Semantics

Material covered in sections 3.1 and 4.1 of
Introduction to Static Analysis: an Abstract Interpretation Perspective

Purpose of this lecture

We set up the basis for being able to study the static analysis of a small language:

- **syntax** of programs
- **semantics** of programs, i.e., formalization of how they run

Semantics is **a crucial prerequisite for static analysis**:

- the structure of semantics guides that of the analysis;
- the proof of soundness of the analysis is done with respect to the semantics

We consider two forms of semantics

- **compositional**: functions that compose
- **transitional**: state transitions that step

Outline

Syntax of expressions and conditions

A **few assumptions**:

- $\mathbb{V}$: **set of scalar values**
- $\mathbb{X}$: **finite set of variables**

Scalar expressions, meant to evaluate to scalar values, defined by induction, using a grammar:

$$
\begin{array}{rcll}
E & ::= & & \text{scalar expressions} \\
& | & n & \text{scalar constant } n \in \mathbb{V} \\
& | & \mathtt{x} & \text{variable } \mathtt{x} \in \mathbb{X} \\
& | & E \odot E & \text{binary operation}
\end{array}
$$

Boolean expressions, meant to evaluate to Booleans:

$$
\begin{array}{rcll}
B & ::= & & \text{Boolean expressions} \\
& | & \mathtt{x} \lessdot n & \text{comparison of a variable with a constant}
\end{array}
$$

Syntax of commands

Commands of a **basic, imperative language**, also defined by induction, using a grammar:

$$
\begin{array}{rcll}
C & ::= & & \text{commands} \\
& \mid & \texttt{skip} & \text{command that ``does nothing"} \\
& \mid & C;\ C & \text{sequence of commands} \\
& \mid & \texttt{x} := E & \text{assignment command} \\
& \mid & \texttt{input(x)} & \text{command reading of a value} \\
& \mid & \texttt{if}(B)\{C\}\texttt{else}\{C\} & \text{conditional command} \\
& \mid & \texttt{while}(B)\{C\} & \text{loop command} \\
P & ::= & C & \text{program}
\end{array}
$$

A program is simply a command.

Example

A very basic program:

$$
\begin{array}{l}
\mathtt{x} = 0; \\
\mathtt{while}(\mathtt{x} < 98)\{ \\
\quad \mathtt{input}(\mathtt{y}); \\
\quad \mathtt{if}(\mathtt{y} = 10)\,\{ \\
\quad\quad \mathtt{x} = \mathtt{x} + 110; \\
\quad \}\,\mathtt{else}\,\mathtt{if}(\mathtt{y} = 20)\,\{ \\
\quad\quad \mathtt{x} = \mathtt{x} + 1; \\
\quad \}\,\mathtt{else}\,\{\,\} \\
\}
\end{array}
$$

Many possible executions:

- may terminate with $\mathtt{x} = 98$
- may terminate with $\mathtt{x} = 110 + k$ with $1 \leq k < 98$
- may not terminate at all...

In fact, there are **infinitely many** possible executions

Verification problem: value analysis

```
x = 0;
while(x < 98){
    input(y);
    if(y = 10){
        x = x + 110;
    } else if(y = 20){
        x = x + 1;
    } else { }
}
```

- what is the value of x at the exit point ?
- same question, but for all program points ?
- same question, but regarding to y ?

Verification problem: correctness of assertions

```
x = 0;
while(x < 98){
    assert(x ≥ 0);
    input(y);
    if(y = 10){
        x = x + 110;
    assert(y < x);
    } else if(y = 20){
        x = x + 1;
    } else { }
}
assert(x < 200);
```

Some assertions were added inside the program.

- Can we assume that no execution will violate them ?
- More generally, can we prove that some class of runtime errors will never happen, such as division by zero or array index out-of-bounds dereferences

To answer such questions, **value information are needed**.

Verification problem: reachability

```
x = 0;
while(x < 98){
    input(y);
    if(y = 10) {
        x = x + 110;
    } else if(y = 20) {
        x = x + 1;
    } else { }
}
if(x > 210) {...}
```

Are all program points reachable ?

Useful to reason over:

- code optimization
- correctness (if some commands may perform offending operations)

Again, to answer such questions, **value information are needed**.

Need for a semantics

To assess whether such properties hold one needs to take into account:

- the **control flow** behavior of commands
- the **mathematical definition of operators**
  i.e., is the addition done with modular arithmetic ?
  i.e., with what precision/rounding are floating point operations computed ?
- the **execution order** (if there are side effects)
- the **error semantics**
  i.e., abrupt crash or undefined behavior ? or even, "implementation defined" behavior ?

