

Chapter 7

Programming Language Semantics

Section 7.1

Introduction

Overview of Chapter

7. Programming Language Semantics

7.1 Introduction

7.2 Big-step semantics

Basic concepts of big-step semantics

Formalization of big-step semantics

7.3 Small-step semantics

Small-step semantics of IMP

Proving properties of the semantics

Extensions of IMP

7.4 Denotational semantics

Motivation

Why studying language semantics?

- Understanding the details and construction of programming and modeling languages
- Foundation for language processing tools (compilers, optimizers, interpreters,...)
- Verifying type systems
- Development of new language abstractions and software concepts
- Reasoning about software

Material

Literature

- Glynn Winskel:
The Formal Semantics of Programming Languages: An Introduction
- Benjamin C. Pierce et al.:
Software Foundations (www.cis.upenn.edu/~bcpierce/sf/)

Acknowledgement

Thanks to Prof. Peter Müller for the slides.

General aspects

Degree of formalization:

- Informal semantics in language reports
- Formalization to support proof tools and as quality control

Goals here:

- Learn distinction between operational and denotational semantics
- Learn about the formalization of operational semantics
- Understand the relationship between programming language semantics and transition systems

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

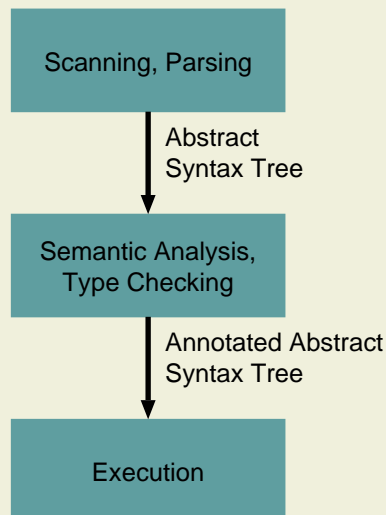
- ▶ Type safety:
In each execution state, a variable of type T holds a value of T or a subtype of T
- ▶ Very important question for language designers
- ▶ Example:
If String is a subtype of Object, should `String[]` be a subtype of `Object[]`?

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

Language Definition

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

Compilation and Execution



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**

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

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

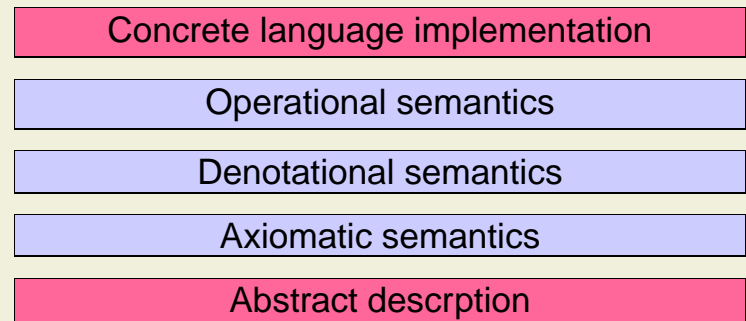
- ▶ “The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state”
- ▶ Two kinds of denotational semantics
 - Direct Style Semantics
 - Continuation Style Semantics

Axiomatic Semantics

```
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

Abstraction



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

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

Boolean expressions

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

Syntax of IMP: Statemens

```

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

```

Notation

Meta-variables (written in *italic* font)

```

x, y, z      for variables (Var)
e, e', e1, e2 for arithmetic expressions (Aexp)
b, b1, b2   for boolean expressions (Bexp)
s, s', s1, s2 for statements (Stm)

```

Keywords are written in `typewriter` font

Syntax of IMP: Example

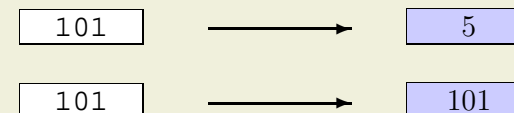
```

