Operational Semantics
Semantics and applications to verification

Xavier Rival
École Normale Supérieure

February 19, 2016
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

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

2. Traces semantics

3. Summary
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 $(S, \rightarrow)$ where:

- $S$ is the **set of states** of the system
- $\rightarrow \subseteq P(S \times S)$ 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

\[ \forall s_0, s_1, s'_1 \in S, (s_0 \rightarrow s_1 \land s_0 \rightarrow s'_1) \implies s_1 = s'_1 \]

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

\[ \exists s_0, s_1, s'_1 \in S, s_0 \rightarrow s_1 \land s_0 \rightarrow s'_1 \land s_1 \neq s'_1 \]

Notes:

- the transition relation \( \rightarrow \) 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 $S_I \subseteq S$ denotes states where the execution should start
- a set of final states $S_F \subseteq S$ denotes states where the execution should reach the end of the program

When needed, we add these to the definition of the transition systems $((S, \rightarrow, S_I, S_F))$.

Blocking state (not the same as final state):

- a state $s_0 \in S$ is blocking when it is the origin of no transition:
  $\forall s_1 \in S, \neg (s_0 \rightarrow s_1)$
- example: we often introduce an error state (usually noted $\Omega$ 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 clearly formalize the word recognition by a finite automaton using a transition system:

- We consider automaton $\mathcal{A} = (Q, q_i, q_f, \rightarrow)$
- A “state” is defined by:
  - the remaining of the word to recognize
  - the automaton state that has been reached so far
  thus, $S = Q \times L^*$
- 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:
  $$S_I = \{(q_i, w) \mid w \in L^*\} \quad S_F = \{(q_f, \epsilon)\}$$
Pure $\lambda$-calculus

A bare bones model of functional programming:

$\lambda$-terms

The set of $\lambda$-terms is defined by:

$$t, u, \ldots \ ::= \ x \quad \text{variable}$$

$$| \quad \lambda x \cdot t \quad \text{abstraction}$$

$$| \quad t \ u \quad \text{application}$$

$\beta$-reduction

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

The $\lambda$-calculus defines a transition system:

- $S$ is the set of $\lambda$-terms and $\to_\beta$ the transition relation
- $\to_\beta$ is non-deterministic; example?
  though, ML fixes an execution order
- given a lambda term $t_0$, we may consider $(S, \to_\beta, S_I)$ where $S_I = \{ t_0 \}$
- 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: $I_{MIPS}$)
  and stored into the same space as data (i.e., $I_{MIPS} \subseteq \mathbb{B}^{32}$)
- we assume a fixed set of addresses $A$

Memory configurations

- Program counter $pc$
  - current instruction
- General purpose registers $r_0 \ldots r_{31}$
- Main memory (RAM)
  - mem : $A \rightarrow \mathbb{B}^{32}$
  - where $A \subseteq \mathbb{B}^{32}$

Instructions

\[
i ::= \begin{cases}
(\in I_{MIPS}) \\
\text{add } r_d, r_s, r_{s'} & \text{addition}
\mid \text{addi } r_d, r_s, v & \text{add. } v \in \mathbb{B}^{32}
\mid \text{sub } r_d, r_s, r_{s'} & \text{subtraction}
\mid b t & \text{branch}
\mid \text{blt } r_s, r_{s'}, t & \text{cond. branch}
\mid \text{ld } r_d, o, r_x & \text{relative load}
\mid \text{st } r_d, o, r_x & \text{relative store}
\end{cases}
\]

\[v, t, o \in \mathbb{B}^{32}, d, s, s', x \in [0, 31]\]
A MIPS like assembly language: states

Definition: state

A state is a tuple \((\pi, \rho, \mu)\) which comprises:

- A program counter value \(\pi \in \mathbb{B}^{32}\)
- A function mapping each general purpose register to its value \(\rho : \{0, \ldots, 31\} \rightarrow \mathbb{B}^{32}\)
- A function mapping each memory cell to its value \(\mu : A \rightarrow \mathbb{B}^{32}\)

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 \( i = \text{add} \ r_d, r_s, r'_s \), then:
  \[
  s \rightarrow (\pi + 4, \rho[d \leftarrow \rho(s) + \rho(s')], \mu)
  \]