To settle all these questions, a formal definition of program behaviors is required; this is the goal of semantics

Outline

Memory states

First step: semantics of expressions.
To evaluate an expression, one needs to read the value of variables.

Memory state:

- maps each variable to its value
  hence, a function

$$m : \mathbb{X} \to \mathbb{V}$$

- example: $\mathtt{x}$ stores 3 and $\mathtt{y}$ stores 8:

$$m : \left\{ \begin{array}{lcl} \mathtt{x} & \longmapsto & 3 \\ \mathtt{y} & \longmapsto & 8 \end{array} \right.$$

- for more complex languages with pointer and dynamic allocation, a more complex definition is needed

Semantics of expressions

How to evaluate an expression:

- maps an **expression** and a **memory state** into a **value**
- the definition of expression is inductive
  so the **evaluation** also proceeds **by induction over the syntax**

Definition:

$$[\![E]\!] : \mathbb{M} \longrightarrow \mathbb{V}$$

$$\begin{array}{rcl} [\![n]\!](m) & = & n \\ [\![\mathrm{x}]\!](m) & = & m(\mathrm{x}) \\ [\![E_0 \odot E_1]\!](m) & = & f_\odot([\![E_0]\!](m), [\![E_1]\!](m)) \end{array}$$

where $f_\odot : \mathbb{V} \times \mathbb{V} \longrightarrow \mathbb{V}$ evaluates operator $\odot$.

Example:

$$m : \left\{ \begin{array}{rcl} \mathrm{x} & \longmapsto & 3 \\ \mathrm{y} & \longmapsto & 8 \end{array} \right.$$

$$\begin{array}{rcl} [\![\mathrm{x} + 7]\!](m) & = & 10 \\ [\![\mathrm{x} * (\mathrm{y} + \mathrm{x})]\!](m) & = & 33 \end{array}$$

Semantics of conditions

Boolean conditions are very **similar**:

- maps an **expression** and a **memory state** into a **Boolean value**

Definition:

$$[\![B]\!] : \mathbb{M} \longrightarrow \mathbb{B}$$

$$[\![\mathtt{x} \otimes n]\!](m) \quad = \quad f_{\otimes}(m(\mathtt{x}), n)$$

where $f_{\otimes} : \mathbb{V} \times \mathbb{V} \longrightarrow \mathbb{V}$ evaluates operator $\otimes$.

Example:

$$m : \left\{ \begin{array}{ccc} \mathtt{x} & \longmapsto & 3 \\ \mathtt{y} & \longmapsto & 8 \end{array} \right.$$

$$[\![\mathtt{x} + 7 < \mathtt{x} * (\mathtt{y} + \mathtt{x})]\!](m) \quad = \quad \mathtt{true}$$

Outline

Principles and basic constructions

A first view at **commands**:

- **input a memory state**
- **return a modified memory state**

This suggests to define the **semantics of command** $\mathtt{C}$ $[\![\mathtt{C}]\!] : M \longrightarrow M$, but:

- some expressions do not return, e.g., $\mathtt{while}(0 < 1)\ \{\}$
- some expressions may return several states, e.g., $\mathtt{input}(\mathtt{x})$

Thus:

$$[\![\mathtt{C}]\!]_{\mathcal{P}} : \wp(\mathbb{M}) \longrightarrow \wp(\mathbb{M})$$

Compositionality:

the semantics of sequence is composition; that of skip is identity

$$\begin{array}{rcl} [\![\mathtt{C}_0;\ \mathtt{C}_1]\!]_{\mathcal{P}}(M) & = & [\![\mathtt{C}_1]\!]_{\mathcal{P}}([\![\mathtt{C}_0]\!]_{\mathcal{P}}(M)) \\ [\![\mathtt{skip}]\!]_{\mathcal{P}}(M) & = & M \end{array}$$

Memory modifying commands

Assignment command $\mathtt{x} := E$ evaluates E in the current memory and updates the value of $\mathtt{x}$

$$[\![\mathtt{x} := E]\!]_{\mathcal{P}}(M) \quad = \quad \{m[\mathtt{x} \mapsto [\![E]\!](m)] \mid m \in M\}$$

Assignment command $\mathtt{input(x)}$ updates the value of $\mathtt{x}$ with a non-deterministically chose value:

$$[\![\mathtt{input(x)}]\!]_{\mathcal{P}}(M) \quad = \quad \{m[\mathtt{x} \mapsto n] \mid m \in M, n \in \mathbb{V}\}$$

Conditions

Condition command $\mathtt{if}(B)\{C_0\}\mathtt{else}\{C_1\}$ first evaluates the condition and then C_0 or C_1 depending on the result.