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

```

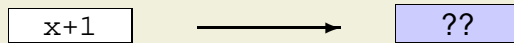
Semantic Categories

Syntactic category: Integer Semantic category: 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



- ▶ 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

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

- ▶ We represent a state σ as a finite function

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

where x_1, x_2, \dots, x_n are different elements of Var and v_1, v_2, \dots, v_n are elements of Val .

Semantics of Arithmetic Expressions

The semantic function

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

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

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

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

Semantics of Boolean Expressions

The semantic function

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

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

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

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

Boolean Expressions (cont'd)

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

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

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

Operational Semantics of Statements

- ▶ Evaluation of an expression in a state yields a value

$$\boxed{x + 2 * y}$$

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

- ▶ Execution of a statement modifies the state

$$\boxed{x := 2 * y}$$

- ▶ 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 $(\Gamma, T, \triangleright)$
 - Γ : a set of **configurations**
 - T : a set of **terminal configurations**, $T \subseteq \Gamma$
 - \triangleright : a **transition relation**, $\triangleright \subseteq \Gamma \times \Gamma$

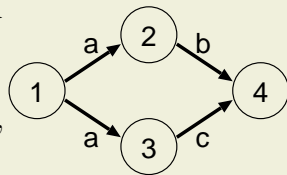
- ▶ Example: Finite automaton

$$\Gamma = \{\langle w, S \rangle \mid w \in \{a, b, c\}^*, S \in \{1, 2, 3, 4\}\}$$

$$T = \{\langle \epsilon, S \rangle \mid S \in \{1, 2, 3, 4\}\}$$

$$\triangleright = \{(\langle aw, 1 \rangle \rightarrow \langle w, 2 \rangle), (\langle aw, 1 \rangle \rightarrow \langle w, 3 \rangle),$$

$$(\langle bw, 2 \rangle \rightarrow \langle w, 4 \rangle), (\langle cw, 3 \rangle \rightarrow \langle w, 4 \rangle)\}$$



Section 7.2

Big-step semantics

Big-step semantics

Introductory remarks:

- Big-step semantics relates *prestates* of statement executions to *poststates*
- Often called *natural semantics*, because it abstracts from intermediate states

Overview:

- Basic concepts of big-step semantics
- Formalization of big-step semantics in Isabelle/HOL

Subsection 7.2.1

Basic concepts of big-step semantics

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: $\langle \text{skip}, \sigma \rangle \rightarrow \sigma$

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

Rules

- ▶ Transition relation is specified by rules

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

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

- ▶ Meaning of the rule

$$\text{If } \textit{Condition} \text{ and } \varphi_1, \dots, \varphi_n \text{ then } \psi$$

- ▶ Terminology

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

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

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

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

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

- ▶ 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'

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

Natural Semantics of IMP (cont'd)

- ▶ Conditional statement `if b then s_1 else s_2 end`
 - If b holds, s_1 is executed
 - If b does not hold, s_2 is executed

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

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

Natural Semantics of IMP (cont'd)

- ▶ Loop statement `while b do s end`

- If b holds, s is executed once, leading to state σ'
- Then the whole while-statement is executed again σ'

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

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

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

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**

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

- ▶ Assignment rule **instance**

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

Derivations: Example

- ▶ What is the final state if statement

$$z := x; x := y; y := z$$

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

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

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 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
```

Subsection 7.2.2

Formalization of big-step semantics

Big-step semantics in Isabelle/HOL

Approach

- Formalize the abstract syntax of the language by a recursive datatype
- Formalize the big-step semantics as an inductive predicate

Arithmetic expressions

```
datatype var = V nat ("v_" 90)
```

```
datatype aexp = Plus aexp aexp (infixr "+." 30)
              | Minus aexp aexp (infixr "-." 30)
              | Mult aexp aexp (infixr "*." 50)
              | Div aexp aexp (infixr "/." 50)
              | Mod aexp aexp (infixr "%." 50)
              | Var var ("_" 90)
              | Const int ("_" 90)
```

Boolean expressions

```
datatype bexp = Or   bexp bexp   (infixr ".|." 20)
              | And  bexp bexp   (infixr ".&." 20)
              | Not  bexp        (".!_" 80)
              | Eq   aexp aexp   (infixr "=". 10)
              | Less aexp aexp   (infixr "<." 10)
              | Leq  aexp aexp   (infixr "<=" 10)
```

commands/statements

```
datatype com =
  Skip          ("SKIP")
  | Assign var aexp (":==" [60, 60] 20)
  | Semi com com  (";" _" [60, 60] 10)
  | Cond bexp com com ("IF _ THEN _ ELSE _ END" 60)
  | While bexp com  ("WHILE _ DO _ END" 60)
```

