Chapter 8

Application: Programming Language Semantics

Programming Language Semantics

Software Foundations Book

- Material: http://sct.ethz.ch/teaching/ss2004/sps/lecture.html
- 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

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller — Semantics of Programming Languages, SS04 – p.7

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[]`?

```
void m(Object[] oa) {
  oa[0]=new Integer(5);
}
```

```
String[] sa=new String[10];
m(sa);
String s = sa[0];
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

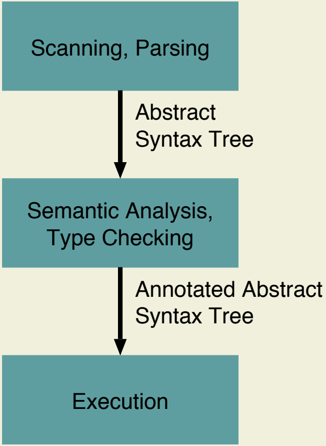
Language Definition

Dynamic Semantics	▸ State of a program execution ▸ Transformation of states
Static Semantics	▸ Type rules ▸ Name resolution
Syntax	▸ Syntax rules, defined by grammar

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.12

Compilation and Execution



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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**

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.15

Denotational Semantics

```
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

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.16

Axiomatic Semantics

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

- "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

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.17

Abstraction

Concrete language implementation

Operational semantics

Denotational semantics

Axiomatic semantics

Abstract descrption

Peter Müller—Semantics of Programming Languages, SS04 – p.18

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.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.19

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

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.30

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

Aexp	= Aexp Op Aexp | Var | Integer
Op	= '+' | '-' | '*' | '/' | 'mod'

Boolean expressions

Bexp	= Bexp 'or' Bexp | Bexp 'and' Bexp
	| 'not' Bexp | Aexp RelOp Aexp
RelOp	= '=' | '#' | '<' | '<=' | '>' | '>='

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Syntax of IMP: Statemens

```
Stm = 'skip'
    | Var ':=' Aexp
    | Stm ';' Stm
    | 'if' Bexp 'then' Stm 'else' Stm 'end'
    | 'while' Bexp 'do' Stm 'end'
```

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.33

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

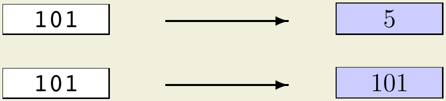
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Syntax of IMP: Example

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

Semantic Categories

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



- 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

`x+1` ⟶ ??

- 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

$$\text{State} : \text{Var} \longrightarrow \text{Val}$$

- We represent a state σ as a finite function

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

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

$$\mathcal{A} : \mathsf{Aexp} \rightarrow \mathsf{State} \rightarrow \mathsf{Val}$$

maps an arithmetic expression e and a state σ to a value $\mathcal{A}[\![e]\!]\sigma$

$$\begin{array}{lll} \mathcal{A}[\![x]\!]\sigma & = \sigma(x) & \\ \mathcal{A}[\![i]\!]\sigma & = i & \text{for } i \in \mathbb{Z} \\ \mathcal{A}[\![e_1 \; op \; e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma \; \overline{op} \; \mathcal{A}[\![e_2]\!]\sigma & \text{for } op \in \mathsf{Op} \end{array}$$

$\overline{op}$ is the operation $\mathsf{Val} \times \mathsf{Val} \rightarrow \mathsf{Val}$ corresponding to op

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semantics of Boolean Expressions

The semantic function

$$\mathcal{B} : \text{Bexp} \longrightarrow \text{State} \longrightarrow \text{Bool}$$

maps a boolean expression b and a state σ to a truth value $\mathcal{B}[\![b]\!]\sigma$

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

$op \in$ RelOp and $\overline{op}$ is the relation Val $\times$ Val corresponding to op

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.40

Boolean Expressions (cont'd)

$$\mathcal{B}[\![b_1 \texttt{ or } b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ or } \mathcal{B}[\![b_2]\!]\sigma = tt \\ ff & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![b_1 \texttt{ and } b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ and } \mathcal{B}[\![b_2]\!]\sigma = tt \\ ff & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![\texttt{not } b]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b]\!]\sigma = ff \\ ff & \text{otherwise} \end{cases}$$