The semantics should consider a *set* of executions where both cases may arise:

$$[\![\mathtt{if}(B)\{C_0\}\mathtt{else}\{C_1\}]\!]_{\mathcal{P}}(M) \quad = \quad [\![C_0]\!]_{\mathcal{P}}(\mathcal{F}_B(M)) \cup [\![C_1]\!]_{\mathcal{P}}(\mathcal{F}_{\neg B}(M))$$

Loops: a first (bounded) attempt

Loop command `while`$(B)\{C\}$ combines condition test and **unbounded iteration**.

Let us consider executions that spend **exactly n iterations** in the loop:

$$\mathcal{F}_{\neg B}\left(\llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B\right)^i(M)$$

We can combine this to describe all iterations that spend **at most n** iterations in the loop:

$$\bigcup_{0 \leq i \leq n} \mathcal{F}_{\neg B}\left(\llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B\right)^i(M)$$

Intuition: unfold an `if` command n times.

Loops: definition of the semantics

To consider unbounded iterations:

$$[\![\mathtt{while}(B)\{C\}]\!]_{\mathcal{P}}(M) \;=\; \mathcal{F}_{\neg B}\left(\bigcup_{i\geq 0}\left([\![C]\!]_{\mathcal{P}}\circ\mathcal{F}_{B}\right)^{i}(M)\right)$$

Notes:

- infinite unions exist since $\wp(M)$ is a complete lattice
- non-terminating executions do not appear in this formula
  indeed, it collects only output states

Summary

The semantics of commands at a glance

$$[\![C]\!]_{\mathcal{P}} : \wp(\mathbb{M}) \longrightarrow \wp(\mathbb{M})$$

$$\begin{array}{rcl}
[\![\mathtt{skip}]\!]_{\mathcal{P}}(M) & = & M \\
[\![C_0;\ C_1]\!]_{\mathcal{P}}(M) & = & [\![C_1]\!]_{\mathcal{P}}([\![C_0]\!]_{\mathcal{P}}(M)) \\
[\![\mathtt{x} := E]\!]_{\mathcal{P}}(M) & = & \{ m[\mathtt{x} \mapsto [\![E]\!](m)] \mid m \in M \} \\
[\![\mathtt{input}(\mathtt{x})]\!]_{\mathcal{P}}(M) & = & \{ m[\mathtt{x} \mapsto n] \mid m \in M, n \in \mathbb{V} \} \\
[\![\mathtt{if}(B)\{C_0\}\mathtt{else}\{C_1\}]\!]_{\mathcal{P}}(M) & = & [\![C_0]\!]_{\mathcal{P}}(\mathcal{F}_B(M)) \cup [\![C_1]\!]_{\mathcal{P}}(\mathcal{F}_{\neg B}(M)) \\
[\![\mathtt{while}(B)\{C\}]\!]_{\mathcal{P}}(M) & = & \mathcal{F}_{\neg B}\left(\bigcup_{i \geq 0} ([\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B)^i (M) \right)
\end{array}$$

Instrumentation of the semantics to calculate **all reachable states**:

- add an argument mapping control states to sets of memory states

Exercises:

- extend the language/semantics: variables scopes, switch commands...
- implement an interpreter in your favorite language

Outline

Transitional semantics

A program execution is a sequence of state transitions from an initial state.

- handy for languages whose compositional semantics (also known as *denotational semantics*) is not obvious
  - for programming languages with dynamic controls (e.g., function calls, gotos, exceptions, or dynamic method dispatches)
- defining the compositional semantics for such features needs an advanced knowledge in programming language semantics
- transitional semantics (one style of *operational semantics*) is relatively easy to define

States

Let a program state be a pair (l, m) of a program label l and the machine state m at that program label during execution.