Evaluation of arithmetic expressions

```
type_synonym state = "var ⇒ int"

primrec aeval :: "aexp ⇒ state ⇒ int" where
  "aeval (Plus la ra) s = ((aeval la s) + (aeval ra s))"
| "aeval (Minus la ra) s = ((aeval la s) - (aeval ra s))"
| "aeval (Mult la ra) s = ((aeval la s) * (aeval ra s))"
| "aeval (Div la ra) s = ((aeval la s) div (aeval ra s))"
| "aeval (Mod la ra) s = ((aeval la s) mod (aeval ra s))"
| "aeval (Var v) s = (s v)"
| "aeval (Const i) s = i"
```

Evaluation of boolean expressions

```
primrec beval :: "bexp ⇒ state ⇒ bool" where
  "beval (Or lb rb) s = ((beval lb s) ∨ (beval rb s))"
| "beval (And lb rb) s = ((beval lb s) ∧ (beval rb s))"
| "beval (Not be) s = (¬ (beval be s))"
| "beval (Eq la ra) s = ((aeval la s) = (aeval ra s))"
| "beval (Less la ra) s = ((aeval la s) < (aeval ra s))"
| "beval (Leq la ra) s = ((aeval la s) ≤ (aeval ra s))"
```

Operational semantics

```

inductive exec :: "state ⇒ com ⇒ state ⇒ bool"
  ("_/ -_→/ _" [50,0,50] 50) where
  Skip: "s -SKIP→ s"
| Assign: "s -(v:= ae)→ s(v:= aeval ae s)"
| Semi: "[[ s0 -c1→ s1; s1 -c2→ s2 ]]
  ⇒ s0 -c1;c2→ s2"
| IfT: "[[ beval b s ; s -c1→ t ]]
  ⇒ s -IF b THEN c1 ELSE c2 END→ t"
| IfF: "[[ ¬(beval b s); s -c2→ t ]]
  ⇒ s -IF b THEN c1 ELSE c2 END→ t"
| WhileF: "¬(beval b s) ⇒ s -WHILE b DO c END→ s"
| WhileT: "[[ beval b s; s-c→t; t -WHILE b DO c END→ u ]]
  ⇒ s -WHILE b DO c END→ u"

```

Some properties

```

lemma [iff]: "(s -c;d→ u) = (∃t. s -c→ t ∧ t -d→ u)"
lemma [iff]: "(s -IF be THEN c ELSE d END→ t) =
  (s -if beval be s then c else d → t)"
lemma unfold_while:
  "(s -WHILE b DO c END→ u) =
  (s -IF b THEN c; WHILE b DO c END ELSE SKIP END→ u)"
lemma while_rule:
  "[[ s -WHILE b DO c END→ t; P s;
  ∀s s'. P s ∧ (beval b s) ∧ s -c→ s' → P s' ]]
  ⇒ P t ∧ ¬(beval b t)"

```

Formalization of semantics and verification

Remarks

- Inductive definition of the “semantics judgement” leads to a semantic predicate satisfying the least fixpoint of the semantical rules
- The operational semantics can be directly used for program verification (Why do we need a programming logic? (cf. Chapter 8))

Small-step semantics

Section 7.3

Overview

- 7.3.1 Small-step semantics of IMP
- 7.3.2 Proving properties of the semantics
- 7.3.3 Extensions of IMP

Subsection 7.3.1

Small-step semantics of IMP

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

$$\Gamma = \{\langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \cup \text{State}$$

$$T = \text{State}$$

$$\rightarrow_1 \subseteq \{\langle s, \sigma \rangle \mid s \in \text{Stm}, \sigma \in \text{State}\} \times \Gamma$$

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

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

- ▶ skip and assignment require only one step

- ▶ Rules are analogous to natural semantics

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

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

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

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

- ▶ s_1 is executed in several steps

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \langle s'_1, \sigma' \rangle}{\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \langle s'_1 ; s_2, \sigma' \rangle}$$

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

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

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

Alternative for Conditional Statement

- ▶ The first step of executing `if b then s1 else s2 end` is the first step of the branch determined by the outcome of the test