- if \( i = \text{addi} \ r_d, r_s, v \), then:
  \[
  s \rightarrow (\pi + 4, \rho[d \leftarrow \rho(s) + v], \mu)
  \]

- if \( i = \text{sub} \ r_d, r_s, r'_s \), then:
  \[
  s \rightarrow (\pi + 4, \rho[d \leftarrow \rho(s) - \rho(s')], \mu)
  \]

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

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

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

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

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

#### Syntax

- **expressions**
  
  $e ::= v \ (v \in V) \ | \ x \ (x \in X) \ | \ e + e \ | \ e \ast e \ | \ldots$

- **conditions**
  
  $c ::= \text{TRUE} \ | \ \text{FALSE} \ | \ e < e \ | \ e = e$

- **assignment**
  
  $i ::= x := e$

- **condition**
  
  $i \ ::= \text{if}(c) \ b \ \text{else} \ b$

- **loop**
  
  $i \ ::= \text{while}(c) \ b$

- **block, program**
  
  $b ::= \{i; \ldots; i;\}$
A simple imperative language: states

A **non-error state** should fully describe the configuration at one instant of the program execution.

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

- analogous to the **program counter**
- can be defined by adding **labels** $\mathbb{L} = \{\ell_0, \ell_1, \ldots\}$ between each pair of consecutive statements; then:
  \[
  S = \mathbb{L} \times M \cup \{\Omega\}
  \]
- or by the **program remaining to be executed**; then:
  \[
  S = P \times M \cup \{\Omega\}
  \]

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

\[
 m \in M = X \longrightarrow V
\]
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 $\Omega$.

Thus: $\llbracket e \rrbracket : M \rightarrow V \cup \{\Omega\}$

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

\[
\begin{align*}
\llbracket v \rrbracket(m) & = v \\
\llbracket x \rrbracket(m) & = m(x) \\
\llbracket e_0 + e_1 \rrbracket(m) & = \llbracket e_0 \rrbracket(m) \oplus \llbracket e_1 \rrbracket(m) \\
\llbracket e_0 / e_1 \rrbracket(m) & = \begin{cases} 
\Omega & \text{if } \llbracket e_1 \rrbracket(m) = 0 \\
\llbracket e_0 \rrbracket(m) / \llbracket e_1 \rrbracket(m) & \text{otherwise}
\end{cases}
\end{align*}
\]

Where $\oplus$ is the machine implementation of operator $\oplus$, and is $\Omega$-strict, i.e., $\forall v \in V, v \oplus \Omega = \Omega \oplus v = \Omega$. 
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:

$$\llbracket c \rrbracket : M \rightarrow \mathbb{V}_\text{bool} \cup \{\Omega\}$$

Definition, by induction over the syntax:

$$\llbracket \text{TRUE} \rrbracket (m) = \text{TRUE}$$
$$\llbracket \text{FALSE} \rrbracket (m) = \text{FALSE}$$

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

$$\llbracket e_0 = e_1 \rrbracket (m) = \begin{cases} 
\text{TRUE} & \text{if } \llbracket e_0 \rrbracket (m) = \llbracket e_1 \rrbracket (m) \\
\text{FALSE} & \text{if } \llbracket e_0 \rrbracket (m) \neq \llbracket e_1 \rrbracket (m) \\
\Omega & \text{if } \llbracket e_0 \rrbracket (m) = \Omega \text{ or } \llbracket e_1 \rrbracket (m) = \Omega 
\end{cases}$$
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 $\llbracket e \rrbracket(m) \neq \Omega$, then $(l_0, m) \rightarrow (l_1, m[x \leftarrow \llbracket e \rrbracket(m))]$
- if $\llbracket e \rrbracket(m) = \Omega$, 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 $\llbracket c \rrbracket(m) = \text{TRUE}$, then $(l_0, m) \rightarrow (l_1, m)$
- if $\llbracket c \rrbracket(m) = \text{FALSE}$, then $(l_0, m) \rightarrow (l_3, m)$
- if $\llbracket c \rrbracket(m) = \Omega$, 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 \( \lbrack c \rbrack(m) = \text{TRUE}, \text{ then} \) \( (l_0, m) \rightarrow (l_1, m) \)
- if \( \lbrack c \rbrack(m) = \text{FALSE}, \text{ then} \) \( (l_0, m) \rightarrow (l_3, m) \)
- if \( \lbrack c \rbrack(m) = \Omega, \text{ then} \) \( (l_0, m) \rightarrow \Omega \)

