Chapter 8

Application: Programming Language Semantics
Programming Language Semantics

Software Foundations Book

- PM intro
- PM bigstep semantics
- Demo MyWhile.thy
- PM smallstep semantics
- Denotational semantics
- Axiomatic semantics: Hoare Logic.
- Demo MyHoare.thy
Why Formal Semantics?

- Programming language design
  - Formal verification of language properties
  - Reveal ambiguities
  - Support for standardization
- Implementation of programming languages
  - Compilers
  - Interpreters
  - Portability
- Reasoning about programs
  - Formal verification of program properties
  - Extended static checking
Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of T or a subtype of T

- Very important question for language designers

- Example:
  If String is a subtype of Object, should String[] be a subtype of Object[]?
Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of T or a subtype of T

- Very important question for language designers

- Example:
  If String is a subtype of Object, should String[] be a subtype of Object[]?

```java
void m(Object[] oa) {
    oa[0] = new Integer(5);
}
String[] sa = new String[10];
m(sa);
String s = sa[0];
```
Language Definition

- State of a program execution
- Transformation of states

- Dynamic Semantics

- Type rules
- Name resolution

- Static Semantics

- Syntax rules, defined by grammar

- Syntax
Compilation and Execution

- Scanning, Parsing
- Abstract Syntax Tree
- Semantic Analysis, Type Checking
- Annotated Abstract Syntax Tree
- Execution

Peter Müller—Semantics of Programming Languages, SS04 p. 13
Three Kinds of Semantics

► Operational semantics
  - Describes execution on an **abstract machine**
  - Describes how the effect is achieved

► Denotational semantics
  - Programs are regarded as **functions** in a mathematical domain
  - Describes **only the effect**, not how it is obtained

► Axiomatic semantics
  - **Specifies properties** of the effect of executing a program are expressed
  - Some aspects of the computation may be **ignored**

Peter Müller—Semantics of Programming Languages, SS04 - p.14
Operational Semantics

```
y := 1;
while not(x=1) do ( y := x*y; x := x-1 )
```

- “First we assign 1 to \( y \), then we test whether \( x \) is 1 or not. If it is then we stop and otherwise we update \( y \) to be the product of \( x \) and the previous value of \( y \) and then we decrement \( x \) by 1. Now we test whether the new value of \( x \) is 1 or not. . . ”

- Two kinds of operational semantics
  - Natural Semantics
  - Structural Operational Semantics
Denotational Semantics

```plaintext
y := 1;
while not(x=1) do ( y := x*y; x := x-1 )
```

- “The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state”

- Two kinds of denotational semantics
  - Direct Style Semantics
  - Continuation Style Semantics
Axiomatic Semantics

\[
\begin{align*}
y &:= 1; \\
\text{while } \neg(x=1) \text{ do } (y := x*y; x := x-1)
\end{align*}
\]

- "If \( x = n \) holds before the program is executed then \( y = n! \) will hold when the execution terminates (if it terminates)"

- Two kinds of axiomatic semantics
  - Partial correctness
  - Total correctness
Abstraction

Concrete language implementation

Operational semantics

Denotational semantics

Axiomatic semantics

Abstract description
Selection Criteria

- Constructs of the programming language
  - Imperative
  - Functional
  - Concurrent
  - Object-oriented
  - Non-deterministic
  - Etc.

- Application of the semantics
  - Understanding the language
  - Program verification
  - Prototyping
  - Compiler construction
  - Program analysis
  - Etc.
The Language IMP

- **Expressions**
  - Boolean and arithmetic expressions
  - No side-effects in expressions

- **Variables**
  - All variables range over integers
  - All variables are initialized
  - No global variables

- **IMP does not include**
  - Heap allocation and pointers
  - Variable declarations
  - Procedures
  - Concurrency
Syntax of IMP: Characters and Tokens

Characters

Letter = 'A' ... 'z' | 'a' ... 'z'
Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Tokens

Ident = Letter { Letter | Digit }
Integer = Digit { Digit }
Var = Ident
Syntax of IMP: Expressions

Arithmetic expressions

\[
\begin{align*}
A\text{exp} & = A\text{exp} \ O p \ A\text{exp} \ | \ V a r \ | \ I n t e g e r \\
O p & = '+' \ | \ '-' \ | \ '*' \ | \ '/' \ | \ 'mod'
\end{align*}
\]