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

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \langle s'_1, \sigma' \rangle}{\langle \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}, \sigma \rangle \rightarrow_1 \langle s'_1, \sigma' \rangle} \quad \text{if } \mathcal{B}[[b]]\sigma = tt$$

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

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

SOS of IMP: Loop Statement

- ▶ The first step is to unrole the loop

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

- ▶ Recall that `while b do s end` and `if b then s; while b do 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

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

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

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

Derivation Sequences

- ▶ A **derivation sequence** of a statement s starting in state σ is a sequence $\gamma_0, \gamma_1, \gamma_2, \dots$, 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

Derivation Sequences: Example

- ▶ What is the final state if statement

$$z := x; \quad x := y; \quad y := z$$

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

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

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

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

$$\rightarrow_1 \{x \mapsto 7, y \mapsto 5, z \mapsto 5\}$$

Derivation Trees

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

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

the derivation tree is

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

- ▶ $z := x; \quad (\quad x := y; \quad y := z \quad)$ 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 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 $\langle s, \sigma \rangle \rightarrow_1^* \sigma'$
 - In IMP, an execution terminates successfully iff it terminates (no stuck configurations)

Subsection 7.3.2

Proving properties of the semantics

Using Induction on Derivations

- ▶ The induction step is often done by inspecting either
 - the structure of the syntactic element or
 - the derivation tree validating the first transition of the derivation sequence

▶ Lemma

$$\langle s_1 ; s_2, \sigma \rangle \rightarrow_1^k \sigma'' \Rightarrow \\ \exists \sigma', k_1, k_2 : \langle s_1, \sigma \rangle \rightarrow_1^{k_1} \sigma' \wedge \langle s_2, \sigma' \rangle \rightarrow_1^{k_2} \sigma'' \wedge \\ k_1 + k_2 = k$$

Induction on Derivations

Induction on the length of derivation sequences

1. **Induction base:** Prove that the property holds for all derivation sequences of length 0
2. **Induction step:** Prove that the property holds for all other derivation sequences:
 - ▶ **Induction hypothesis:** Assume that the property holds for all derivation sequences of length at most k
 - ▶ Prove that it also holds for derivation sequences of length $k + 1$

Induction on the length of derivation sequences is an application of strong mathematical induction.

Proof

- ▶ Proof by induction on k , that is, by induction on the length of the derivation sequence for $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1^k \sigma''$
- ▶ Induction base: $k = 0$: There is no derivation sequence of length 0 for $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1^k \sigma''$
- ▶ Induction step
 - We assume that the lemma holds for $k \leq m$
 - We prove that the lemma holds for $m + 1$
 - The derivation sequence $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1^{m+1} \sigma''$ can be written as $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \gamma \rightarrow_1^m \sigma''$ for some configuration γ

Induction Step

- ▶ $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \gamma \rightarrow_1^m \sigma''$
- ▶ Consider the two rules that could lead to the transition $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \gamma$

- ▶ Case 1

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

- ▶ Case 2

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \langle s'_1, \sigma' \rangle}{\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \langle s'_1 ; s_2, \sigma' \rangle}$$

Induction Step: Case 1

- ▶ From $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \gamma \rightarrow_1^m \sigma''$ and $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \langle s_2, \sigma' \rangle$ we conclude $\langle s_2, \sigma' \rangle \rightarrow_1^m \sigma''$
- ▶ The required result follows by choosing $k_1 = 1$ and $k_2 = m$

Induction Step: Case 2

- ▶ From $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \gamma \rightarrow_1^m \sigma''$ and $\langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \langle s'_1 ; s_2, \sigma' \rangle$ we conclude $\langle s'_1 ; s_2, \sigma' \rangle \rightarrow_1^m \sigma''$
- ▶ By applying the induction hypothesis, we get $\exists \sigma_0, l_1, l_2 : \langle s'_1, \sigma' \rangle \rightarrow_1^{l_1} \sigma_0 \wedge \langle s_2, \sigma_0 \rangle \rightarrow_1^{l_2} \sigma'' \wedge l_1 + l_2 = m$
- ▶ From $\langle s_1, \sigma \rangle \rightarrow_1 \langle s'_1, \sigma' \rangle$ and $\langle s'_1, \sigma' \rangle \rightarrow_1^{l_1} \sigma_0$ we get $\langle s_1, \sigma \rangle \rightarrow_1^{l_1+1} \sigma_0$
- ▶ By $\langle s_2, \sigma_0 \rangle \rightarrow_1^{l_2} \sigma''$ and $(l_1 + 1) + l_2 = m + 1$ we have proved the required result