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:

- \( i ::= \ldots \mid x := \text{input}() \)
- \( \ell_0 : x := \text{input}() ; \ell_1 \) generates transitions
  \[ \forall v \in \mathbb{V}, (\ell_0, m) \rightarrow (\ell_1, m[x \leftarrow v]) \]
- one instruction induces non determinism

... with a random function:

- \( e ::= \ldots \mid \text{rand}() \)
- expressions have a non-deterministic semantics:
  \[ [e] : \mathcal{M} \rightarrow \mathcal{P} (\mathbb{V} \cup \{\Omega\}) \]
  \[ [\text{rand}()] (m) = \mathbb{V} \]
  \[ [v] (m) = \{v\} \]
  \[ [c] : \mathcal{M} \rightarrow \mathcal{P} (\mathbb{V}_{\text{bool}} \cup \{\Omega\}) \]
- 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

1 Transition systems and small step semantics

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

3 Summary
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:

- $S^*$ for the set of finite traces
- $S^\omega$ for the set of infinite traces
- $S^\alpha = S^* \cup S^\omega$ for the set of finite or infinite traces
Operations on traces: concatenation

Definition: concatenation

The concatenation operator $\cdot$ is defined by:

\[
\langle s_0, \ldots, s_n \rangle \cdot \langle s'_0, \ldots, s'_n \rangle = \langle s_0, \ldots, s_n, s'_0, \ldots, s'_n \rangle
\]
\[
\langle s_0, \ldots, s_n \rangle \cdot \langle s'_0, \ldots \rangle = \langle s_0, \ldots, s_n, s'_0, \ldots \rangle
\]
\[
\langle s_0, \ldots, s_n, \ldots \rangle \cdot \sigma' = \langle s_0, \ldots, s_n, \ldots \rangle
\]

We also define:

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

\[
\begin{align*}
|\epsilon| & = 0 \\
|\langle s_0, \ldots, s_n \rangle| & = n + 1 \\
|\langle s_0, \ldots \rangle| & = \omega
\end{align*}
\]
Comparing traces: the prefix order relation

**Definition: prefix order relation**

Relation $\prec$ is defined by:

\[
\langle s_0, \ldots, s_n \rangle \prec \langle s'_0, \ldots, s'_{n'} \rangle \iff \begin{cases} 
    n \leq n' \\
    \forall i \in [0, n], s_i = s'_i
\end{cases}
\]

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

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

**Proof:** straightforward application of the definition of order relations
Outline

1 Transition systems and small step semantics

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

3 Summary
**Semantics of finite traces**

We consider a transition system $S = (S, \rightarrow)$

**Definition**

The **finite traces semantics** $[S]^*$ is defined by:

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

**Example:**

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

$$[S]^* = \{ \langle a, b, \ldots, a, b, a \rangle, \langle b, a, \ldots, a, b, a \rangle, \langle a, b, \ldots, a, b, a, b \rangle, \langle b, a, \ldots, a, b, a, b \rangle, \langle a, b, \ldots, a, b, a, b, c \rangle, \langle b, a, \ldots, a, b, a, b, c \rangle, \langle c \rangle, \langle d \rangle \}$$
Interesting subsets of the finite trace semantics

We consider a transition system $S = (S, \rightarrow, S_I, S_F)$

- **the initial traces**, i.e., starting from an initial state:
  \[ \{ \langle s_0, \ldots, s_n \rangle \in [S]^* \mid s_0 \in S_I \} \]

- **the traces reaching a blocking state**:
  \[ \{ \sigma \in [S]^* \mid \forall \sigma' \in [S]^*, \sigma \prec \sigma' \implies \sigma = \sigma' \} \]