Boolean expressions

\[
\begin{align*}
B\text{exp} & = B\text{exp} \ 'or' \ B\text{exp} \ | B\text{exp} \ 'and' \ B\text{exp} \\
& \ | \ 'not' \ B\text{exp} \ | \ A\text{exp} \ R e l O p \ A\text{exp} \\
R e l O p & = '=' \ | \ '#' \ | \ '<' \ | \ '<=' \ | \ '>' \ | \ '>='
\end{align*}
\]
Syntax of IMP: Statements

\[ \text{Stm} = '\text{skip}' \]

\[ \text{Var} ':=' \text{Aexp} \]

\[ \text{Stm} ';' \text{Stm} \]

\[ '\text{if} \ ' \text{Bexp} '\text{then} \ ' \text{Stm} '\text{else} \ ' \text{Stm} '\text{end}' \]

\[ '\text{while} \ ' \text{Bexp} '\text{do} \ ' \text{Stm} '\text{end}' \]
Notation

Meta-variables (written in *italic* font)

- $x, y, z$ for variables (Var)
- $e, e', e_1, e_2$ for arithmetic expressions (Aexp)
- $b, b_1, b_2$ for boolean expressions (Bexp)
- $s, s', s_1, s_2$ for statements (Stm)

Keywords are written in *typewriter* font

Peter Müller—Semantics of Programming Languages, SS04 p.34
Syntax of IMP: Example

res := 1;
while n > 1 do
    res := res * n;
    n := n - 1
end
Semantic Categories

Syntactic category: Integer  Semantic category: $\text{Val} = \mathbb{Z}$

$$
\begin{align*}
101 & \rightarrow 5 \\
101 & \rightarrow 101
\end{align*}
$$

- Semantic functions map elements of syntactic categories to elements of semantic categories
- To define the semantics of IMP, we need semantic functions for
  - Arithmetic expressions (syntactic category Aexp)
  - Boolean expressions (syntactic category Bexp)
  - Statements (syntactic category Stm)
States

- The meaning of an expression depends on the values bound to the variables that occur in it
- A state associates a value to each variable

State : Var → Val

- We represent a state $\sigma$ as a finite function

$$\sigma = \{ x_1 \rightarrow v_1, x_2 \rightarrow v_2, \ldots, x_n \rightarrow v_n \}$$

where $x_1, x_2, \ldots, x_n$ are different elements of Var and $v_1, v_2, \ldots, v_n$ are elements of Val.
Semantics of Arithmetic Expressions

The semantic function

\[ A : \text{Aexp} \rightarrow \text{State} \rightarrow \text{Val} \]

maps an arithmetic expression \( e \) and a state \( \sigma \) to a value \( A[e] \sigma \)

\[
A[x] \sigma = \sigma(x) \\
A[i] \sigma = i \quad \text{for} \ i \in \mathbb{Z} \\
A[e_1 \ op \ e_2] \sigma = A[e_1] \sigma \overline{\text{op}} A[e_2] \sigma \quad \text{for} \ \text{op} \in \text{Op}
\]

\( \overline{\text{op}} \) is the operation \( \text{Val} \times \text{Val} \rightarrow \text{Val} \) corresponding to \( \text{op} \)

Peter Müller—Semantics of Programming Languages, SS04 = p.39
Semantics of Boolean Expressions

The semantic function

\[ B : \text{Bexp} \rightarrow \text{State} \rightarrow \text{Bool} \]

maps a boolean expression \( b \) and a state \( \sigma \) to a truth value \( B[b]\sigma \)

\[
B[e_1 \text{ op } e_2]\sigma = \begin{cases} 
  \text{tt} & \text{if } A[e_1]\sigma \text{ op } A[e_2]\sigma \\
  \text{ff} & \text{otherwise}
\end{cases}
\]

\( \text{op} \in \text{RelOp} \) and \( \text{op} \) is the relation \( \text{Val} \times \text{Val} \) corresponding to \( \text{op} \)
Boolean Expressions (cont’d)

\[
\begin{align*}
B[b_1 \text{ or } b_2]_\sigma &= \begin{cases} 
tt & \text{if } B[b_1]_\sigma = \tt \text{ or } B[b_2]_\sigma = \tt \\
ff & \text{otherwise}
\end{cases} \\
B[b_1 \text{ and } b_2]_\sigma &= \begin{cases} 
\tt & \text{if } B[b_1]_\sigma = \tt \text{ and } B[b_2]_\sigma = \tt \\
ff & \text{otherwise}
\end{cases} \\
B[\text{not } b]_\sigma &= \begin{cases} 
\tt & \text{if } B[b]_\sigma = \ff \\
ff & \text{otherwise}
\end{cases}
\end{align*}
\]
Operational Semantics of Statements

