definition of order relations

Outline

Semantics of finite traces

We consider a transition system $\mathcal{S} = (\mathbb{S}, \rightarrow)$

Definition

The **finite traces semantics** $[\![\mathcal{S}]\!]^*$ is defined by:

$$[\![\mathcal{S}]\!]^* = \{\langle s_0, \ldots, s_n \rangle \in \mathbb{S}^* \mid \forall i, \, s_i \rightarrow s_{i+1}\}$$

Example:

- contrived transition system $\mathcal{S} = (\{a, b, c, d\}, \{(a, b), (b, a), (b, c)\})$
- finite traces semantics:

$$
\begin{aligned}
[\![\mathcal{S}]\!]^* = \{ \quad &\epsilon, \\
&\langle a, b, \ldots, a, b, a \rangle, \quad \langle b, a, \ldots, a, b, a \rangle, \\
&\langle a, b, \ldots, a, b, a, b \rangle, \quad \langle b, a, \ldots, a, b, a, b \rangle, \\
&\langle a, b, \ldots, a, b, a, b, c \rangle, \quad \langle b, a, \ldots, a, b, a, b, c \rangle \\
&\langle c \rangle, \quad\quad\quad\quad\quad\quad\quad \langle d \rangle \quad\quad\quad\quad\quad \}
\end{aligned}
$$

Interesting subsets of the finite trace semantics

We consider a transition system $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_{\mathcal{I}}, \mathbb{S}_{\mathcal{F}})$

- the **initial traces**, i.e., starting from an initial state:

$$\{\langle s_0, \ldots, s_n \rangle \in [\![\mathcal{S}]\!]^* \mid s_0 \in \mathbb{S}_{\mathcal{I}}\}$$

- the **traces reaching a blocking state**:

$$\{\sigma \in [\![\mathcal{S}]\!]^* \mid \forall \sigma' \in [\![\mathcal{S}]\!]^*, \sigma \prec \sigma' \Longrightarrow \sigma = \sigma'\}$$

- the **traces ending in a final state**:

$$\{\langle s_0, \ldots, s_n \rangle \in [\![\mathcal{S}]\!]^* \mid s_n \in \mathbb{S}_{\mathcal{F}}\}$$

- the **maximal traces** are both initial and final

Example (same transition system, with $\mathbb{S}_{\mathcal{I}} = \{a\}$ and $\mathbb{S}_{\mathcal{F}} = \{c\}$):

- traces from an initial state ending in a final state are all of the form:
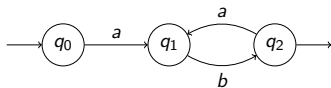 $\langle a, b, \ldots, a, b, a, b, c \rangle$

Example: finite automaton

We consider the **example of the previous lecture:**

$$L = \{a, b\} \qquad Q = \{q_0, q_1, q_2\}$$
$$q_i = q_0 \qquad q_f = q_2$$
$$q_0 \xrightarrow{a} q_1 \qquad q_1 \xrightarrow{b} q_2 \qquad q_2 \xrightarrow{a} q_1$$

Then, we have the following traces:

$$
\begin{aligned}
\tau_0 &= \langle (q_0, ab), (q_1, b), (q_2, \epsilon) \rangle \\
\tau_1 &= \langle (q_0, abab), (q_1, bab), (q_2, ab), (q_1, b), (q_2, \epsilon) \rangle \\
\tau_2 &= \langle (q_0, ababab), (q_1, babab), (q_2, abab), (q_1, bab) \rangle \\
\tau_3 &= \langle (q_0, abaaa), (q_1, baaa), (q_2, aaa), (q_1, aa) \rangle
\end{aligned}
$$

Then:

- τ_0, τ_1 are initial traces, reaching a final state
- τ_2 is an initial trace, and is not maximal
- τ_3 reaches a blocking state, but not a final state

Example: λ-term

We consider λ-term $\lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x\, x)\, (\lambda x \cdot x\, x)))$, and show two traces generated from it (at each step the reduced lambda is shown in red):

$$
\begin{aligned}
\tau_0 \;=\; \langle \quad & \lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x\, x)\, (\lambda x \cdot x\, x))) \\
& \lambda y \cdot y \quad \rangle
\end{aligned}
$$

$$
\begin{aligned}
\tau_1 \;=\; \langle \quad & \lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x\, x)\, (\lambda x \cdot x\, x))), \\
& \lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x\, x)\, (\lambda x \cdot x\, x))), \\
& \lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x\, x)\, (\lambda x \cdot x\, x))) \quad \rangle
\end{aligned}
$$