- **the traces ending in a final state**:
  \[ \{ \langle s_0, \ldots, s_n \rangle \in [S]^* \mid s_n \in S_F \} \]

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

**Example** (same transition system, with $S_I = \{a\}$ and $S_F = \{c\}$):

- traces from an initial state ending in a final state:
  \[ \{ \langle a, b, \ldots, a, b, a, b, c \rangle \} \]
Example: finite automaton

We consider the example of the previous course:

\[ L = \{a, b\} \quad Q = \{q_0, q_1, q_2\} \]

\[ q_i = q_0 \quad q_f = q_2 \]

\[ q_0 \xrightarrow{a} q_1 \quad q_1 \xrightarrow{b} q_2 \quad q_2 \xrightarrow{a} q_1 \]

Then, we have the following traces:

\[ \tau_0 = \langle (q_0, ab), (q_1, b), (q_2, ) \rangle \]
\[ \tau_1 = \langle (q_0, abab), (q_1, bab), (q_2, ab), (q_1, b), (q_2, ) \rangle \]
\[ \tau_2 = \langle (q_0, ababab), (q_1, babab), (q_2, abab), (q_1, bab) \rangle \]
\[ \tau_3 = \langle (q_0, abaaa), (q_1, baaa), (q_2, aaa), (q_1, aa) \rangle \]

Then:

- \( \tau_0, \tau_1 \) are initial traces, reaching a final state
- \( \tau_2 \) is an initial trace, and is not maximal
- \( \tau_3 \) reaches a blocking state, but not a final state
Example: $\lambda$-term

We consider $\lambda$-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):

$\tau_0 = \langle \lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x))), 
\lambda y \cdot y \rangle$

$\tau_1 = \langle \lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x))), 
\lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x))), 
\lambda y \cdot ((\lambda x \cdot y)((\lambda x \cdot x x)(\lambda x \cdot x x))) \rangle$

Then:
- $\tau_0$ is a maximal trace; it reaches a blocking state (no more reduction can be done)
- $\tau_1$ 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:

\[
\begin{align*}
\ell_0 : & \quad x := 1; \\
\ell_1 : & \quad y := 0; \\
\ell_2 : & \quad \text{while}(x < 4)\{ \\
\ell_3 : & \quad y := y + x; \\
\ell_4 : & \quad x := x + 1; \\
\ell_5 : & \quad \} \\
\ell_6 : & \quad \text{(final program point)} \\
\end{align*}
\]

\[
\tau = \langle (\ell_0, (x = 6, y = 8)), (\ell_1, (x = 1, y = 8)), (\ell_2, (x = 1, y = 0)), (\ell_3, (x = 1, y = 0)), (\ell_4, (x = 1, y = 1)), (\ell_5, (x = 2, y = 1)), (\ell_3, (x = 2, y = 1)), (\ell_4, (x = 2, y = 3)), (\ell_5, (x = 3, y = 3)), (\ell_3, (x = 3, y = 3)), (\ell_4, (x = 3, y = 6)), (\ell_5, (x = 4, y = 6)), (\ell_6, (x = 4, y = 6)) \rangle
\]

- 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

\[ S = (\{a, b, c, d\}, \{(a, b), (b, a), (b, c)\}) \]

**Traces by length:**

<table>
<thead>
<tr>
<th>(i)</th>
<th>traces of length (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle d\rangle)</td>
</tr>
<tr>
<td>1</td>
<td>(\langle a, b\rangle, \langle b, a\rangle, \langle b, c\rangle)</td>
</tr>
<tr>
<td>2</td>
<td>(\langle a, b, a\rangle, \langle b, a, b\rangle, \langle a, b, c\rangle)</td>
</tr>
<tr>
<td>3</td>
<td>(\langle a, b, a, b\rangle, \langle b, a, b, a\rangle, \langle b, a, b, c\rangle)</td>
</tr>
</tbody>
</table>

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
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 \(I = \{\} \cup \{\langle s \rangle | s \in S\}\). Let \(F_*\) be the function defined by:

\[
F_* : \mathcal{P}(S^*) \rightarrow \mathcal{P}(S^*)
\]

\[
X \mapsto I \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle | \langle s_0, \ldots, s_n \rangle \in X \land s_n \rightarrow s_{n+1}\}
\]