- Evaluation of an expression in a state yields a value
  \[ x + 2 \times y \]
  \[ \mathcal{A} : \text{Aexp} \rightarrow \text{State} \rightarrow \text{Val} \]

- Execution of a statement modifies the state
  \[ x := 2 \times y \]

- Operational semantics describe how the state is modified during the execution of a statement
Big-Step and Small-Step Semantics

- Big-step semantics describe how the overall results of the executions are obtained
  - Natural semantics

- Small-step semantics describe how the individual steps of the computations take place
  - Structural operational semantics
  - Abstract state machines
Transition Systems

A transition system is a tuple \((\Gamma, T, \triangleright)\)
- \(\Gamma\): a set of configurations
- \(T\): a set of terminal configurations, \(T \subseteq \Gamma\)
- \(\triangleright\): a transition relation, \(\triangleright \subseteq \Gamma \times \Gamma\)

Example: Finite automaton

\[
\Gamma = \{\langle w, S \rangle \mid w \in \{a, b, c\}^*, S \in \{1, 2, 3, 4\}\}
\]
\[
T = \{\langle \epsilon, S \rangle \mid S \in \{1, 2, 3, 4\}\}
\]
\[
\triangleright = \{(\langle aw, 1 \rangle \rightarrow \langle w, 2 \rangle), (\langle aw, 1 \rangle \rightarrow \langle w, 3 \rangle),
\quad (\langle bw, 2 \rangle \rightarrow \langle w, 4 \rangle), (\langle cw, 3 \rangle \rightarrow \langle w, 4 \rangle)\}\}
\]
Transitions in Natural Semantics

- Two types of configurations for operational semantics
  1. \( \langle s, \sigma \rangle \), which represents that the statement \( s \) is to be executed in state \( \sigma \)
  2. \( \sigma \), which represents a terminal state

- The transition relation \( \rightarrow \) describes how executions take place
  - Typical transition: \( \langle s, \sigma \rangle \rightarrow \sigma' \)
  - Example: \( \langle \text{skip}, \sigma \rangle \rightarrow \sigma \)

\[
\begin{align*}
\Gamma &= \{ \langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State} \} \cup \text{State} \\
T &= \text{State} \\
\rightarrow &\subseteq \{ \langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State} \} \times \text{State}
\end{align*}
\]
Rules

Transition relation is specified by rules

\[
\frac{\varphi_1, \ldots, \varphi_n}{\psi} \quad \text{if } Condition
\]

where \( \varphi_1, \ldots, \varphi_n \) and \( \psi \) are transitions

Meaning of the rule

If \( Condition \) and \( \varphi_1, \ldots, \varphi_n \) then \( \psi \)

Terminology

- \( \varphi_1, \ldots, \varphi_n \) are called premises
- \( \psi \) is called conclusion
- A rule without premises is called axiom

Peter Müller—Semantics of Programming Languages, SS04 p.62
Notation

- Updating States: $\sigma[y \mapsto v]$ is the function that
  - overrides the association of $y$ in $\sigma$ by $y \mapsto v$ or
  - adds the new association $y \mapsto v$ to $\sigma$

\[
(\sigma[y \mapsto v])(x) = \begin{cases} 
  v & \text{if } x = y \\
  \sigma(x) & \text{if } x \neq y
\end{cases}
\]
Natural Semantics of IMP

- **skip** does not modify the state
  \[\langle \text{skip}, \sigma \rangle \rightarrow \sigma\]

- **x := e** assigns the value of \(e\) to variable \(x\)
  \[\langle x := e, \sigma \rangle \rightarrow \sigma[x \leftarrow A[e]\sigma]\]

- **Sequential composition** \(s_1 ; s_2\)
  - First, \(s_1\) is executed in state \(\sigma\), leading to \(\sigma'\)
  - Then \(s_2\) is executed in state \(\sigma'\)
  \[\langle s_1, \sigma \rangle \rightarrow \sigma', \langle s_2, \sigma' \rangle \rightarrow \sigma''\]
  \[\langle s_1 ; s_2, \sigma \rangle \rightarrow \sigma''\]
Natural Semantics of IMP (cont’d)

- Conditional statement \(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}\)
  - If \(b\) holds, \(s_1\) is executed
  - If \(b\) does not hold, \(s_2\) is executed

\[
\begin{align*}
\Delta \sigma \rightarrow \sigma' & \quad \text{if } B[b]\sigma = \text{tt} \\
\Delta \sigma \rightarrow \sigma' & \quad \text{if } B[b]\sigma = \text{ff}
\end{align*}
\]
Natural Semantics of IMP (cont’d)