Semantic Equivalence

Two statements s_1 and s_2 are **semantically equivalent** if for all states σ :

- ▶ $\langle s_1, \sigma \rangle \rightarrow_1^* \gamma$ iff $\langle s_2, \sigma \rangle \rightarrow_1^* \gamma$, whenever γ is a configuration that is either stuck or terminal, and
- ▶ there is an infinite derivation sequence starting in $\langle s_1, \sigma \rangle$ iff there is one starting in $\langle s_2, \sigma \rangle$

Note: In the first case, the length of the two derivation sequences may be different

Determinism

Lemma: The structural operational semantics of IMP is deterministic. That is, for all s, σ, γ , and γ' we have that

$$\langle s, \sigma \rangle \rightarrow_1 \gamma \wedge \langle s, \sigma \rangle \rightarrow_1 \gamma' \Rightarrow \gamma = \gamma'$$

- ▶ The proof runs by induction on the shape of the derivation tree for the transition $\langle s, \sigma \rangle \rightarrow_1 \gamma$

Corollary: There is exactly one derivation sequence starting in configuration $\langle s, \sigma \rangle$

- ▶ The proof runs by induction on the length of the derivation sequence

Subsection 7.3.3

Extensions of IMP

Extensions of IMP

- Local variable declarations
- Statement “abort”
- Non-determinism
- Parallelism

Local Variable Declarations

- ▶ Local variable declaration `var $x := e$ in s end`
- ▶ The small steps are
 1. Assign e to x
 2. Execute s
 3. Restore the initial value of x
(necessary if x exists in the enclosing scope)
- ▶ Problem: There is no history of states that could be used to restore the value of x
- ▶ Idea: Represent states as execution stacks

Modelling Execution Stacks

- ▶ We model execution stacks by providing a mapping $\text{Var} \rightarrow \text{Val}$ for each scope

State : *stack of*($\text{Var} \rightarrow \text{Val}$)

- ▶ Assignment and lookup have to determine the highest stack element in which a variable is defined
- ▶ Example: $\sigma(x) = 3$

$z \mapsto 4$
$x \mapsto 3$
$x \mapsto 1, y \mapsto 2$

SOS for Variable Declarations

- ▶ The small steps are
 1. Create new scope and assign e to x in this scope
 2. Execute s
 3. Restore the initial value of x using a `return` statement

$$\langle \text{var } x := e \text{ in } s \text{ end}, \sigma \rangle \rightarrow_1$$

$$\langle s ; \text{return}, \text{push}(\{x \mapsto \mathcal{A}[e]\sigma\}, \sigma) \rangle$$

$$\langle \text{return}, \sigma \rangle \rightarrow_1 \text{pop}(\sigma)$$

- ▶ Similar techniques can be used for procedure calls

Abortion

- ▶ Statement `abort` stops the execution of the complete program
- ▶ Abortion is modeled by ensuring that the configurations $\langle \text{abort}, \sigma \rangle$ are **stuck**
- ▶ There is no additional rule for `abort` in the structural operational semantics
- ▶ `abort` and `skip` are not semantically equivalent
 - $\langle \text{abort}, \sigma \rangle$ is the only derivation sequence for `abort` starting in s
 - $\langle \text{skip}, \sigma \rangle \rightarrow_1 \sigma$ is the only derivation sequence for `skip` starting in s

Abortion: Observations

- ▶ `abort` and `while true do skip end` are not semantically equivalent:

$$\langle \text{while true do skip end}, \sigma \rangle \rightarrow_1$$

$$\langle \text{if true then skip ; while true do skip end end}, \sigma \rangle \rightarrow_1$$

$$\langle \text{skip ; while true do skip end} \rangle \rightarrow_1$$

$$\langle \text{while true do skip end}, \sigma \rangle$$

- ▶ In a structural operational semantics,
 - looping is reflected by infinite derivation sequences
 - abnormal termination by finite derivation sequences ending in a stuck configuration