- program label l denotes the part of the program that is to be executed next.
- machine state m is usually the memory state that contains the effect of the program's hitherto execution and a data for the program's continuation.
    - for conventional imperative languages with local blocks and function calls, the machine state would consist of
    - a memory (a table from locations to storable values), an environment (a table from program variables to locations), and a continuation (a stack of return contexts, where a return context is a program label and an environment to resume at function return)

State transitions

- one-step state transition

$$(l, m) \hookrightarrow (l', m')$$

- a sequence

$$(l_0, m_0) \hookrightarrow (l_1, m_1) \hookrightarrow (l_2, m_2) \hookrightarrow \cdots$$

  of state transitions is the chain that links the one-step transitions

$$(l_0, m_0) \hookrightarrow (l_1, m_1),\ (l_1, m_1) \hookrightarrow (l_2, m_2),\ \cdots.$$

- a transition sequence for a program can be infinitely long if the program has non-terminating executions.
- the number of transition sequences can be infinite too if the initial states can be infinitely many.

Transition sequence example

Example program:

$$\begin{array}{l} \texttt{input(x);} \\ \texttt{while}\ (\texttt{x} \leq 99) \\ \qquad \{\texttt{x} := \texttt{x} + 1\} \end{array}$$

The labeled representation:

$$\begin{array}{ll} 0 & \texttt{input(x)} \\ 1 & \texttt{while}\ (\texttt{x} \leq 99) \\ 2 & \qquad \texttt{x} := \texttt{x} + 1 \\ 3 & \end{array}$$

From empty memory $\emptyset$, some transition sequences are:

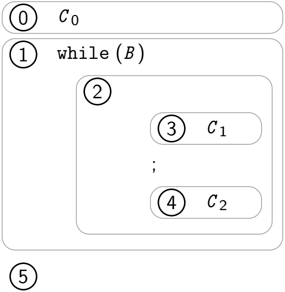
- for input 100:
  $(0, \emptyset) \hookrightarrow (1, x \mapsto 100) \hookrightarrow (3, x \mapsto 100)$
- for input 99:
  $(0, \emptyset) \hookrightarrow (1, x \mapsto 99) \hookrightarrow (2, x \mapsto 99) \hookrightarrow (1, x \mapsto 100) \hookrightarrow (3, x \mapsto 100)$
- for input 0:
  $(0, \emptyset) \hookrightarrow (1, x \mapsto 0) \hookrightarrow (2, x \mapsto 0) \hookrightarrow (1, x \mapsto 1) \hookrightarrow \cdots \hookrightarrow (3, x \mapsto 100)$

Example simple imperative language

$$
\begin{array}{lcll}
\mathtt{x} & \in & \mathbb{X} & \text{program variables} \\
C & ::= & & \text{statements} \\
 & | & \mathtt{skip} & \text{no-op statement} \\
 & | & C \,;\, C & \text{sequence of statements} \\
 & | & \mathtt{x} := E & \text{assignment} \\
 & | & \mathtt{input}(\mathtt{x}) & \text{read an integer input} \\
 & | & \mathtt{if}(B)\{C\}\mathtt{else}\{C\} & \text{condition statement} \\
 & | & \mathtt{while}(B)\{C\} & \text{loop statement} \\
 & | & \mathtt{goto}\ E & \text{goto with dynamically computed label} \\
E & & & \text{expression} \\
B & & & \text{boolean expression} \\
P & ::= & C & \text{program}
\end{array}
$$

- an expression E computes an integer or a program label
- labels are integers, unique to every statement of the program

Example: program labels and execution order



$$\begin{array}{ll} \mathtt{next}(0) = 1 & \\ \mathtt{nextTrue}(1) = 2 & \mathtt{next}(2) = 3 \\ \mathtt{nextFalse}(1) = 5 & \mathtt{next}(3) = 4 \\ \mathtt{next}(4) = 1 & \end{array}$$

Except for "goto E", the execution order (or *control flow*) between the statements in a program is clear from the program syntax.

Definitions of next, nextTrue, and nextFalse

Given a program p and label l_{end} for the end label of the program, $\langle\!\langle \mathtt{p}, l_{\text{end}} \rangle\!\rangle$ collects the function graphs of `next`, `nextTrue`, and `nextFalse`.

