Semantics

The semantics of a programming language is concerned with the meaning of programs, that is, how programs behave when executed on computers.

The semantics of a programming language assigns a precise meaning to every sentence of the language that can be formed using the given syntax definition. There are three approaches to define the semantics of a programming language:

- Axiomatic semantics,
- Operational semantics,
- Denotational semantics.
The Hilbert-style Proof System

- A Hilbert-style proof system consists of axioms and proof rules.
  - An axiom of a proof system is a formula that is provable by definition.
  - An inference rule asserts that if some list of formulas is provable, then so is another formula.
  - A proof is a structured object built from formulas according to constraints established by a set of axioms and inference rules.

- The rule format:

  \[
  \frac{\text{Premise}_1 \quad \text{Premise}_2 \quad \ldots \quad \text{Premise}_n}{\text{Conclusion}}
  \]

- We construct a proof from proofs:

  \[
  \frac{\text{Premise}_1 \quad \text{Premise}_2 \quad \ldots \quad \text{Premise}_n}{\text{Conclusion}_1 \quad \text{Conclusion}_2 \quad \ldots \quad \text{Conclusion}_n}
  \]
Axiomatic Semantics

The axiomatic semantics is a formal (proof) system for deriving equations between expressions.

The basic idea of the axiomatic method is to define the meaning of language elements indirectly using logical assertions. For example, we can write \( \{E_1\} C \{E_2\} \), called a Hoare triple, to state that if the boolean expression \( E_1 \) holds prior the computation of \( C \), and if \( C \) terminates, then the boolean expression \( E_2 \) must hold as well.

Examples:

\[
\{ a > 0 \} a := a + 1 \{ a > 1 \}
\]

\[
\{E_1\} C_1 \{E_2\} \quad \{E_2\} C_2 \{E_3\}
\]

\[
\{E_1\} C_1; C_2 \{E_3\}
\]
Example Rules

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Proof Rule</th>
</tr>
</thead>
</table>
| \( C = \text{def} \) \[
C.\text{Target} := C.\text{Source} \]
| \[
\text{true} \]
\[
\{Q[C.\text{Source}\setminus C.\text{Target}]\}C\{Q\} \]
| \( C = \text{def} \)
\[
\text{if } C.\text{Test} \]
\[\text{then } C.\text{Then} \]
\[\text{else } C.\text{Else} \]
| \[\{C.\text{Test} \land P\}C.\text{Then}\{Q\} \]
\[\{\neg C.\text{Test} \land P\}C.\text{Else}\{Q\} \]
| \[
(P)C\{Q\} \]
| Rule of Consequence |
| \[P \Rightarrow P' \]
\[\{P'\}C\{Q'\} \]
\[Q' \Rightarrow Q \]
| \[\{P\}C\{Q\} \]
Using axiomatic semantics, we need to prove the validity of a given Hoare triple.

**Example:**

\[ C \overset{\text{def}}{=} \begin{align*}
\{ & \text{true} \} \\
& \text{if } (a \geq b) \\
& \quad \text{then} \\
& \quad m = a; \\
& \quad \text{else} \\
& \quad m = b; \\
& \{ m = \max(a, b) \}
\end{align*} \]
Proof

Premise I:

\[ a \geq b \Rightarrow a = \max(a, b) \]

\[ \text{true} \]

\[ \{a = \max(a, b)\} m = a; \{m = \max(a, b)\} \]

\[ \{a \geq b\} m = a; \{m = \max(a, b)\} \]

Premise II:

\[ a < b \Rightarrow b = \max(a, b) \]

\[ \text{true} \]

\[ \{b = \max(a, b)\} m = b; \{m = \max(a, b)\} \]

\[ \{a < b\} m = b; \{m = \max(a, b)\} \]

\[ \{a \geq b \land \text{true}\} m = a; \{m = \max(a, b)\} \]

\[ \{a < b \land \text{true}\} m = b; \{m = \max(a, b)\} \]

\[ \{\text{true}\} C\{m = \max(a, b)\} \]
Operational Semantics

- The operational semantics is based on a directed form of equational reasoning called “reduction”. Reduction may be regarded as a form of symbolic evaluation.