Operational Semantics of Statements

- Evaluation of an expression in a state yields a value

```
x + 2 * y
```

$\mathcal{A}$: Aexp $\longrightarrow$ State $\longrightarrow$ Val

- Execution of a statement modifies the state

```
x := 2 * y
```

- Operational semantics describe **how** the state is modified during the execution of a statement

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.57

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

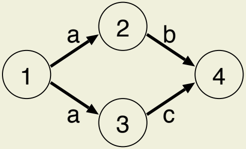
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Transition Systems

- A transition system is a tuple $(\Gamma, T, \triangleright)$
  - Γ: 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

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



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 σ
  2. σ, which represents a terminal state
- The transition relation $\rightarrow$ describes how executions take place
  - Typical transition: $\langle s, \sigma \rangle \rightarrow \sigma'$
  - Example: $\langle \texttt{skip}, \sigma \rangle \rightarrow \sigma$

$$\Gamma = \{\langle s, \sigma \rangle \mid s \in \mathsf{Stm}, \sigma \in \mathsf{State}\} \cup \mathsf{State}$$
$$T = \mathsf{State}$$
$$\rightarrow \subseteq \{\langle s, \sigma \rangle \mid s \in \mathsf{Stm}, \sigma \in \mathsf{State}\} \times \mathsf{State}$$

Rules

- Transition relation is specified by rules

$$\frac{\varphi_1, \ldots, \varphi_n}{\psi} \quad \text{if } \textit{Condition}$$

where $\varphi_1, \ldots, \varphi_n$ and ψ are transitions

- Meaning of the rule

If *Condition* and $\varphi_1, \ldots, \varphi_n$ then ψ

- Terminology
  - $\varphi_1, \ldots, \varphi_n$ are called **premises**
  - ψ is called **conclusion**
  - A rule without premises is called **axiom**

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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 σ by $y \mapsto v$ or
  - adds the new association $y \mapsto v$ to σ

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

Natural Semantics of IMP

- `skip` does not modify the state

$$\overline{\langle \texttt{skip}, \sigma \rangle \rightarrow \sigma}$$

- x `:=` e assigns the value of e to variable e

$$\overline{\langle x \texttt{:=} e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]}$$

- Sequential composition s_1 `;` s_2
  - First, s_1 is executed in state σ, leading to σ'
  - Then s_2 is executed in state σ'

$$\frac{\langle s_1, \sigma \rangle \rightarrow \sigma', \langle s_2, \sigma' \rangle \rightarrow \sigma''}{\langle s_1 \texttt{;} s_2, \sigma \rangle \rightarrow \sigma''}$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Natural Semantics of IMP (cont'd)

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

$$\frac{\langle s_1, \sigma \rangle \rightarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma \rangle \rightarrow \sigma'} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\frac{\langle s_2, \sigma \rangle \rightarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma \rangle \rightarrow \sigma'} \quad \text{if} \mathcal{B}[\![b]\!]\sigma = ff$$

Natural Semantics of IMP (cont'd)

- Loop statement $\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}$
  - If b holds, s is executed once, leading to state σ'
  - Then the whole while-statement is executed again σ'

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

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

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \rightarrow \sigma} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = ff$$

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**

$$\langle x\,\texttt{:=}e, \sigma\rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

- Assignment rule **instance**

$$\langle \texttt{v:=v+1}, \{\texttt{v} \mapsto 3\}\rangle \rightarrow \{\texttt{v} \mapsto 4\}$$

Derivations: Example

- What is the final state if statement

  `z:=x; x:=y; y:=z`

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

$$\dfrac{\dfrac{\langle \mathtt{z:=x}, [5,7,0]\rangle \rightarrow [5,7,5], \langle \mathtt{x:=y}, [5,7,5]\rangle \rightarrow [7,7,5]}{\langle \mathtt{z:=x;\ x:=y}, [5,7,0]\rangle \rightarrow [7,7,5]}, \quad \langle \mathtt{y:=z}, [7,7,5]\rangle \rightarrow [7,5,5]}{\langle \mathtt{z:=x;\ x:=y;\ y:=z}, [5,7,0]\rangle \rightarrow [7,5,5]}$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.68