Then:

- τ_0 is a maximal trace; it reaches a blocking state (no more reduction can be done)
- τ_1 can be extended for arbitrarily many steps ;
  the second part of the course will study **infinite traces**

Example: imperative program

Similarly, we can write the traces of a simple imperative program:

$$
\begin{aligned}
l_0 : \quad & x := 1; \\
l_1 : \quad & y := 0; \\
l_2 : \quad & \text{while}(x < 4)\{ \\
l_3 : \quad & \quad y := y + x; \\
l_4 : \quad & \quad x := x + 1; \\
l_5 : \quad & \} \\
l_6 : \quad & \text{(final program point)}
\end{aligned}
$$

$$
\begin{aligned}
\tau = \langle \ & (l_0, (\!| x = 6, y = 8 |\!)), (l_1, (\!| x = 1, y = 8 |\!)), \\
& (l_2, (\!| x = 1, y = 0 |\!)), (l_3, (\!| x = 1, y = 0 |\!)), \\
& (l_4, (\!| x = 1, y = 1 |\!)), (l_5, (\!| x = 2, y = 1 |\!)), \\
& (l_3, (\!| x = 2, y = 1 |\!)), (l_4, (\!| x = 2, y = 3 |\!)), \\
& (l_5, (\!| x = 3, y = 3 |\!)), (l_3, (\!| x = 3, y = 3 |\!)), \\
& (l_4, (\!| x = 3, y = 6 |\!)), (l_5, (\!| x = 4, y = 6 |\!)), \\
& (l_6, (\!| x = 4, y = 6 |\!)) \ \rangle
\end{aligned}
$$

- very **precise** description of what the program does...
- ... but **quite cumbersome**

Outline

1. Transition systems and small step semantics

2. Traces semantics
   - Definitions
   - Finite traces semantics
   - Fixpoint definition
   - Compositionality
   - Infinite traces semantics

3. Summary

Towards a fixpoint definition

We consider again our contrived transition system

$$\mathcal{S} = (\{a, b, c, d\}, \{(a, b), (b, a), (b, c)\})$$

Traces **by length:**

i	traces of length i
0	ϵ
1	$\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle$
2	$\langle a, b \rangle, \langle b, a \rangle, \langle b, c \rangle$
3	$\langle a, b, a \rangle, \langle b, a, b \rangle, \langle a, b, c \rangle$
4	$\langle a, b, a, b \rangle, \langle b, a, b, a \rangle, \langle b, a, b, c \rangle$

Like the automaton in lecture 1, this suggests a least fixpoint definition: traces of length $i + 1$ can be derived from the traces of length i, by adding a transition

Trace semantics fixpoint form

We define a **semantic function**, that computes **the traces of length $i+1$ from the traces of length i** (where $i \geq 1$), and **adds the traces of length** 1:

Finite traces semantics as a fixpoint

Let $\mathcal{I} = \{\epsilon\} \cup \{\langle s \rangle \mid s \in \mathbb{S}\}$. Let F_* be the function defined by:

$$F_* : \quad \mathcal{P}(\mathbb{S}^*) \quad \longrightarrow \quad \mathcal{P}(\mathbb{S}^*)$$
$$X \quad \longmapsto \quad \mathcal{I} \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle \mid \langle s_0, \ldots, s_n \rangle \in X \wedge s_n \to s_{n+1}\}$$

Then, F_* is **continuous** and thus has a least-fixpoint and:

$$\text{lfp}\, F_* = \llbracket \mathcal{S} \rrbracket^* = \bigcup_{n \in \mathbb{N}} F_*^n(\emptyset)$$

Fixpoint definition: proof (1), fixpoint existence

First, we prove that F_* is **continuous**.