- Loop statement \( \text{while } b \text{ do } s \text{ end} \)
  - If \( b \) holds, \( s \) is executed once, leading to state \( \sigma' \)
  - Then the whole while-statement is executed again \( \sigma' \)

\[
\langle s, \sigma \rangle \rightarrow \sigma', \langle \text{while } b \text{ do } s \text{ end}, \sigma' \rangle \rightarrow \sigma'' \quad \text{if } B[b] \sigma = \text{tt}
\]

- If \( b \) does not hold, the while-statement does not modify the state

\[
\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow \sigma \quad \text{if } B[b] \sigma = \text{ff}
\]
Rule Instantiations

- Rules are actually *rule schemes*
  - Meta-variables stand for arbitrary variables, expressions, statements, states, etc.
  - To apply rules, they have to be *instantiated* by selecting particular variables, expressions, statements, states, etc.

- Assignment rule *scheme*
  \[ \langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto A[e] \sigma] \]

- Assignment rule *instance*
  \[ \langle v := v + 1, \{ v \mapsto 3 \} \rangle \rightarrow \{ v \mapsto 4 \} \]
Derivations: Example

What is the final state if statement

\[
\begin{align*}
z & := x; \quad x := y; \quad y := z
\end{align*}
\]

is executed in state \( \{ x \mapsto 5, y \mapsto 7, z \mapsto 0 \} \)
(abbreviated by \([5, 7, 0]\))?

\[
\begin{align*}
\langle z := x, [5, 7, 0] \rangle & \rightarrow [5, 7, 5], \\
\langle x := y, [5, 7, 5] \rangle & \rightarrow [7, 7, 5] \\
\langle z := x; \quad x := y, [5, 7, 0] \rangle & \rightarrow [7, 7, 5] \\
\langle y := z, [7, 7, 5] \rangle & \rightarrow [7, 5, 5] \\
\langle z := x; \quad x := y; \quad y := z, [5, 7, 0] \rangle & \rightarrow [7, 5, 5]
\end{align*}
\]
Derivation Trees

- Rule instances can be combined to derive a transition $\langle s, \sigma \rangle \rightarrow \sigma'$

- The result is a **derivation tree**
  - The root is the transition $\langle s, \sigma \rangle \rightarrow \sigma'$
  - The leaves are axiom instances
  - The internal nodes are conclusions of rule instances and have the corresponding premises as immediate children

- The conditions of all instantiated rules must be satisfied

- There can be several derivations for one transition (non-deterministic semantics)
Termination

- The execution of a statement \( s \) in state \( \sigma \)
  - **terminates** iff there is a state \( \sigma' \) such that \( \langle s, \sigma \rangle \rightarrow \sigma' \)
  - **loops** iff there is no state \( \sigma' \) such that \( \langle s, \sigma \rangle \rightarrow \sigma' \)

- A statement \( s \)
  - **always terminates** if the execution in a state \( \sigma \) terminates for all choices of \( \sigma \)
  - **always loops** if the execution in a state \( \sigma \) loops for all choices of \( \sigma \)
Semantic Equivalence

Definition

Two statements $s_1$ and $s_2$ are **semantically equivalent** (denoted by $s_1 \equiv s_2$) if the following property holds for all states $\sigma, \sigma'$:

$$\langle s_1, \sigma \rangle \rightarrow \sigma' \iff \langle s_2, \sigma \rangle \rightarrow \sigma'$$

Example

```
while b do s end \equiv
if b then s; while b do s end
```
Structural Operational Semantics

- The emphasis is on the individual steps of the execution
  - Execution of assignments
  - Execution of tests
- Describing small steps of the execution allows one to express the order of execution of individual steps
  - Interleaving computations
  - Evaluation order for expressions (not shown in the course)
- Describing always the next small step allows one to express properties of looping programs
Transitions in SOS

- The configurations are the same as for natural semantics.
- The transition relation $\rightarrow_1$ can have two forms:
  - $\langle s, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle$: the execution of $s$ from $\sigma$ is not completed and the remaining computation is expressed by the intermediate configuration $\langle s', \sigma' \rangle$.
  - $\langle s, \sigma \rangle \rightarrow_1 \sigma'$: the execution of $s$ from $\sigma$ has terminated and the final state is $\sigma'$.
- A transition $\langle s, \sigma \rangle \rightarrow_1 \gamma$ describes the first step of the execution of $s$ from $\sigma$. 
Transition System

\[ \Gamma = \{ \langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State} \} \cup \text{State} \]
\[ T = \text{State} \]
\[ \rightarrow_1 \subseteq \{ \langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State} \} \times \Gamma \]