Non-determinism

- ▶ For the statement $s_1 \parallel s_2$ either s_1 or s_2 is non-deterministically chosen to be executed
- ▶ The statement

$$x := 1 \parallel x := 2; x := x + 2$$

could result in a state in which x has the value 1 or 4

- ▶ Rules

$$\langle s_1 \parallel s_2, \sigma \rangle \rightarrow_1 \langle s_1, \sigma \rangle \quad \langle s_1 \parallel s_2, \sigma \rangle \rightarrow_1 \langle s_2, \sigma \rangle$$

Non-determinism: Observations

- ▶ There are two derivation sequences
 - $\langle x := 1 \parallel x := 2; x := x + 2, \sigma \rangle \rightarrow_1^* \sigma[x \mapsto 1]$
 - $\langle x := 1 \parallel x := 2; x := x + 2, \sigma \rangle \rightarrow_1^* \sigma[x \mapsto 4]$
- ▶ There are also two derivation sequences for $\langle \text{while true do skip end} \parallel x := 2; x := x + 2, \sigma \rangle$
 - an finite derivation sequence leading to $\sigma[x \mapsto 4]$
 - an infinite derivation sequence
- ▶ A structural operational semantics can choose the "wrong" branch of a non-deterministic choice
- ▶ In a structural operational semantics **non-determinism does not suppress looping**

Parallelism

- ▶ For the statement $s_1 \text{ par } s_2$ both statements s_1 and s_2 are executed, but execution can be **interleaved**

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \langle s'_1, \sigma' \rangle}{\langle s_1 \text{ par } s_2, \sigma \rangle \rightarrow_1 \langle s'_1 \text{ par } s_2, \sigma' \rangle}$$

$$\frac{\langle s_1, \sigma \rangle \rightarrow_1 \sigma'}{\langle s_1 \text{ par } s_2, \sigma \rangle \rightarrow_1 \langle s_2, \sigma' \rangle}$$

$$\frac{\langle s_2, \sigma \rangle \rightarrow_1 \langle s'_2, \sigma' \rangle}{\langle s_1 \text{ par } s_2, \sigma \rangle \rightarrow_1 \langle s_1 \text{ par } s'_2, \sigma' \rangle}$$

$$\frac{\langle s_2, \sigma \rangle \rightarrow_1 \sigma'}{\langle s_1 \text{ par } s_2, \sigma \rangle \rightarrow_1 \langle s_1, \sigma' \rangle}$$

Example: Interleaving

- ▶ The statement

$$x := 1 \text{ par } x := 2; x := x + 2$$

could result in a state in which x has the value 4, 1, or 3

- Execute $x := 1$, then $x := 2$, and then $x := x + 2$
 - Execute $x := 2$, then $x := x + 2$, and then $x := 1$
 - Execute $x := 2$, then $x := 1$, and then $x := x + 2$
- ▶ In a structural operational semantics we can easily express interleaving of computations

Example: Derivation Sequences

$$\langle x:=1 \text{ par } x:=2; x:=x+2, \sigma \rangle \rightarrow_1 \langle x:=2; x:=x+2, \sigma[x \mapsto 1] \rangle$$

$$\rightarrow_1 \langle x:=x+2, \sigma[x \mapsto 2] \rangle$$

$$\rightarrow_1 \sigma[x \mapsto 4]$$

$$\langle x:=1 \text{ par } x:=2; x:=x+2, \sigma \rangle \rightarrow_1 \langle x:=1 \text{ par } x:=x+2, \sigma[x \mapsto 2] \rangle$$

$$\rightarrow_1 \langle x:=1, \sigma[x \mapsto 4] \rangle$$

$$\rightarrow_1 \sigma[x \mapsto 1]$$

$$\langle x:=1 \text{ par } x:=2; x:=x+2, \sigma \rangle \rightarrow_1 \langle x:=1 \text{ par } x:=x+2, \sigma[x \mapsto 2] \rangle$$

$$\rightarrow_1 \langle x:=x+2, \sigma[x \mapsto 1] \rangle$$

$$\rightarrow_1 \sigma[x \mapsto 3]$$

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

Section 7.4

Denotational semantics

Motivation

Goals of a semantics definition

- Semantics defines *observational* behavior
- Semantics defines an equivalence relation on **programs**:
When are two programs considered to be equal?
- Semantics should also provide semantics of **program parts**:
E.g.: When can a Java class be used for another class?
- Semantics should be defined compositional