Derivation Trees

- Rule instances can be combined to derive a transition $\langle s, \sigma \rangle \to \sigma'$
- The result is a **derivation tree**
  - The root is the transition $\langle s, \sigma \rangle \to \sigma'$
  - 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 σ
  - **terminates** iff there is a state σ' such that $\langle s, \sigma \rangle \rightarrow \sigma'$
  - **loops** iff there is no state σ' such that $\langle s, \sigma \rangle \rightarrow \sigma'$
- A statement s
  - **always terminates** if the execution in a state σ terminates for all choices of σ
  - **always loops** if the execution in a state σ loops for all choices of σ

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 σ, σ':
  $$\langle s_1, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle s_2, \sigma \rangle \rightarrow \sigma'$$

- 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 σ 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 σ **has terminated** and the final state is σ'
- A transition $\langle s, \sigma \rangle \rightarrow_1 \gamma$ describes the **first step** of the execution of s from σ

Transition System

$$
\begin{aligned}
\Gamma &= \{\langle s,\sigma\rangle \mid s \in \mathsf{Stm}, \sigma \in \mathsf{State}\} \cup \mathsf{State} \\
T &= \mathsf{State} \\
\rightarrow_1 &\subseteq \{\langle s,\sigma\rangle \mid s \in \mathsf{Stm}, \sigma \in \mathsf{State}\} \times \Gamma
\end{aligned}
$$

- We say that $\langle s,\sigma\rangle$ is **stuck** if there is no γ such that $\langle s,\sigma\rangle \rightarrow_1 \gamma$

SOS of IMP

- `skip` does not modify the state

$$\langle \mathtt{skip}, \sigma \rangle \rightarrow_1 \sigma$$

- x:=e assigns the value of e to variable x

$$\langle x \mathtt{:=} e, \sigma \rangle \rightarrow_1 \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

- `skip` and assignment require only one step
- Rules are analogous to natural semantics

$$\langle \mathtt{skip}, \sigma \rangle \rightarrow \sigma$$

$$\langle x \mathtt{:=} e, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SOS of IMP: Sequential Composition

- Sequential composition $s_1 \,\mathbf{;}\, s_2$
- First step of executing $s_1 \,\mathbf{;}\, s_2$ is the first step of executing s_1
- s_1 is executed in one step

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \sigma'}{\langle s_1 \,\mathbf{;}\, s_2, \sigma \rangle \rightarrow_1 \langle s_2, \sigma' \rangle}$$

- s_1 is executed in several steps

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \langle s_1', \sigma' \rangle}{\langle s_1 \,\mathbf{;}\, s_2, \sigma \rangle \rightarrow_1 \langle s_1' \,\mathbf{;}\, s_2, \sigma' \rangle}$$

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

$$\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow_1 \langle s_1, \sigma\rangle \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow_1 \langle s_2, \sigma\rangle \quad \text{if} \mathcal{B}[\![b]\!]\sigma = ff$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.105

Alternative for Conditional Statement

- The first step of executing $\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$ is the first step of the branch determined by the outcome of the test

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow_1 \sigma'} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \langle s_1', \sigma'\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \rightarrow_1 \langle s_1', \sigma'\rangle} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

and two similar rules for $\mathcal{B}[\![b]\!]\sigma = ff$

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.106

SOS of IMP: Loop Statement

- The first step is to unrole the loop

$$\langle \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \rightarrow_1$$
$$\langle \texttt{if}\ b\ \texttt{then}\ s\texttt{;while}\ b\ \texttt{do}\ s\ \texttt{end}\ \texttt{else}\ \texttt{skip}\ \texttt{end}, \sigma\rangle$$