Let l be $\mathtt{label}(C)$ in:

$$
\begin{array}{rcl}
\langle\!\langle C, l' \rangle\!\rangle = \text{case } C \text{ of} & & \\
\mathtt{skip} & : & \{\mathtt{next}(l) = l'\} \\
\mathtt{x} := E & : & \{\mathtt{next}(l) = l'\} \\
\mathtt{input}(\mathtt{x}) & : & \{\mathtt{next}(l) = l'\} \\
C_1; C_2 & : & \{\mathtt{next}(l) = \mathtt{label}(C_1)\} \cup \langle\!\langle C_1, \mathtt{label}(C_2) \rangle\!\rangle \cup \langle\!\langle C_2, l' \rangle\!\rangle \\
\mathtt{if}(B)\{C_1\}\mathtt{else}\{C_2\} & : & \{\mathtt{nextTrue}(l) = \mathtt{label}(C_1), \mathtt{nextFalse}(l) = \mathtt{label}(C_2)\} \\
 & & \cup \langle\!\langle C_1, l' \rangle\!\rangle \cup \langle\!\langle C_2, l' \rangle\!\rangle \\
\mathtt{while}(B)\{C\} & : & \{\mathtt{nextTrue}(l) = \mathtt{label}(C), \mathtt{nextFalse}(1) = l'\} \cup \langle\!\langle C, l \rangle\!\rangle \\
\mathtt{goto}\ E & : & \{\} \qquad \text{(* to be determined at run-time by evaluating } E \text{ *)}
\end{array}
$$

State transition definition (1/2)

$$\begin{array}{llcll} \text{states} & \mathbb{S} & = & \mathbb{L} \times \mathbb{M} & \text{pair of label \& memory} \\ \text{memories} & \mathbb{M} & = & \mathbb{X} \to \mathbb{V} & \text{finite map from variables to values} \\ \text{values} & \mathbb{V} & = & \mathbb{Z} \cup \mathbb{L} & \text{integer or label.} \end{array}$$

The state transition relation $(l, m) \hookrightarrow (l', m')$ is:

$$\begin{array}{rcl} \texttt{skip} & : & (l, m) \hookrightarrow (\texttt{next}(l),\ m) \\ \texttt{input(x)} & : & (l, m) \hookrightarrow (\texttt{next}(l),\ \mathit{update}_{\texttt{x}}(m, z)) \quad \text{for int } z \\ \texttt{x} := E & : & (l, m) \hookrightarrow (\texttt{next}(l),\ \mathit{update}_{\texttt{x}}(m, \mathit{eval}_E(m))) \\ C_1; C_2 & : & (l, m) \hookrightarrow (\texttt{next}(l),\ m) \\ \texttt{if}(B)\{C_1\}\texttt{else}\{C_2\} & : & (l, m) \hookrightarrow (\texttt{nextTrue}(l),\ \mathit{filter}_B(m)) \\ & : & (l, m) \hookrightarrow (\texttt{nextFalse}(l),\ \mathit{filter}_{\neg B}(m)) \\ \texttt{while}(B)\{C\} & : & (l, m) \hookrightarrow (\texttt{nextTrue}(l),\ \mathit{filter}_B(m)) \\ & : & (l, m) \hookrightarrow (\texttt{nextFalse}(l),\ \mathit{filter}_{\neg B}(m)) \\ \texttt{goto } E & : & (l, m) \hookrightarrow (\mathit{eval}_E(m),\ m) \end{array}$$

State transition definition (2/2)

The auxiliary operations are:

- $update_{\mathtt{x}}(m, v)$ updates m for $\mathtt{x}$ with v, i.e., returns a memory that is the same as m except its image for $\mathtt{x}$ is v.
- $eval_E(m)$ computes the value of expression E for memory m
- $filter_B(m)$ is m if the value of B for m is true. Otherwise, no transition happens.
- $filter_{\neg B}(m))$ is m if the value of B for m is false. Otherwise, no transition happens.

Semantics as reachable states

- given a program, let I be the set of its initial states, and let *Step* be the powerset-lifted version of $\hookrightarrow$:

$$\begin{array}{l} Step : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S}) \\ Step(X) = \{s' \mid s \hookrightarrow s', s \in X\} \end{array}$$

- then, the set of all reachable states of a program from the initial state set I is

$$\bigcup_{i \geq 0} Step^i(I)$$

  where $Step^0(X) = X$ and $Step^{i+1}(X) = Step(Step^i(X))$

- the above set corresponds to an entity called "least fixpoint" of monotonic function F

$$\begin{array}{l} F : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S}) \\ F(X) = I \cup Step(X). \end{array}$$

Outline

Important points to remember, and what to learn next

Summary:

- typical definitions heavily rely on **induction** over the syntax for compositonal semantics, **global fixpoint** for transitional semantics
- compositional and transitional **capture the same information** but with different formats
- as such they are adapted for **different applications**

What comes next ?

- if interested, learn more about different kinds of semantics operational, denotational...
- abstraction, i.e., logical connection across two semantics