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

\[
\text{lfp } F_* = [S]^* = \bigcup_{n \in \mathbb{N}} F_*^n(\emptyset)
\]
Fixpoint definition: proof (1), fixpoint existence

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

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

$$F_*(\bigcup_{X \in \mathcal{X}} X)$$

$$= I \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle | (\langle s_0, \ldots, s_n \rangle \in \bigcup_{U \in \mathcal{X}} U) \land s_n \rightarrow s_{n+1} \}$$

$$= I \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle | \exists U \in \mathcal{X}, (\langle s_0, \ldots, s_n \rangle \in U \land s_n \rightarrow s_{n+1} \}$$

$$= I \cup \left( \bigcup_{U \in \mathcal{X}} \{\langle s_0, \ldots, s_n, s_{n+1} \rangle | (\langle s_0, \ldots, s_n \rangle \in U \land s_n \rightarrow s_{n+1} \} \right)$$

$$= \bigcup_{U \in \mathcal{X}} F_*(U)$$

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

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

$$\text{lfp } F_* = \bigcup_{n \in \mathbb{N}} F_*^n(\emptyset)$$
We now show that $[S]^*$ is equal to $\text{lfp } F^*_\bullet$, by showing the property below, by induction over $n$:

$$\forall k < n, \langle s_0, \ldots, s_k \rangle \in F^n_\bullet(\emptyset) \iff \langle s_0, \ldots, s_k \rangle \in [S]^*$$

* at rank 0, only trace $\epsilon$ needs to be considered, and its 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 $n \geq 1$:

$$\langle s_0, \ldots, s_k, s_{k+1} \rangle \in [S]^* \\
\iff \langle s_0, \ldots, s_k \rangle \in [S]^* \land s_k \rightarrow s_{k+1} \\
\iff \langle s_0, \ldots, s_k \rangle \in F^n_\bullet(\emptyset) \land s_k \rightarrow s_{k+1} \quad (k < n \text{ since } k + 1 < n + 1) \\
\iff \langle s_0, \ldots, s_k, s_{k+1} \rangle \in F^{n+1}_\bullet(\emptyset)$$
Trace semantics fixpoint form: example

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

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

Then, the first iterates are:

\[
\begin{align*}
F^0_\ast(\emptyset) &= \emptyset \\
F^1_\ast(\emptyset) &= \{\emptyset, \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle\} \\
F^2_\ast(\emptyset) &= F^1_\ast(\emptyset) \cup \{\langle b, a \rangle, \langle a, b \rangle, \langle b, c \rangle\} \\
F^3_\ast(\emptyset) &= F^2_\ast(\emptyset) \cup \{\langle a, b, a \rangle, \langle b, a, b \rangle, \langle a, b, c \rangle\} \\
F^4_\ast(\emptyset) &= F^3_\ast(\emptyset) \cup \{\langle b, a, b, a \rangle, \langle a, b, a, b \rangle, \langle b, a, b, c \rangle\} \\
F^5_\ast(\emptyset) &= F^4_\ast(\emptyset) \cup \{\langle a, b, a, b, a \rangle, \langle b, a, b, a, b \rangle, \langle a, b, a, b, c \rangle\} \\
F^6_\ast(\emptyset) &= \ldots
\end{align*}
\]

The traces of $[S]^\ast$ of length $n + 1$ appear in $F^n_\ast(\emptyset)$.
Outline

1 Transition systems and small step semantics

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

3 Summary
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 $\llbracket . \rrbracket$ 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 $\pi$ 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

$\llbracket \pi \rrbracket = F_C(\llbracket \pi_0 \rrbracket, \ldots, \llbracket \pi_k \rrbracket)$, where $F_C$ depends only on syntactic construction $F_C$. 
Case of a simplified imperative language

Case of a sequence of two instructions $b \equiv \ell_0 : i_0; \ell_1 : i_1; \ell_2$:

$$[b]^* = [i_0]^* \cup [i_1]^* \cup \{\langle s_0, \ldots, s_m \rangle \mid \exists n \in [0, m], \langle s_0, \ldots, s_n \rangle \in [i_0]^* \land \langle s_n, \ldots, s_m \rangle \in [i_1]^*\}$$

This amounts to concatenating traces of $[i_0]^*$ and $[i_1]^*$ that share a state in common (necessarily at point $\ell_1$).

