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
7. Programming Language Semantics 7.1 Introduction

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.

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

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

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

### 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**

---

**Language Definition**

- **Dynamic Semantics**
  - State of a program execution
  - Transformation of states

- **Static Semantics**
  - Type rules
  - Name resolution

- **Syntax**
  - Syntax rules, defined by grammar
7. Programming Language Semantics

7.1 Introduction

Operational Semantics

\[
y := 1; \\
\text{while not}(x=1) \text{ do } ( y := x*y; x := x-1 )
\]

- "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

\[
y := 1; \\
\text{while not}(x=1) \text{ do } ( y := x*y; x := x-1 )
\]

- "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

\[
y := 1; \\
\text{while not}(x=1) \text{ do } ( y := x*y; x := x-1 )
\]

- "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

Concrete language implementation
- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Abstract description
7. Programming Language Semantics

7.1 Introduction

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 = '=' | '#' | '<' | '<=' | '>' | '>='

Peter Müller—Semantics of Programming Languages, SS04 — p.19
© Peter Müller
7. Programming Language Semantics

7.1 Introduction

Syntax of IMP: Statements

\[ \text{Stm} = '\text{skip}' \]
| \text{Var} ':=' \text{Aexp} |
| \text{Stm} ';' \text{Stm} |
| 'if' \text{Bexp} 'then' \text{Stm} 'else' \text{Stm} 'end' |
| 'while' \text{Bexp} 'do' \text{Stm} 'end' |

Notation

Meta-variables (written in italic font)
\( x, y, z \) for variables (Var)
\( e, e_0, e_1, e_2 \) 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

Semantic Categories

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

| 101 | 5 |
| 101 | 101 |

- 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.
- We represent a state $\sigma$ as a finite function $\sigma = \{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n\}$.

Semantics of Arithmetic Expressions

The semantic function $A : Aexp \to State \to Val$ maps an arithmetic expression $e$ and a state $\sigma$ to a value $A[e]_\sigma$.

$$A[x]_\sigma = \sigma(x)$$

$$A[i]_\sigma = i \quad \text{for } i \in \mathbb{Z}$$

$$A[e_1 \ op \ e_2]_\sigma = A[e_1]_\sigma \ op A[e_2]_\sigma \quad \text{for } op \in Op$$

$op$ is the operation $Val \times Val \to Val$ corresponding to $op$.

Boolean Expressions (cont’d)

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

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

$$B[\neg b]_\sigma = \begin{cases} 
  tt & \text{if } B[b]_\sigma = ff \\
  ff & \text{otherwise}
\end{cases}$$

Semantics of Boolean Expressions

The semantic function $B : Bexp \to State \to Bool$ maps a boolean expression $b$ and a state $\sigma$ to a truth value $B[b]_\sigma$.

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

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

Semantics of Arithmetic Expressions

The semantic function $A : Aexp \to State \to Val$ maps an arithmetic expression $e$ and a state $\sigma$ to a value $A[e]_\sigma$.

$$A[x]_\sigma = \sigma(x)$$

$$A[i]_\sigma = i \quad \text{for } i \in \mathbb{Z}$$

$$A[e_1 \ op \ e_2]_\sigma = A[e_1]_\sigma \ op A[e_2]_\sigma \quad \text{for } op \in Op$$

$op$ is the operation $Val \times Val \to Val$ corresponding to $op$. 

Semantics of Boolean Expressions

The semantic function $B : Bexp \to State \to Bool$ maps a boolean expression $b$ and a state $\sigma$ to a truth value $B[b]_\sigma$.

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

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

$$B[\neg b]_\sigma = \begin{cases} 
  tt & \text{if } B[b]_\sigma = ff \\
  ff & \text{otherwise}
\end{cases}$$

Boolean Expressions (cont’d)

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

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

$$B[\neg b]_\sigma = \begin{cases} 
  tt & \text{if } B[b]_\sigma = ff \\
  ff & \text{otherwise}
\end{cases}$$
Operational Semantics of Statements