- We say that \( \langle s, \sigma \rangle \) is **stuck** if there is no \( \gamma \) such that \( \langle s, \sigma \rangle \rightarrow_1 \gamma \)
SOS of IMP

- skip does not modify the state
  \[ \langle \text{skip}, \sigma \rangle \rightarrow_1 \sigma \]

- \( x := e \) assigns the value of \( e \) to variable \( x \)
  \[ \langle x := e, \sigma \rangle \rightarrow_1 \sigma[x \mapsto A[e] \sigma] \]

- skip and assignment require only one step

- Rules are analogous to natural semantics
  \[ \langle \text{skip}, \sigma \rangle \rightarrow \sigma \]
  \[ \langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto A[e] \sigma] \]

Peter Müller — Semantics of Programming Languages, SS04 – p.103
SOS of IMP: Sequential Composition

- Sequential composition $s_1; s_2$
- First step of executing $s_1; s_2$ is the first step of executing $s_1$
- $s_1$ is executed in one step

\[
\begin{align*}
\langle s_1, \sigma \rangle &\rightarrow_1 \sigma' \\
\langle s_1; s_2, \sigma \rangle &\rightarrow_1 \langle s_2, \sigma' \rangle
\end{align*}
\]

- $s_1$ is executed in several steps

\[
\begin{align*}
\langle s_1, \sigma \rangle &\rightarrow_1 \langle s'_1, \sigma' \rangle \\
\langle s_1; s_2, \sigma \rangle &\rightarrow_1 \langle s'_1; s_2, \sigma' \rangle
\end{align*}
\]
SOS of IMP: Conditional Statement

- The first step of executing if $b$ then $s_1$ else $s_2$ end is to determine the outcome of the test and thereby which branch to select.

\[
\langle \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}, \sigma \rangle \rightarrow_1 \langle s_1, \sigma \rangle \quad \text{if } B[b] \sigma = \text{tt} \\
\langle \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}, \sigma \rangle \rightarrow_1 \langle s_2, \sigma \rangle \quad \text{if } B[b] \sigma = \text{ff}
\]
Alternative for Conditional Statement

The first step of executing \( \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end} \) is the first step of the branch determined by the outcome of the test

\[
\begin{align*}
\langle s_1, \sigma \rangle \rightarrow_1 \sigma' & \quad \text{if } B[b] \sigma = tt \\
\langle \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}, \sigma \rangle \rightarrow_1 \sigma' & \quad \text{if } B[b] \sigma = tt
\end{align*}
\]

and two similar rules for \( B[b] \sigma = ff \)

- Alternatives are equivalent for IMP
- Choice is important for languages with parallel execution

Peter Müller—Semantics of Programming Languages, SS04 – p.106
SOS of IMP: Loop Statement

- The first step is to unrole the loop

\[
\begin{align*}
\langle \text{while } b \text{ do } s \text{ end, } \sigma \rangle & \rightarrow_1 \\
\langle \text{if } b \text{ then } s ; \text{while } b \text{ do } s \text{ end else skip end, } \sigma \rangle
\end{align*}
\]

- Recall that \texttt{while } \texttt{b do } \texttt{s end} and \texttt{if } \texttt{b then } \texttt{s ; while } \texttt{b do } \texttt{s end else skip end} are semantically equivalent in the natural semantics
Alternatives for Loop Statement

- The first step is to decide the outcome of the test and thereby whether to unrole the body of the loop or to terminate.

\[
\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow_1 \langle s; \text{while } b \text{ do } s \text{ end}, \sigma \rangle \quad \text{if } B[b] \sigma = tt
\]

\[
\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow_1 \sigma \quad \text{if } B[b] \sigma = ff
\]

- Or combine with the alternative semantics of the conditional statement

- Alternatives are equivalent for IMP
Derivation Sequences

- A **derivation sequence** of a statement $s$ starting in state $\sigma$ is a sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$, where
  - $\gamma_0 = \langle s, \sigma \rangle$
  - $\gamma_i \rightarrow_1 \gamma_{i+1}$ for $0 \leq i$

- A derivation sequence is either **finite** or **infinite**
  - Finite derivation sequences end with a configuration that is either a terminal configuration or a stuck configuration

