Programming Language Semantics
How to give a formal meaning to a program

Steffen Lösch

University of Cambridge - Computer Laboratory
Why Formal Semantics?

Steffen Lösch

Programming Language Semantics
Why Formal Semantics?

An error has occurred. To continue:

Press Enter to return to Windows, or

Press CTRL+ALT+DEL to restart your computer. If you do this, you will lose any unsaved information in all open applications.

Error: 0E: 016F: BFF9B3D4

Press any key to continue _
Why Formal Semantics?
Some results of program errors:

- 1999: Mars Polar Lander crashes into Mars
- 1985: Deaths through radiation overdose in therapy
- 2009: Google mistakes the entire internet for malware
Some results of program errors:

▷ 1999: Mars Polar Lander crashes into Mars
Why Formal Semantics? (2)

Some results of program errors:

- 1999: Mars Polar Lander crashes into Mars
- 1985: Deaths through radiation overdose in therapy
Some results of program errors:

- 1999: Mars Polar Lander crashes into Mars
- 1985: Deaths through radiation overdose in therapy
- 2009: Google mistakes the entire internet for malware
Some results of program errors:

- 1999: Mars Polar Lander crashes into Mars
- 1985: Deaths through radiation overdose in therapy
- 2009: Google mistakes the entire internet for malware

How to avoid errors?
Some results of program errors:

- 1999: Mars Polar Lander crashes into Mars
- 1985: Deaths through radiation overdose in therapy
- 2009: Google mistakes the entire internet for malware
Some results of program errors:

- 1999: Mars Polar Lander crashes into Mars
- 1985: Deaths through radiation overdose in therapy
- 2009: Google mistakes the entire internet for malware

How to avoid errors?

Testing  
Verification
Overview

Motivation

Example programming language
Overview

Motivation

Example programming language

Operational semantics
Running Example: IMP

Commands

\[ c ::= \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid x := c \mid c_1 ; c_2 \mid \text{skip} \]

Boolean expressions

\[ b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2 \]

Arithmetic expressions

\[ a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \]

Numbers:
\[ n \in \{-\ldots, -1, 0, 1, \ldots\} \]

Memory locations:
\[ x \in \{l_0, l_1, l_2, \ldots\} \]
Running Example: IMP

A very very simple programming language...

Commands:
- `c ::= if b then c₁ else c₂ | while b do c | x := c | c₁; c₂ | skip`

Boolean expressions `Bexp`:
- `b ::= true | false | a₁ = a₂ | a₁ ≤ a₂ | ¬b | b₁ ∧ b₂ | b₁ ∨ b₂`

Arithmetic expressions `Aexp`:
- `a ::= n | x | a₁ + a₂ | a₁ − a₂ | a₁ × a₂`

Numbers: `n ∈ {..., −1, 0, 1, ...}`

Memory locations: `x ∈ {l₀, l₁, l₂, ...}`
Running Example: IMP

A very very simple programming language...

Commands **Com**: 
\[ c ::= \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid x := c \mid c_1; c_2 \mid \text{skip} \]
Running Example: IMP

A very very simple programming language...

Commands **Com**:  
\[ c ::= \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid x := c \mid c_1; c_2 \mid \text{skip} \]

Boolean expressions **Bexp**:  
\[ b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2 \]
Running Example: IMP

A very very simple programming language...

**Commands Com:**
\[ c ::= \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid x := c \mid c_1 ; c_2 \mid \text{skip} \]

**Boolean expressions Bexp:**
\[ b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2 \]

**Arithmetic expressions Aexp:**
\[ a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \]
A very very simple programming language...

Commands **Com**:  
\[ c ::= \text{if } b \text{ then } c_1 \text{ else } c_2 | \text{while } b \text{ do } c | x := c | c_1; c_2 | \text{skip} \]

Boolean expressions **Bexp**:  
\[ b ::= \text{true} | \text{false} | a_1 = a_2 | a_1 \leq a_2 | \neg b | b_1 \land b_2 | b_1 \lor b_2 \]

Arithmetic expressions **Aexp**:  
\[ a ::= n | x | a_1 + a_2 | a_1 - a_2 | a_1 \times a_2 \]

Numbers: \( n \in \{-\ldots,-1,0,1,\ldots\} \)

Memory locations: \( x \in \{l_0, l_1, l_2, \ldots\} \)
Even though IMP is very simple, it is fairly powerful.
Even though \textbf{IMP} is very simple, it is fairly powerful.

A program computing the faculty of a number (input stored in $l_0$, output stored in $l_1$):

\begin{verbatim}
l_1 := 1;
while 2 \leq l_0 do
  l_1 := l_1 \times l_0;
  l_0 := l_0 - 1
\end{verbatim}
Even though IMP is very simple, it is fairly powerful.