Let $\mathcal{X} \subseteq \mathcal{P}(\mathbb{S}^*)$ such that $\mathcal{X} \neq \emptyset$ and $A = \bigcup_{U \in \mathcal{X}} U$. Then:

$\quad F_*(\bigcup_{X \in \mathcal{X}} X)$
$\quad = \ \mathcal{I} \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid (\langle s_0, \ldots, s_n\rangle \in \bigcup_{U \in \mathcal{X}} U) \wedge s_n \rightarrow s_{n+1}\}$
$\quad = \ \mathcal{I} \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \exists U \in \mathcal{X}, \ \langle s_0, \ldots, s_n\rangle \in U \wedge s_n \rightarrow s_{n+1}\}$
$\quad = \ \mathcal{I} \cup \left(\bigcup_{U \in \mathcal{X}} \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \langle s_0, \ldots, s_n\rangle \in U \wedge s_n \rightarrow s_{n+1}\}\right)$
$\quad = \ \bigcup_{U \in \mathcal{X}} \left(\mathcal{I} \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \langle s_0, \ldots, s_n\rangle \in U \wedge s_n \rightarrow s_{n+1}\}\right)$
$\quad = \ \bigcup_{U \in \mathcal{X}} F_*(U)$

In particular, this is true for any increasing chain $\mathcal{X}$ (here, we considered any non empty family), hence F_* is continuous.

As $(\mathcal{P}(\mathbb{S}^*), \subseteq)$ is a CPO, the continuity of F_* entails the **existence of a least-fixpoint** (Kleene theorem); moreover, it implies that:

$$\mathsf{lfp}\, F_* = \bigcup_{n \in \mathbb{N}} F_*^n(\emptyset)$$

Fixpoint definition: proof (2), fixpoint equality

We now show that $[\![\mathcal{S}]\!]^*$ is equal to $\mathrm{lfp}\, F_*$, by showing the property below, by induction over n:

$$\forall n \geq 1, \; \forall k < n, \; \langle s_0, \ldots, s_k \rangle \in F_*^n(\emptyset) \iff \langle s_0, \ldots, s_k \rangle \in [\![\mathcal{S}]\!]^*$$

- at rank 1, only trace ϵ and the traces of length 1 need to be considered, and this case is trivial
- at rank $n + 1$, we need to consider both traces of length 1 (the case of which is trivial) and traces of length $n + 1$ for some integer $n \geq 1$:

$$
\begin{aligned}
&\langle s_0, \ldots, s_k, s_{k+1} \rangle \in [\![\mathcal{S}]\!]^* \\
&\iff \langle s_0, \ldots, s_k \rangle \in [\![\mathcal{S}]\!]^* \land s_k \to s_{k+1} \\
&\iff \langle s_0, \ldots, s_k \rangle \in F_*^n(\emptyset) \land s_k \to s_{k+1} \quad (k < n \text{ since } k + 1 < n + 1) \\
&\iff \langle s_0, \ldots, s_k, s_{k+1} \rangle \in F_*^{n+1}(\emptyset)
\end{aligned}
$$

Trace semantics fixpoint form: example

Example, with the same simple transition system $\mathcal{S} = (\mathbb{S}, \rightarrow)$:

- $\mathbb{S} = \{a, b, c, d\}$
- $\rightarrow$ is defined by $a \rightarrow b$, $b \rightarrow a$ and $b \rightarrow c$

Then, the first iterates are:

$$
\begin{array}{rcl}
F_*^0(\emptyset) &=& \emptyset \\
F_*^1(\emptyset) &=& \{\epsilon, \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle\} \\
F_*^2(\emptyset) &=& F_*^1(\emptyset) \cup \{\langle b, a \rangle, \langle a, b \rangle, \langle b, c \rangle\} \\
F_*^3(\emptyset) &=& F_*^2(\emptyset) \cup \{\langle a, b, a \rangle, \langle b, a, b \rangle, \langle a, b, c \rangle\} \\
F_*^4(\emptyset) &=& F_*^3(\emptyset) \cup \{\langle b, a, b, a \rangle, \langle a, b, a, b \rangle, \langle b, a, b, c \rangle\} \\
F_*^5(\emptyset) &=& F_*^4(\emptyset) \cup \{\langle a, b, a, b, a \rangle, \langle b, a, b, a, b \rangle, \langle a, b, a, b, c \rangle\} \\
F_*^6(\emptyset) &=& \ldots
\end{array}
$$

The traces of $[\![\mathcal{S}]\!]^*$ of length $n + 1$ appear in $F_*^n(\emptyset)$

Outline

Notion of compositional semantics

The traces semantics definition we have seen is **global:**

- the **whole system** defines a **transition relation**
- we **iterate** this relation until we get a fixpoint

Though, a **modular** definition would be nicer, to allow reasoning on program fragments, or derive properties of a program from properties of its pieces...

Can we derive a more modular expression of the semantics ?