- The basic idea of the operational method is to define the meaning of the language elements by means of a (labeled) transition system.

- The operational semantics definition provides means to display the computation steps undertaken when a program is evaluated to its output.

- Some forms of operational semantics are interpreted-based, with instruction counters, data structures, and the like, and others are inference rule-based, with proof trees that show control flows and data dependencies.
# Example Transition Rules

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Transition Rule</th>
</tr>
</thead>
</table>
| \( C = \text{def} \) \begin{align*} & \text{C.Target} := \text{C.Source} \\
|                 | \end{align*}          | \( \sigma(\text{C.Source}) = \nu \) |
|                 | \( \sigma(\text{C.Target := C.Source}) \rightarrow \sigma \cup \{(\text{C.Target}, \nu)\} \) |
| \( C = \text{def} \) \begin{align*} & \text{if C.Test} \\
|                 | \text{then C.Then} \\
|                 | \text{else C.Else} \end{align*} | \( \sigma(\text{C.Test = true}) \sigma(\text{C.Then}) \rightarrow \sigma' \) |
|                 | \( \sigma(\text{if C.Test then C.Then else C.Else}) \rightarrow \sigma' \) |
|                 | \( \sigma(\text{C.Test = false}) \sigma(\text{C.Else}) \rightarrow \sigma' \) |
|                 | \( \sigma(\text{if C.Test then C.Then else C.Else}) \rightarrow \sigma' \) |
Denotational Semantics

The denotational semantics, or model theory, is defined in the spirit of equational logic or first-order logic. A denotational semantics definition (model) consists of a family of sets, one for each type, with the property that each well-typed expression may be interpreted as a specific element of the appropriate set.

The denotational semantics is a recursive definition that maps well-typed derivation trees to their mathematical meanings. For example, the set Bool consists of two meanings: Bool = \{true, false\} and an operation

\[ \text{not} : \text{Bool} \to \text{Bool} \text{ with } \text{not}(\text{false}) = \text{true}, \text{not}(\text{true}) = \text{false}. \]

The denotational method does not maintain states, but the meaning of a program is given as a function that interprets all language elements of a given program as elements of a corresponding set of values.
Example Meaning Functions

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Meaning Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C \overset{\text{def}}{=} )</td>
<td>( f(\langle C.\text{Target} := C.\text{Source} \rangle, \sigma) = )</td>
</tr>
<tr>
<td>( C.\text{Target} := C.\text{Source} )</td>
<td>\quad \text{if } (f(\langle C.\text{Target} \rangle, \sigma) = \nu) )</td>
</tr>
<tr>
<td></td>
<td>\quad \text{then } \sigma \cup { f(\langle C.\text{Target} \rangle, \sigma), f(\langle C.\text{Source} \rangle, \sigma) } } )</td>
</tr>
<tr>
<td></td>
<td>\quad \text{else ERROR}</td>
</tr>
<tr>
<td>( C \overset{\text{def}}{=} )</td>
<td>( f(\langle \text{if } C.\text{Test then } C.\text{Then else } C.\text{Else} \rangle, \sigma) = )</td>
</tr>
<tr>
<td>( \text{if } C.\text{Test} )</td>
<td>\quad \text{if } f(\langle C.\text{Test} \rangle, \sigma) )</td>
</tr>
<tr>
<td>( \text{then } C.\text{Then} )</td>
<td>\quad \text{then } f(\langle C.\text{Then} \rangle, \sigma) )</td>
</tr>
<tr>
<td>( \text{else } C.\text{Else} )</td>
<td>\quad \text{else } f(\langle C.\text{Else} \rangle, \sigma) ) }</td>
</tr>
</tbody>
</table>