A program computing the faculty of a number (input stored in $l_0$, output stored in $l_1$):

$$!4 = 4 \times 3 \times 2 \times 1$$
Even though **IMP** is very simple, it is fairly powerful.

A program computing the faculty of a number (input stored in $l_0$, output stored in $l_1$):

1. $l_1 := 1$
2. while $2 \leq l_0$ do
   1. $l_1 := l_1 \times l_0$
   2. $l_0 := l_0 - 1$

$4! = 4 \times 3 \times 2 \times 1$
Even though IMP is very simple, it is fairly powerful.

A program computing the faculty of a number (input stored in $l_0$, output stored in $l_1$):

$$
l_1 := 1; \\
\text{while } 2 \leq l_0 \text{ do} \\
\quad l_1 := l_1 \times l_0; \\
\quad l_0 := l_0 - 1
$$

How can we prove things about IMP-programs?
Operational Semantics of IMP

Idea:
Define an abstract machine and describe how it would execute IMP-programs.

Steffen Lösch
Programming Language Semantics
Idea: Define an abstract machine and describe how it would execute IMP-programs.
The abstract machine consists of

- A memory: stores numbers in memory locations.

- A memory state is a function \( \{ l_0, l_1, \ldots \} \rightarrow \{ \ldots, -1, 0, 1, \ldots \} \).

- An execution mechanism for IMP-programs \( \langle p, m \rangle \).

Machine configuration:
- \( p \): program to be executed.
- \( m \): memory state.

The execution mechanism works by taking computation steps and thereby changing the machine configuration:

\[ \langle p, m \rangle \rightarrow \langle p', m' \rangle \]
The abstract machine consists of

- A memory: stores numbers in memory locations

A memory state is a function \( \{ l_0, l_1, \ldots \} \rightarrow \{ \ldots, -1, 0, 1, \ldots \} \)
Operational Semantics of IMP (2)

The abstract machine consists of

- A memory: stores numbers in memory locations
  
  A memory state is a function \( \{l_0, l_1, \ldots\} \rightarrow \{\ldots, -1, 0, 1, \ldots\} \)

- An execution mechanism for IMP-programs
Operational Semantics of **IMP** (2)

The abstract machine consists of

- A memory: stores numbers in memory locations

  A memory state is a function \( \{l_0, l_1, \ldots\} \rightarrow \{\ldots, -1, 0, 1, \ldots\} \)

- An execution mechanism for **IMP**-programs

  Machine configuration: \( \langle p, m \rangle \)

  - \( p \): program to be executed
  - \( m \): memory state
The abstract machine consists of

- A memory: stores numbers in memory locations
  
  A memory state is a function \( \{l_0, l_1, \ldots \} \rightarrow \{\ldots, -1, 0, 1, \ldots \} \)

- An execution mechanism for \textbf{IMP}-programs

The execution mechanism works by taking computation steps and thereby changing the machine configuration:

\[
\langle p, m \rangle \rightarrow \langle p', m' \rangle
\]
Operational Semantics of \textbf{IMP} (3)

An example computation:

\[
\langle l_0 := 3; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{} l_0 \mapsto 0, \ldots \rangle
\]

\[
\rightarrow \langle \text{skip}; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{} l_0 \mapsto 3, \ldots \rangle
\]

\[
\rightarrow \langle \text{if } 3 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{} l_0 \mapsto 3, \ldots \rangle
\]

\[
\rightarrow \langle \text{if true then } 7 \text{ else } l_0 \rangle, \{} l_0 \mapsto 3, \ldots \rangle
\]

\[
\rightarrow \langle 7, \{} l_0 \mapsto 3, \ldots \rangle
\]
An example computation:

\[
\langle l_0 := 3; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0, \{l_0 \mapsto 0, \ldots\}\rangle \\
\rightarrow \langle \text{skip}; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0, \{l_0 \mapsto 3, \ldots\}\rangle
\]
An example computation:

\[
\begin{align*}
\langle l_0 := 3; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{l_0 \mapsto 0, \ldots\} \\
\rightarrow \langle \text{skip}; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{l_0 \mapsto 3, \ldots\} \\
\rightarrow \langle \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{l_0 \mapsto 3, \ldots\}
\end{align*}
\]
Operational Semantics of **IMP** (3)