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

- 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
  \[
  \begin{align*}
  \Gamma &= \{\langle w, S \rangle \mid w \in \{a, b, c\}^*, S \in \{1, 2, 3, 4\}\} \\
  T &= \{\langle \epsilon, S \rangle \mid S \in \{1, 2, 3, 4\}\} \\
  \triangleright &= \{\langle \langle aw, 1 \rangle \rightarrow \langle w, 2 \rangle \rangle, \langle \langle aw, 1 \rangle \rightarrow \langle w, 3 \rangle \rangle, \\
  & \quad \langle \langle bw, 2 \rangle \rightarrow \langle w, 4 \rangle \rangle, \langle \langle cw, 3 \rangle \rightarrow \langle w, 4 \rangle \rangle\}
  \end{align*}
  \]
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

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: \(\langle \text{skip}, \sigma \rangle \rightarrow \sigma\)

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

Rules

- Transition relation is specified by rules
  \[
  \varphi_1, \ldots, \varphi_n \quad \psi\quad \text{if } \text{Condition}
  \]
  where \(\varphi_1, \ldots, \varphi_n\) and \(\psi\) are transitions
- Meaning of the rule
  \[
  \text{If } \text{Condition} \text{ and } \varphi_1, \ldots, \varphi_n \text{ then } \psi
  \]
- Terminology
  - \(\varphi_1, \ldots, \varphi_n\) are called *premises*
  - \(\psi\) 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 $\sigma$ 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 (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

\[ \langle s_1, \sigma \rangle \rightarrow \sigma' \quad \text{if } B[b]\sigma = \text{tt} \]

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

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 $x$

\[ \langle x := e, \sigma \rangle \rightarrow \sigma[x \mapsto A\|e\|\sigma] \]

- Sequential composition $s_1; s_2$
  - First, $s_1$ is executed in state $\sigma$, leading to $\sigma'$
  - Then $s_2$ is executed in state $\sigma'$

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

Natural Semantics of IMP (cont’d)

- Loop statement while $b$ do $s$ 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
  \[ z := x; x := y; y := z \]
  is executed in state \( \{x \mapsto 5, y \mapsto 7, z \mapsto 0\} \) 
  (abbreviated by \([5, 7, 0]\))?

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

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 $\sigma, \sigma'$:

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

Example

```
while b do s end \equiv
if b then s; while b do s end
```
### 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)

### 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))"

### 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"
Operational semantics

inductive exec :: "state ⇒ com ⇒ state ⇒ bool"

| 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)"

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))
7.3 Small-step semantics

Overview

7.3.1 Small-step semantics of IMP

7.3.2 Proving properties of the semantics

7.3.3 Extensions of IMP

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 $\sigma$ 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: 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

\[ \langle s_1, \sigma \rangle \rightarrow_1 \sigma' \]

\[ \langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \langle s_2, \sigma' \rangle \]

- \( s_1 \) is executed in several steps

\[ \langle s_1, \sigma \rangle \rightarrow_1 \langle s'_1, \sigma' \rangle \]

\[ \langle s_1 ; s_2, \sigma \rangle \rightarrow_1 \langle s_1 ; s_2, \sigma' \rangle \]

SOS of IMP: Conditional Statement

- The first step of executing \( \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end} \) is to determine the outcome of the test and thereby which branch to select

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

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

- If `B[b]σ = tt`:
  \[ \langle s₁, σ \rangle \rightarrow_1 s₁' \]
  \[ \langle if b then s₁ else s₂ end, σ \rangle \rightarrow_1 \langle s₁', σ \rangle \]

- If `B[b]σ = ff` and two similar rules for `B[b]σ = 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.

- If `while b do s end, σ`:
  \[ \langle if b then s; while b do s end else skip end, σ \rangle \rightarrow_1 \]

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.

### Derivation Sequences

A derivation sequence of a statement `s` starting in state `σ` is a sequence `γ₀, γ₁, γ₂, ...`, where

- `γ₀ = ⟨s, σ⟩`
- `γᵢ → γᵢ₊₁` for `0 ≤ 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:

- `γ₀ →_i γᵢ` indicates that there are `i` steps in the execution from `γ₀` to `γᵢ`.
- `γ₀ →_i γᵢ` indicates that there is a finite number of steps in the execution from `γ₀` to `γᵢ`.
- `γ₀ →_i γᵢ` and `γ₀ →_i γᵢ` need not be derivation sequences.
Derivation Sequences: Example