- Recall that $\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}$ and $\texttt{if}\ b\ \texttt{then}\ s\texttt{;while}\ b\ \texttt{do}\ s\ \texttt{end}\ \texttt{else}\ \texttt{skip}\ \texttt{end}$ 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

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

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

- Or combine with the alternative semantics of the conditional statement
- Alternatives are equivalent for IMP

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.108

Derivation Sequences

- A **derivation sequence** of a statement s starting in state σ 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 γ_0 to γ_i
  - $\gamma_0 \rightarrow_1^* \gamma_i$ indicates that there is a **finite number of steps** in the execution from γ_0 to γ_i
  - $\gamma_0 \rightarrow_1^i \gamma_i$ and $\gamma_0 \rightarrow_1^* \gamma_i$ need **not** be derivation sequences

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.109

Derivation Sequences: Example

- What is the final state if statement

$$\boxed{\texttt{z:=x; x:=y; y:=z}}$$

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

$$\langle \texttt{z:=x; x:=y; y:=z}, \{\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0\}\rangle$$
$$\rightarrow_1 \langle \texttt{x:=y; y:=z}, \{\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5\}\rangle$$
$$\rightarrow_1 \langle \texttt{y:=z}, \{\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5\}\rangle$$
$$\rightarrow_1 \{\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 5\}$$

Derivation Trees

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

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

  the derivation tree is

$$\cfrac{\cfrac{\langle \texttt{z:=x}, \sigma\rangle \rightarrow_1 \sigma[\texttt{z} \mapsto 5]}{\langle \texttt{z:=x; x:=y}, \sigma\rangle \rightarrow_1 \langle \texttt{x:=y}, \sigma[\texttt{z} \mapsto 5]\rangle}}{\langle \texttt{z:=x; x:=y; y:=z}, \sigma\rangle \rightarrow_1 \langle \texttt{x:=y; y:=z}, \sigma[\texttt{z} \mapsto 5]\rangle}$$

- `z:=x; ( x:=y; y:=z )` would lead to a simpler tree with only one rule application

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.111

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 σ
  - **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 σ
  - **terminates successfully** if $\langle s, \sigma \rangle \to_1^* \sigma'$
  - In IMP, an execution terminates successfully iff it terminates (no stuck configurations)

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Comparison: Summary

Natural Semantics

- Local variable declarations and procedures can be modeled easily
- No distinction between abortion and looping
- 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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.134

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

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

- 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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Compositionality

- In denotational semantics, semantic functions are defined **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 **semantic function applied to the immediate constituents** of the composite element

Examples

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

$$
\begin{array}{lll}
\mathcal{A}[\![x]\!]\sigma & = \sigma(x) & \\
\mathcal{A}[\![i]\!]\sigma & = i & \text{for } i \in \mathbb{Z} \\
\mathcal{A}[\![e_1\ op\ e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma\ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma & \text{for } op \in \text{ Op}
\end{array}
$$

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

Counterexamples

- The semantic functions $\mathcal{S}_{NS}$ and $\mathcal{S}_{SOS}$ are not denotational definitions because they are not defined compositionally

$$\mathcal{S}_{NS} : \text{Stm} \rightarrow (\text{State} \hookrightarrow \text{State})$$

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

$$\mathcal{S}_{SOS} : \text{Stm} \rightarrow (\text{State} \hookrightarrow \text{State})$$

$$\mathcal{S}_{SOS}[\![s]\!]\sigma = \begin{cases} \sigma' & \text{if } \langle s, \sigma \rangle \rightarrow_1^* \sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Semantic Functions

- The effect of executing a statement is described by the partial function $\mathcal{S}_{DS}$

$$\mathcal{S}_{DS} : \text{Stm} \longrightarrow (\text{State} \hookrightarrow \text{State})$$

- Partiality is needed to model non-termination
- The effects of evaluating expressions is defined by the functions $\mathcal{A}$ and $\mathcal{B}$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.200

Direct Style Semantics of IMP

- `skip` does not modify the state

$$\mathcal{S}_{DS}[\![\texttt{skip}]\!] = id$$

$$id : \text{State} \rightarrow \text{State}$$
$$id(\sigma) = \sigma$$

- x`:=`e assigns the value of e to variable x

$$\mathcal{S}_{DS}[\![x \texttt{:=} e]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.201