An example computation:

\[
\langle l_0 := 3; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{l_0 \mapsto 0, \ldots\}
\]

\[\rightarrow \langle \text{skip}; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{l_0 \mapsto 3, \ldots\}\]

\[\rightarrow \langle \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{l_0 \mapsto 3, \ldots\}\]

\[\rightarrow \langle \text{if } 3 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{l_0 \mapsto 3, \ldots\}\]
Operational Semantics of IMP (3)

An example computation:

\[
\begin{align*}
\langle l_0 := 3; \text{if} \ l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 0, \ldots \} \\
\rightarrow \langle \text{skip}; \text{if} \ l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \} \\
\rightarrow \langle \text{if} \ l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \} \\
\rightarrow \langle \text{if} \ 3 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \} \\
\rightarrow \langle \text{if} \ \text{true} \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \}
\end{align*}
\]
Operational Semantics of IMP (3)

An example computation:

\[
\langle l_0 := 3; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 0, \ldots \} \\
\rightarrow \langle \text{skip}; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \} \\
\rightarrow \langle \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \} \\
\rightarrow \langle \text{if } 3 \leq 5 \text{ then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \} \\
\rightarrow \langle \text{if true then } 7 \text{ else } l_0 \rangle, \{ l_0 \mapsto 3, \ldots \} \\
\rightarrow \langle 7 \rangle, \{ l_0 \mapsto 3, \ldots \}
\]
An example computation:

\[
\begin{align*}
\langle l_0 := 3; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0, \{l_0 \mapsto 0, \ldots\}\rangle \\
\rightarrow \langle \text{skip}; \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0, \{l_0 \mapsto 3, \ldots\}\rangle \\
\rightarrow \langle \text{if } l_0 \leq 5 \text{ then } 7 \text{ else } l_0, \{l_0 \mapsto 3, \ldots\}\rangle \\
\rightarrow \langle \text{if } 3 \leq 5 \text{ then } 7 \text{ else } l_0, \{l_0 \mapsto 3, \ldots\}\rangle \\
\rightarrow \langle \text{if } \text{true} \text{ then } 7 \text{ else } l_0, \{l_0 \mapsto 3, \ldots\}\rangle \\
\rightarrow \langle 7, \{l_0 \mapsto 3, \ldots\}\rangle
\end{align*}
\]

The steps the abstract machine can take are defined by rules. For example:

\[
\frac{\langle n_1 \leq n_2, m \rangle \rightarrow \langle \text{false}, m \rangle}{\text{if } n_1 > n_2}
\]

\[
\frac{\langle p_1, m \rangle \rightarrow \langle p_1', m' \rangle}{\langle p_1 \leq p_2, m \rangle \rightarrow \langle p_1', p_2, m' \rangle}
\]

\[
\frac{\langle p_2, m \rangle \rightarrow \langle p_2', m' \rangle}{\langle n_1 \leq p_2, m \rangle \rightarrow \langle n_1 \leq p_2', m' \rangle}
\]
Operational Semantics of IMP (4)

Some more rules:

\[
\begin{align*}
\langle l := n, m \rangle & \rightarrow \langle \text{skip}, m + \{l \mapsto n\} \rangle \\
\langle l := p, m \rangle & \rightarrow \langle l := p', m' \rangle \\
\langle \text{skip}; p, m \rangle & \rightarrow \langle p, m \rangle \\
\langle l, m \rangle & \rightarrow \langle n, m \rangle \\
\langle \text{while} \; p \; \text{do} \; p', m \rangle & \rightarrow \langle \text{if} \; p \; \text{then}(p'; \text{while} \; p \; \text{do} \; p') \; \text{else} \; \text{skip}, m \rangle
\end{align*}
\]

The operational semantics allows us to prove theorems about the programming language IMP. For example:

- If \( \langle p, m \rangle \rightarrow \langle p', m' \rangle \) and \( \langle p, m \rangle \rightarrow \langle p'', m'' \rangle \), then \( \langle p', m' \rangle = \langle p'', m'' \rangle \).
Operational Semantics of IMP (4)

Some more rules:

\[
\begin{align*}
\langle l := n, m \rangle & \rightarrow \langle \text{skip}, m + \{ l \mapsto n \} \rangle \\
\langle l \mapsto p, m \rangle & \rightarrow \langle l := p', m' \rangle \\
\langle \text{skip}; p, m \rangle & \rightarrow \langle p, m \rangle \\
\langle l, m \rangle & \rightarrow \langle n, m \rangle \\
\langle \text{while } p \text{ do } p', m \rangle & \rightarrow \langle \text{if } p \text{ then } (p'; \text{while } p \text{ do } p') \text{ else skip}, m \rangle \\
\end{align*}
\]

The operational semantics allows us to prove theorems about the programming language IMP. For example:

- If \( \langle p, m \rangle \rightarrow \langle p', m' \rangle \) and \( \langle p, m \rangle \rightarrow \langle p'', m'' \rangle \) then \( \langle p', m' \rangle = \langle p'', m'' \rangle \).
Idea: Semantics should map a program directly to its "mathematical meaning".