- **Notation**
  - $\gamma_0 \rightarrow_1^i \gamma_i$ indicates that there are $i$ steps in the execution from $\gamma_0$ to $\gamma_i$
  - $\gamma_0 \rightarrow_1^* \gamma_i$ indicates that there is a **finite number of steps** in the execution from $\gamma_0$ to $\gamma_i$
  - $\gamma_0 \rightarrow_1^* \gamma_i$ and $\gamma_0 \rightarrow_1^* \gamma_i$ need **not** be derivation sequences

Peter Müller—Semantics of Programming Languages, SS04 – p.109
Derivation Sequences: Example

What is the final state if statement

\[
z := x; \quad x := y; \quad y := z
\]

is executed in state \{ x \mapsto 5, y \mapsto 7, z \mapsto 0 \}?
Derivation Trees

- Derivation trees explain why transitions take place
- For the first step

\[
\langle z := x; \ x := y; \ y := z, \sigma \rangle \rightarrow_1 \langle x := y; \ y := z, \sigma[z \mapsto 5] \rangle
\]

the derivation tree is

\[
\begin{align*}
\langle z := x, \sigma \rangle & \rightarrow_1 \sigma[z \mapsto 5] \\
\langle z := x; \ x := y, \sigma \rangle & \rightarrow_1 \langle x := y, \sigma[z \mapsto 5] \rangle \\
\langle z := x; \ x := y; \ y := z, \sigma \rangle & \rightarrow_1 \langle x := y; \ y := z, \sigma[z \mapsto 5] \rangle
\end{align*}
\]

- \( z := x; \ ( x := y; \ y := z ) \) would lead to a simpler tree with only one rule application
Derivation Sequences and Trees

- **Natural (big-step) semantics**
  - The execution of a statement (sequence) is described by one big transition
  - The big transition can be seen as trivial derivation sequence with exactly one transition
  - The derivation tree explains why this transition takes place

- **Structural operational (small-step) semantics**
  - The execution of a statement (sequence) is described by one or more transitions
  - Derivation sequences are important
  - Derivation trees justify each individual step in a derivation sequence
Termination

- The execution of a statement $s$ in state $\sigma$
  - **terminates** iff there is a finite derivation sequence starting with $\langle s, \sigma \rangle$
  - **loops** iff there is an infinite derivation sequence starting with $\langle s, \sigma \rangle$

- The execution of a statement $s$ in state $\sigma$
  - **terminates successfully** if $\langle s, \sigma \rangle \rightarrow^*_1 \sigma'$
  - In IMP, an execution terminates successfully iff it terminates (no stuck configurations)
Comparison: Summary

Natural Semantics

- Local variable declarations and procedures can be modeled easily
- No distinction between abortion and loopsing
- Non-determinism suppresses looping (if possible)
- Parallelism cannot be modeled

Structural Operational Semantics

- Local variable declarations and procedures require modeling the execution stack
- Distinction between abortion and looping
- Non-determinism does not suppress looping
- Parallelism can be modeled
Motivation

- Operational semantics is at a rather low abstraction level
  - Some arbitrariness in choice of rules (e.g., size of steps)
  - Syntax involved in description of behavior

- Semantic equivalence in natural semantics

\[ \langle s_1, \sigma \rangle \rightarrow \sigma' \iff \langle s_2, \sigma \rangle \rightarrow \sigma' \]

- Idea
  - We can describe the behavior on an abstract level if we are only interested in equivalence
  - We specify only the partial function on states
Approach

- Denotational semantics describes the effect of a computation

- A semantic function is defined for each syntactic construct
  - maps syntactic construct to a mathematical object, often a function
  - the mathematical object describes the effect of executing the syntactic construct
Compositionality

In denotational semantics, semantic functions are defined \textit{compositionally}

There is a semantic clause for each of the basis elements of the syntactic category

For each method of constructing a composite element (in the syntactic category) there is a semantic clause defined in terms of the \textit{semantic function applied to the immediate constituents} of the composite element
Examples

The semantic functions $\mathcal{A} : \text{Aexp} \rightarrow \text{State} \rightarrow \text{Val}$ and $\mathcal{B} : \text{Bexp} \rightarrow \text{State} \rightarrow \text{Bool}$ are denotational definitions.

\[
\begin{align*}
\mathcal{A}[x] \sigma &= \sigma(x) \\
\mathcal{A}[i] \sigma &= i \quad \text{for } i \in \mathbb{Z} \\
\mathcal{A}[e_1 \text{ op } e_2] \sigma &= \mathcal{A}[e_1] \sigma \overline{\text{op}} \mathcal{A}[e_2] \sigma \quad \text{for } \text{op} \in \text{Op}
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}[e_1 \text{ op } e_2] \sigma &= \begin{cases} 
\text{tt} & \text{if } \mathcal{A}[e_1] \sigma \overline{\text{op}} \mathcal{A}[e_2] \sigma \\
\text{ff} & \text{otherwise}
\end{cases}
\end{align*}
\]
Counterexamples