Direct Style Semantics of IMP (cont'd)

- Sequential composition $s_1 \,;\, s_2$

$$\mathcal{S}_{DS}[\![s_1 \,;\, s_2]\!] = \mathcal{S}_{DS}[\![s_2]\!] \circ \mathcal{S}_{DS}[\![s_1]\!]$$

- 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

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

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.202

Direct Style Semantics of IMP (cont'd)

- Conditional statement `if` b `then` s_1 `else` s_2 `end`

$$\mathcal{S}_{DS}[\![\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{DS}[\![s_1]\!], \mathcal{S}_{DS}[\![s_2]\!])$$

- The function $cond$
  - 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

$$cond : (\text{State} \rightarrow \text{Bool}) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Definition of $cond$

$$cond : (\text{State} \to \text{Bool}) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State}) \to (\text{State} \hookrightarrow \text{State})$$

$$cond(b, f, g)\sigma = \begin{cases} f(\sigma) & \text{if } b(\sigma) = tt \\ & \text{and } f(\sigma) \neq \text{undefined} \\ g(\sigma) & \text{if } b(\sigma) = \mathit{ff} \\ & \text{and } g(\sigma) \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

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:

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = \\ cond(\mathcal{B}[\![b]\!], \mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] \circ \mathcal{S}_{DS}[\![s]\!], id)$$

- We cannot use this equation as a definition because it is not compositional

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.205

Functionals and Fixed Points

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] \circ \mathcal{S}_{DS}[\![s]\!], id)$$

- The above equation has the form $g = F(g)$
  - $g = \mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$
  - $F(g) = cond(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{DS}[\![s]\!], id)$
- F is a **functional** (a function from functions to functions)
- $\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$ is a **fixed point** of the functional F

Direct Style Semantics of IMP: Loops

- Loop statement `while` b `do` s `end`

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = FIX\ F$$
$$\text{where } F(g) = cond(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{DS}[\![s]\!], id)$$

- We write $FIX\ F$ to denote the fixed point of the functional F:

$$FIX : ((\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})) \rightarrow (\text{State} \hookrightarrow \text{State})$$

- This defintion of $\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$ is compositional

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example

- Consider the statement

```
while x # 0 do skip end
```

- The functional for this loop is defined by

$$
\begin{aligned}
F'(g)\sigma &= cond(\mathcal{B}[\![\mathtt{x\#0}]\!], g \circ \mathcal{S}_{DS}[\![\mathtt{skip}]\!], id)\sigma \\
&= cond(\mathcal{B}[\![\mathtt{x\#0}]\!], g \circ id, id)\sigma \\
&= cond(\mathcal{B}[\![\mathtt{x\#0}]\!], g, id)\sigma \\
&= \begin{cases} g(\sigma) & \text{if } \sigma(x) \neq 0 \\ \sigma & \text{if } \sigma(x) = 0 \end{cases}
\end{aligned}
$$

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

$$\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!] = \mathit{FIX}\ F$$
$$\text{where } F(g) = \mathit{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{DS}[\![s]\!], \mathit{id})$$

- The function $\mathcal{S}_{DS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!]$ is well-defined if $FIX\,F$ 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 $g' : \text{State} \hookrightarrow \text{State}$ with $g'(\sigma) = \sigma$ if $\sigma(x) = 0$ is a fixed point for F'

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

$$F_1(g) = \begin{cases} g_1 & \text{if } g = g_2 \\ g_2 & \text{otherwise} \end{cases}$$

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Peter Müller—Semantics of Programming Languages, SS04 – p.212

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 <= x }
  c  := 0 ;
  sq := 1 ;
  WHILE sq <= x DO (*INV=(c*c <= x&sq=(c+1)*(c+1))*)
     c := c + 1 ;
     sq :=  sq + (2*c + 1);
{ c*c <= x  &  x < (c+1)*(c+1) }
```

Demo: MyHoare.thy