Characteristics:

▶ Each program, \( p \), is given a denotation, \( J_p \) - a mathematical object representing the contribution of \( p \) to the meaning of any complete program in which it occurs.

▶ The denotation of a program is determined just by the denotations of its subprograms (one says that the semantics is compositional).
Denotational Semantics

**Idea:** Semantics should map a program directly to its "mathematical meaning".
Idea:  Semantics should map a program directly to its "mathematical meaning".

Characteristics:

- Each program, $p$, is given a denotation, $\llbracket p \rrbracket$ - a mathematical object representing the contribution of $p$ to the meaning of any complete program in which it occurs.
- The denotation of a program is determined just by the denotations of its subprograms (one says that the semantics is *compositional*).
Define $\text{IMP}^-$ to be $\text{IMP}$ without while-loops. $\text{IMP}^-$ can be given a very easy denotational semantics:
Define $\text{IMP}^-$ to be $\text{IMP}$ without while-loops. $\text{IMP}^-$ can be given a very easy denotational semantics:

The denotational semantics of $\text{IMP}^-$ consists of the functions

$$\begin{align*}
B[-] &: \text{Bexp} \rightarrow (\text{State} \rightarrow \{\text{true, false}\}) \\
A[-] &: \text{Aexp} \rightarrow (\text{State} \rightarrow \{\ldots, -1, 0, 1, \ldots\}) \\
C[-] &: \text{Com} \rightarrow (\text{State} \Rightarrow \text{State})
\end{align*}$$

where

$$\text{State} = (\{l_0, l_1, \ldots\} \rightarrow \{\ldots, -1, 0, 1, \ldots\})$$
Denotational Semantics of $\text{IMP}^-$ (2)

$\mathcal{B}[-], \mathcal{A}[-]$ and $\mathcal{C}[-]$ are defined recursively. For example:

\[
\begin{align*}
\mathcal{A}[n](s) &= n \\
\mathcal{A}[l](s) &= s(l) \\
\mathcal{A}[a_1 + a_2](s) &= \mathcal{A}[a_1](s) + \mathcal{A}[a_2](s)
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}[\text{true}](s) &= \text{true} \\
\mathcal{B}[\text{false}](s) &= \text{false} \\
\mathcal{B}[a_1 = a_2](s) &= \begin{cases} 
\text{true} & \text{if } \mathcal{A}[a_1](s) = \mathcal{A}[a_2](s) \\
\text{false} & \text{if } \mathcal{A}[a_1](s) \neq \mathcal{A}[a_2](s)
\end{cases}
\end{align*}
\]
Denotational Semantics of \( \text{IMP}^- \) (3)

\[
\begin{align*}
\mathcal{C}[\text{skip}](s) &= s \\
\mathcal{C}[\text{if } b \text{ then } c_1 \text{ else } c_2](s) &= \begin{cases} 
\mathcal{C}[c_1](s) & \text{if } \mathcal{B}[b](s) = \text{true} \\
\mathcal{C}[c_2](s) & \text{if } \mathcal{B}[b](s) = \text{false}
\end{cases} \\
\mathcal{C}[c_1; c_2] &= \mathcal{C}[c_2] \circ \mathcal{C}[c_1]
\end{align*}
\]
A denotational semantics for while-loops has to fulfil

\[ C[\text{while } b \text{ do } c] = C[\text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}] \]
A denotational semantics for while-loops has to fulfil

\[ C[\text{while } b \text{ do } c] = C[\text{if } b \text{ then}(c; \text{while } b \text{ do } c) \text{ else skip}] \]

Does such a function always exist? Is it unique?
Semantics for While-Loops

A denotational semantics for while-loops has to fulfil

\[ C[\text{while } b \text{ do } c] = C[\text{if } b \text{ then}(c; \text{while } b \text{ do } c) \text{ else } \text{skip}] \]

Does such a function always exist? Is it unique?

To give a denotational for while loops we have improve our mathematics and reason about:

- partial orders of information
- limits
- continuous functions
- unique fixed points
Denotational semantics does not work for all programming languages (yet).
Denotational semantics does not work for all programming languages (yet).

I try to find a denotational semantics for a language including
- concurrency
- dynamic names
Denotational semantics does not work for all programming languages (yet).

I try to find a denotational semantics for a language including

- concurrency
- dynamic names

Overall theme: Improve the mathematical basis of programming.