Notion of compositional semantics

Observation: programs often have an inductive structure

- λ-**terms** are defined by induction over the syntax
- **imperative programs** are defined by induction over the syntax
- **there are exceptions:** our MIPS language does not naturally look that way

Definition: compositional semantics

A semantics $[\![.]\!]$ is said to be **compositional** when the semantics of a program can be defined as a function of the semantics of its parts, i.e.,

When program π writes down as $C[\pi_0, \ldots, \pi_k]$ where $\pi_0, \ldots, \pi_k$ are its components, there exists a function F_C such that $[\![\pi]\!] = F_C([\![\pi_0]\!], \ldots, [\![\pi_k]\!])$, where F_C depends only on syntactic construction F_C.

Case of a simplified imperative language

Case of **a sequence of two instructions** $b \equiv l_0 : i_0; l_1 : i_1; l_2$:

$$
\begin{aligned}
\llbracket b \rrbracket^* = \quad & \llbracket i_0 \rrbracket^* \cup \llbracket i_1 \rrbracket^* \\
\cup \quad & \{ \langle s_0, \ldots, s_m \rangle \mid \exists n \in \llbracket 0, m \rrbracket, \\
& \quad \langle s_0, \ldots, s_n \rangle \in \llbracket i_0 \rrbracket^* \wedge \langle s_n, \ldots, s_m \rangle \in \llbracket i_1 \rrbracket^* \}
\end{aligned}
$$

This amounts to **concatenating** traces of $\llbracket i_0 \rrbracket^*$ and $\llbracket i_1 \rrbracket^*$ that share a state in common (necessarily at point l_1).

Cases of **a condition, a loop: similar**

- by **concatenation** of traces around **junction points**
- by doing a **least-fixpoint computation** over loops

We can provide a compositional semantics for our simplified imperative language

Case of λ-calculus

Case of a λ-**term** $t = (\lambda x \cdot u) \, v$:

- executions may start with a reduction in u
- executions may start with a reduction in v
- executions may start with the reduction of the head redex
- an execution may mix reductions steps in u and v in an arbitrary order

No nice compositional trace semantics of λ-calculus...

Outline

Non termination

Can the finite traces semantics express non termination ?

Consider the case of our contrived system:

$$\mathbb{S} = \{a, b, c, d\} \qquad (\rightarrow) = \{(a, b), (b, a), (b, c)\}$$

System behaviors:

- this system clearly **has non-terminating behaviors**:
  it can loop from a to b and back forever
- the finite traces semantics does show **the existence of this cycle** as
  there exists an **infinite chain of finite traces for the prefix order** $\prec$:
  $$\langle a, b \rangle, \langle a, b, a \rangle, \langle a, b, a, b \rangle, \langle a, b, a, b, a \rangle, \ldots \in [\![\mathcal{S}]\!]^*$$
- though, the existence of this chain is **not very obvious**

Thus, we now define a semantics made of infinite traces

Semantics of infinite traces

We consider a transition system $\mathcal{S} = (\mathbb{S}, \rightarrow)$

Definition

The **infinite traces semantics** $[\![\mathcal{S}]\!]^\omega$ is defined by:

$$[\![\mathcal{S}]\!]^\omega = \{\langle s_0, \ldots \rangle \in \mathbb{S}^\omega \mid \forall i, \ s_i \rightarrow s_{i+1}\}$$

Infinite traces starting from an initial state (considering $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_\mathcal{I}, \mathbb{S}_\mathcal{F})$):

$$\{\langle s_0, \ldots \rangle \in [\![\mathcal{S}]\!]^\omega \mid s_0 \in \mathbb{S}_\mathcal{I}\}$$

Example:

- contrived transition system defined by

$$\mathbb{S} = \{a, b, c, d\} \qquad (\rightarrow) = \{(a, b), (b, a), (b, c)\}$$

- the infinite traces semantics contains **exactly two** traces

$$[\![\mathcal{S}]\!]^\omega = \{\langle a, b, \ldots, a, b, a, b, \ldots \rangle, \langle b, a, \ldots, b, a, b, a, \ldots \rangle\}$$

Fixpoint form

Can we also provide a fixpoint form for $[\![\mathcal{S}]\!]^{\omega}$?