- The semantic functions $S_{NS}$ and $S_{SOS}$ are not denotational definitions because they are not defined compositionally

\[
S_{NS} : \text{Stm} \to (\text{State} \leftrightarrow \text{State})
\]

\[
S_{NS}[s]_\sigma = \begin{cases} 
\sigma' & \text{if } \langle s, \sigma \rangle \rightarrow \sigma' \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

\[
S_{SOS} : \text{Stm} \to (\text{State} \leftrightarrow \text{State})
\]

\[
S_{SOS}[s]_\sigma = \begin{cases} 
\sigma' & \text{if } \langle s, \sigma \rangle \rightarrow^* \sigma' \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Semantic Functions

- The effect of executing a statement is described by the partial function $S_{DS}$

$$S_{DS} : \text{Stm} \rightarrow (\text{State} \leftrightarrow \text{State})$$

- Partiality is needed to model non-termination

- The effects of evaluating expressions is defined by the functions $A$ and $B$
Direct Style Semantics of IMP

- \texttt{skip} does not modify the state

\[ S_{DS}[\texttt{skip}] = id \]

\[ id : \text{State} \rightarrow \text{State} \]

\[ id(\sigma) = \sigma \]

- \texttt{x:=e} assigns the value of \( e \) to variable \( x \)

\[ S_{DS}[x:=e] \sigma = \sigma[x \mapsto A[e] \sigma] \]
Direct Style Semantics of IMP (cont’d)

- Sequential composition $s_1 ; s_2$

$$S_{DS}[s_1 ; s_2] = S_{DS}[s_2] \circ S_{DS}[s_1]$$

- Function composition $\circ$ is defined in a **strict** way
  - If one of the functions is undefined on the given argument then the composition is undefined

$$ (f \circ g)\sigma = \begin{cases} 
  f(g(\sigma)) & \text{if } g(\sigma) \neq \text{undefined} \\
  \text{undefined} & \text{otherwise}
\end{cases} $$

Peter Müller—Semantics of Programming Languages, SS04 – p.202
Direct Style Semantics of IMP (cont’d)

- **Conditional statement** \( \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end} \)

\[
S_{DS}[\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}] = \text{cond}(\mathcal{B}[b], S_{DS}[s_1], S_{DS}[s_2])
\]

- **The function** \( \text{cond} \)
  - takes the semantic functions for the condition and the two statements
  - when supplied with a state selects the second or third argument depending on the first

\[
\text{cond} : (\text{State } \rightarrow \text{Bool}) \times (\text{State } \leftrightarrow \text{State}) \times (\text{State } \leftrightarrow \text{State}) \rightarrow (\text{State } \leftrightarrow \text{State})
\]
Definition of \textit{cond}

\[
\text{cond} : \ (\text{State} \rightarrow \text{Bool}) \times (\text{State} \leftrightarrow \text{State}) \times (\text{State} \leftrightarrow \text{State}) \\
\rightarrow (\text{State} \leftrightarrow \text{State})
\]

\[
\text{cond}(b, f, g)\sigma = \begin{cases} 
  f(\sigma) & \text{if } b(\sigma) = \text{tt} \\
  g(\sigma) & \text{if } b(\sigma) = \text{ff} \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

Peter Müller—Semantics of Programming Languages, SS04 – p.204
Semantics of Loop: Observations

- Defining the semantics of `while` is difficult.
- The semantics of `while b do s end` must be equal to `if b then s; while b do s end else skip end`.
- This requirement yields:

\[
S_{DS}\left[\text{while } b \text{ do } s \text{ end}\right] = \text{cond}(B[b], S_{DS}\left[\text{while } b \text{ do } s \text{ end}\right] \circ S_{DS}[s], id)
\]

- We cannot use this equation as a definition because it is not compositional.
Functionals and Fixed Points

\[ S_{DS}[\text{while } b \text{ do } s \text{ end}] = \]
\[ \text{cond}(B[b], S_{DS}[\text{while } b \text{ do } s \text{ end}] \circ S_{DS}[s], id) \]

- The above equation has the form \( g = F(g) \)
  - \( g = S_{DS}[\text{while } b \text{ do } s \text{ end}] \)
  - \( F(g) = \text{cond}(B[b], g \circ S_{DS}[s], id) \)