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 \( \lambda \)-calculus

Case of a \( \lambda \)-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 \( \lambda \)-calculus...
Outline

1. Transition systems and small step semantics

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

3. Summary
Non termination

Can the finite traces semantics express non termination?

Consider the case of our contrived system:

\[ S = \{a, b, c, d\} \quad (\rightarrow) = \{(a, b), (b, a), (b, c)\} \]

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 \(\prec\):
  \[ \langle a, b \rangle, \langle a, b, a \rangle, \langle a, b, a, b \rangle, \langle a, b, a, b, a \rangle, \ldots \in [S]^* \]
- 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 $S = (S, \rightarrow)$

**Definition**

The infinite traces semantics $[S]^\omega$ is defined by:

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

Infinite traces starting from an initial state (considering $S = (S, \rightarrow, S_I, S_F)$):

$$\{ \langle s_0, \ldots \rangle \in [S]^\omega \mid s_0 \in S_I \}$$

**Example:**

- contrived transition system defined by
  $$S = \{ a, b, c, d \} \quad (\rightarrow) = \{ (a, b), (b, a), (b, c) \}$$

- the infinite traces semantics contains **exactly two** traces
  $$[S]^\omega = \{ \langle a, b, \ldots, a, b, a, b, \ldots \rangle, \langle b, a, \ldots, b, a, b, a, \ldots \rangle \}$$
Fixpoint form

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

Intuitively, $\langle s_0, s_1, \ldots \rangle \in [S]^{\omega}$ 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 $F_{\omega}$ be defined by:

$$F_{\omega} : \mathcal{P}(S^{\omega}) \rightarrow \mathcal{P}(S^{\omega})$$

$$X \mapsto \{ \langle s_0, s_1, \ldots, s_n, \ldots \rangle \mid \langle s_1, \ldots, s_n, \ldots \rangle \in X \land s_0 \rightarrow s_1 \}$$

Then, we can show by induction that:

$$\sigma \in [S]^{\omega} \iff \forall n \in \mathbb{N}, \sigma \in F_{\omega}^{n}(S^{\omega})$$

$$\iff \bigcap_{n \in \mathbb{N}} F_{\omega}^{n}(S^{\omega})$$
Fixpoint form of the semantics of infinite traces

Infinite traces semantics as a fixpoint

Let $F_\omega$ be the function defined by:

$$F_\omega : \mathcal{P}(S^\omega) \longrightarrow \mathcal{P}(S^\omega)$$

$$X \longmapsto \{ \langle s_0, s_1, \ldots, s_n, \ldots \rangle | \langle s_1, \ldots, s_n, \ldots \rangle \in X \land s_0 \rightarrow s_1 \}$$

Then, $F_\omega$ is $\land$-continuous and thus has a greatest-fixpoint; moreover:

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

Proof sketch:

- the $\land$-continuity proof is similar as for the $\cup$-continuity of $F_*$
- by the dual version of Kleene's theorem, $\text{gfp } F_\omega$ exists and is equal to $\bigcap_{n \in \mathbb{N}} F_\omega^n(S^\omega)$, i.e. to $\llbracket S \rrbracket^\omega$ (similar induction proof)
Fixpoint form of the infinite traces semantics: iterates

Example, with the same simple transition system:

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

Then, the first iterates are:

$$
F_0^0(S) = S
$$
$$
F_0^1(S) = \langle a, b \rangle \cdot S \cup \langle b, a \rangle \cdot S \cup \langle b, c \rangle \cdot S
$$
$$
F_0^2(S) = \langle b, a, b \rangle \cdot S \cup \langle a, b, a \rangle \cdot S \cup \langle a, b, c \rangle \cdot S
$$
$$
F_0^3(S) = \langle a, b, a, b \rangle \cdot S \cup \langle b, a, b, a \rangle \cdot S \cup \langle b, a, b, c \rangle \cdot S
$$
$$
F_0^4(S) = \ldots
$$

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