Intuitively, $\langle s_0, s_1, \ldots \rangle \in [\![\mathcal{S}]\!]^{\omega}$ if and only if $\forall n, \ s_n \rightarrow s_{n+1}$, i.e.,

$$\forall n \in \mathbb{N}, \ \forall k \leq n, \ s_k \rightarrow s_{k+1}$$

Let F_{ω} be defined by:

$$
\begin{array}{rrcl}
F_{\omega} : & \mathcal{P}(\mathbb{S}^{\omega}) & \longrightarrow & \mathcal{P}(\mathbb{S}^{\omega}) \\
& X & \longmapsto & \{ \langle s_0, s_1, \ldots, s_n, \ldots \rangle \mid \langle s_1, \ldots, s_n, \ldots \rangle \in X \wedge s_0 \rightarrow s_1 \}
\end{array}
$$

Then, we can show by induction that:

$$
\begin{array}{rcl}
\sigma \in [\![\mathcal{S}]\!]^{\omega} & \Longleftrightarrow & \forall n \in \mathbb{N}, \ \sigma \in F_{\omega}^{n}(\mathbb{S}^{\omega}) \\
& \Longleftrightarrow & \sigma \in \bigcap_{n \in \mathbb{N}} F_{\omega}^{n}(\mathbb{S}^{\omega})
\end{array}
$$

Fixpoint form of the semantics of infinite traces

Infinite traces semantics as a fixpoint

Let F_ω be the function defined by:

$$
\begin{array}{rrcl}
F_\omega : & \mathcal{P}(\mathbb{S}^\omega) & \longrightarrow & \mathcal{P}(\mathbb{S}^\omega) \\
& X & \longmapsto & \{\langle s_0, s_1, \ldots, s_n, \ldots \rangle \mid \langle s_1, \ldots, s_n, \ldots \rangle \in X \wedge s_0 \rightarrow s_1\}
\end{array}
$$

Then, F_ω is $\cap$-**continuous** and thus has a **greatest-fixpoint**; moreover:

$$
\mathbf{gfp}\, F_\omega = \llbracket \mathcal{S} \rrbracket^\omega = \bigcap_{n \in \mathbb{N}} F_\omega^n(\mathbb{S}^\omega)
$$

Proof sketch:

- the $\cap$-contiunity proof is similar as for the $\cup$-continuity of F_*
- by the dual version of Kleene's theorem, $\mathbf{gfp}\, F_\omega$ exists and is equal to $\bigcap_{n \in \mathbb{N}} F_\omega^n(\mathbb{S}^\omega)$, i.e. to $\llbracket \mathcal{S} \rrbracket^\omega$ (similar induction proof)

Fixpoint form of the infinite traces semantics: iterates

Example, with the same simple transition system:

- $\mathbb{S} = \{a, b, c, d\}$
- $\rightarrow$ is defined by $a \rightarrow b$, $b \rightarrow a$ and $b \rightarrow c$

Then, the first iterates are:

$$
\begin{array}{rcl}
F_\omega^0(\mathbb{S}^\omega) & = & \mathbb{S}^\omega \\
F_\omega^1(\mathbb{S}^\omega) & = & \langle a, b \rangle \cdot \mathbb{S}^\omega \cup \langle b, a \rangle \cdot \mathbb{S}^\omega \cup \langle b, c \rangle \cdot \mathbb{S}^\omega \\
F_\omega^2(\mathbb{S}^\omega) & = & \langle b, a, b \rangle \cdot \mathbb{S}^\omega \cup \langle a, b, a \rangle \cdot \mathbb{S}^\omega \cup \langle a, b, c \rangle \cdot \mathbb{S}^\omega \\
F_\omega^3(\mathbb{S}^\omega) & = & \langle a, b, a, b \rangle \cdot \mathbb{S}^\omega \cup \langle b, a, b, a \rangle \cdot \mathbb{S}^\omega \cup \langle b, a, b, c \rangle \cdot \mathbb{S}^\omega \\
F_\omega^4(\mathbb{S}^\omega) & = & \ldots
\end{array}
$$

Intuition

- at iterate n, prefixes of length $n + 1$ match the traces in the infinite semantics

- only $\langle a, b, \ldots, a, b, a, b, \ldots \rangle$ and $\langle b, a, \ldots, b, a, b, a, \ldots \rangle$ belong to *all* iterates

Outline

1. Transition systems and small step semantics

2. Traces semantics

3. Summary

Summary

We have discussed today:

- **small-step / structural operational semantics:**
  individual program steps

- **big-step / natural semantics:**
  program executions as sequences of transitions

- their **fixpoint definitions** and properties
  will play a great role to design verification techniques

Next lectures:

- another family of semantics, **more compact** and **compositional**
- **semantic program** and **proof methods**