- \( F \) is a **functional** (a function from functions to functions)

- \( S_{DS}[\text{while } b \text{ do } s \text{ end}] \) is a **fixed point** of the functional \( F \)
Direct Style Semantics of IMP: Loops

- Loop statement `while b do s end`

\[
S_{DS}[\text{while } b \text{ do } s \text{ end}] = \text{FIX } F
\]

where \( F(g) = \text{cond}(\mathcal{B}[b], g \circ S_{DS}[s], \text{id}) \)

- We write `FIX F` to denote the fixed point of the functional \( F \):

\[
\text{FIX : } ((\text{State } \leftrightarrow \text{State}) \rightarrow (\text{State } \leftrightarrow \text{State})) \\
\rightarrow (\text{State } \leftrightarrow \text{State})
\]

- This definition of \( S_{DS}[\text{while } b \text{ do } s \text{ end}] \) is compositional

Peter Müller—Semantics of Programming Languages, SS04 – p.208
Example

Consider the statement

```
while x ≠ 0 do skip end
```

The functional for this loop is defined by

\[
F'(g)\sigma = \text{cond}(B[x#0], g \circ S_{DS}[\text{skip}], id)\sigma
\]

\[
= \text{cond}(B[x#0], g \circ id, id)\sigma
\]

\[
= \text{cond}(B[x#0], g, id)\sigma
\]

\[
= \begin{cases} 
g(\sigma) & \text{if } \sigma(x) \neq 0 \\
\sigma & \text{if } \sigma(x) = 0
\end{cases}
\]
Example (cont’d)

- The function

\[ g_1(\sigma) = \begin{cases} 
  \text{undefined} & \text{if } \sigma(x) \neq 0 \\
  \sigma & \text{if } \sigma(x) = 0 
\end{cases} \]

is a fixed point of \( F' \)

- The function \( g_2(\sigma) = \text{undefined} \) is not a fixed point for \( F' \)
Well-Definedness

\[
S_{DS}[\text{while } b \text{ do } s \text{ end}] = \text{FIX } F
\]
where \( F(g) = \text{cond}(B[b], g \circ S_{DS}[s], id) \)

The function \( S_{DS}[\text{while } b \text{ do } s \text{ end}] \) is well-defined if \( \text{FIX } F \) defines a **unique fixed point** for the functional \( F \)

- There are functionals that have more than one fixed point
- There are functionals that have no fixed point at all
Examples

- \( F' \) from the previous example has more than one fixed point

\[
F'(g)\sigma = \begin{cases} 
    g(\sigma) & \text{if } \sigma(x) \neq 0 \\
    \sigma & \text{otherwise}
\end{cases}
\]

- Every function \( g' : \text{State} \rightarrow \text{State} \) with \( g'(\sigma) = \sigma \) if \( \sigma(x) = 0 \) is a fixed point for \( F' \)

- The functional \( F_1 \) has no fixed point if \( g_1 \neq g_2 \)

\[
F_1(g) = \begin{cases} 
    g_1 & \text{if } g = g_2 \\
    g_2 & \text{otherwise}
\end{cases}
\]
Hoare Logic

Hoare axioms and rules for simple while languages

▶ { P } skip { P }
▶ { P[x/e] } x := e { P }
▶ { P } c1 { R }, { R } c2 { Q } => { P } c1;c2 { Q }
▶ { P ∧ b } c1 { Q }, { P ∧ !b } c2 { Q } =>
  { P } if b then c1 else c2 { Q }
▶ { INV ∧ b } c { INV } => { INV } while b do c { INV ∧ !b }
▶ P → P’, { P’ } c { Q’ }, Q’ → Q => { P } c { Q }
▶ Semantics of the Hoare Logic:
▶ { P } c { Q } == ( ALL s. ( P(s) ∧ s -c-> t ) → P(t) )
Hoare Logic

Example

\{ 0 \leq x \} 
\begin{align*} 
c & := 0 ; 
sq & := 1 ; 
\textbf{WHILE } \sq \leq x \textbf{ DO } (* \text{INV}=(c\cdot c \leq x \& \sq=(c+1)\cdot(c+1))*) 
& \quad c \ := \ c + 1 ; 
& \quad \sq \ := \ \sq + (2\cdot c + 1) ; 
\end{align*} 
\begin{align*} 
\{ \ c\cdot c \leq x \ & \& x < (c+1)\cdot(c+1) \} 
\end{align*}

Demo: MyHoare.thy