Observations

Operational semantics:

- often not sufficiently abstract and non-compositional
- often unclear how to handle program parts


```

package cells;

public interface Val{}

public class Cell {
  private Val vatt;
  public void set(Val v)
  {
    vatt = v;
  }
  public Val get() {
    return vatt;
  }
}

```

```

package cells;

public interface Val {}

public class Cell {
  private Val v1, v2;
  private boolean f;
  public void set(Val v) {
    f = !f ;
    if (f) v1 = v;
    else v2 = v;
  }
  public Val get() {
    return f ? v1 : v2;
  }
  public Val getPrevious() {
    return f ? v2 : v1;
  }
}

```

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 **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 $\mathcal{A} : \text{Aexp} \rightarrow \text{State} \rightarrow \text{Val}$ and $\mathcal{B} : \text{Bexp} \rightarrow \text{State} \rightarrow \text{Bool}$ are denotational definitions

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

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

Counterexamples

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

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

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

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

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

Semantic Functions

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

$$\mathcal{S}_{DS} : \text{Stm} \rightarrow (\text{State} \leftrightarrow \text{State})$$

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

Direct Style Semantics of IMP

- ▶ `skip` does not modify the state

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

$$id : \text{State} \rightarrow \text{State}$$

$$id(\sigma) = \sigma$$

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

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

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

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

Direct Style Semantics of IMP (cont'd)

- ▶ Conditional statement `if b then s1 else s2 end`

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

- ▶ 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

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

Definition of *cond*

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

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

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:

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

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

Functionals and Fixed Points

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

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

Fixed Points: Examples

- ▶ x is a fixed point of function f if $f(x) = x$ holds
- ▶ Consider a function $f : \mathbb{N} \rightarrow \mathbb{N}$
 - $f(x) = x + 1$ does not have a fixed point
 - $f(x) = 0$ has exactly one fixed point, 0
 - $f(x) = x^2$ has two fixed points, 0 and 1
 - $f(x) = x$ has an infinite number of fixed points

Direct Style Semantics of IMP: Loops

- ▶ Loop statement `while b do s end`

$$\mathcal{S}_{DS}[\text{while } b \text{ do } s \text{ end}] = \text{FIX } F$$

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

- ▶ We write $\text{FIX } F$ 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 $\mathcal{S}_{DS}[\text{while } b \text{ do } s \text{ end}]$ is compositional

Example

- ▶ Consider the statement

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

- ▶ The functional for this loop is defined by

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

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

$$\mathcal{S}_{DS}[\text{while } b \text{ do } s \text{ end}] = \text{FIX } F$$

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

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

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

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

Fixed point theory for functions

Strict functions

Let $D_{\perp} =_{\text{def}} D \cup \{\perp\}$ with $\perp \notin D$.

A function $f :: D_{\perp} \Rightarrow D_{\perp}$ is called *strict* iff $f(\perp) = \perp$.

Complete partial order (CPO)

Let $\mathcal{D} =_{\text{def}} D_{\perp} \Rightarrow D_{\perp}$ be the set of all strict functions from D_{\perp} to D_{\perp} and $f \leq g$ if $f(x) = g(x)$ or $f(x) = \perp$.

(\mathcal{D}, \leq) is a *pointed complete partial order* (i.e., every chain in \mathcal{D} has a least upper bound and \mathcal{D} has a least element)

Continuous functions

Let \mathcal{D} be a CPO; a monotonic function $F :: \mathcal{D} \Rightarrow \mathcal{D}$ is *continuous* iff for every chain X in \mathcal{D} :

$$F(\bigvee X) = \bigvee \{F(x) \mid x \in X\}$$

Fixed point theorem

Fixed point theorem

If (\mathcal{D}, \leq) is a pointed CPO and

$F :: \mathcal{D} \Rightarrow \mathcal{D}$ is continuous, then

$\bigvee \{F^i(\perp) \mid i \geq 0\}$ is a fixed point of F

Remarks

Denotational semantics for imperative programs

- are based on fixed point theory for function domains
- allow for compositional definitions and related proof techniques