▶ What is the final state if statement

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

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

\[
\begin{align*}
(z := x; x := y; y := z, \{ x \mapsto 5, y \mapsto 7, z \mapsto 0 \}) & \quad \rightarrow_1 (x := y; y := z, \{ x \mapsto 5, y \mapsto 7, z \mapsto 5 \}) \\
& \quad \rightarrow_1 (y := z, \{ x \mapsto 7, y \mapsto 7, z \mapsto 5 \}) \\
& \quad \rightarrow_1 \{ x \mapsto 7, y \mapsto 5, z \mapsto 5 \}
\end{align*}
\]

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)

Derivation Trees

▶ Derivation trees explain why transitions take place

▶ For the first step

\[ (z := x; x := y; y := z, \sigma) \rightarrow_1 (x := y; y := z, \sigma[z \mapsto 5]) \]

the derivation tree is

\[
\begin{align*}
(z := x, \sigma[z \mapsto 5]) & \quad \rightarrow_1 (x := y, \sigma[z \mapsto 5]) \\
& \quad \rightarrow_1 (y := z, \sigma[z \mapsto 5]) \\
& \quad \rightarrow_1 \{ x \mapsto 7, y \mapsto 5, z \mapsto 5 \}
\end{align*}
\]

\[ z := x; (x := y; y := z) \]

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
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

\[
\langle s_1; s_2, \sigma \rangle \rightarrow_1^{k_1} \sigma'' \Rightarrow \\
\exists \sigma', k_1, k_2 : \langle s_1, \sigma \rangle \rightarrow_1^{k_1} \sigma' \land \langle s_2, \sigma'_1 \rangle \rightarrow_1^{k_2} \sigma'' \land \\
k_1 + k_2 = k
\]

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 \( \gamma \)
**Induction Step**

- \( \langle s_1; s_2, \sigma \rangle \rightarrow_1 \gamma \rightarrow_1 \sigma'' \)
- Consider the two rules that could lead to the transition \( \langle s_1; s_2, \sigma \rangle \rightarrow_1 \gamma \)
- Case 1

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

- 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 2**

- From \( \langle s_1; s_2, \sigma \rangle \rightarrow_1 \gamma \rightarrow_1 \sigma'' \) and \( \langle s_1; s_2, \sigma \rangle \rightarrow_1 \langle s_1'; s_2, \sigma' \rangle \)
  we conclude \( \langle s_1'; s_2, \sigma' \rangle \rightarrow_1 \sigma'' \)
- By applying the induction hypothesis, we get
  \( \exists \sigma_0, l_1, l_2 : \langle s_1', \sigma' \rangle \rightarrow_1 \sigma_0 \) \( \land \) \( \langle s_2, \sigma_0 \rangle \rightarrow_1 \sigma'' \) \( \land \) \( l_1 + l_2 = m \)
- From \( \langle s_1, \sigma \rangle \rightarrow_1 \langle s_1', \sigma' \rangle \) and \( \langle s_1', \sigma'' \rangle \rightarrow_1 \sigma_0 \)
  we get \( \langle s_1, \sigma \rangle \rightarrow_1 \langle s_1', \sigma'' \rangle \)
- By \( \langle s_2, \sigma_0 \rangle \rightarrow_1 \sigma'' \) 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 \( \sigma \):

- \( \langle s_1, \sigma \rangle \rightarrow_1 \gamma \) iff \( \langle s_2, \sigma \rangle \rightarrow_1 \gamma \), whenever \( \gamma \) 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 $s, \sigma, \gamma, \gamma'$ we have that

$$\langle s, \sigma \rangle \mathbin{\rightarrow_1} \gamma \land \langle s, \sigma \rangle \mathbin{\rightarrow_1} \gamma' \Rightarrow \gamma = \gamma'$$

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

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

Extensions of IMP

- Local variable declaration $\text{var } x := e \text{ in } s \text{ end}$
- Statement "abort"
- Non-determinism
- Parallelism

Local Variable Declarations

- Local variable declaration $\text{var } x := e \text{ in } s \text{ end}$
- 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

\[
\text{State} : \text{stack of } (\text{Var} \rightarrow \text{Val})
\]

- Assignment and lookup have to determine the highest stack element in which a variable is defined

Example: \( \sigma(x) = 3 \)

\[
\begin{align*}
z & \mapsto 4 \\
x & \mapsto 3 \\
y & \mapsto 2
\end{align*}
\]

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

\[
\langle \text{var } x := e \text{ in } s \text{ end, } \sigma \rangle \rightarrow_1 \\
\langle s; \text{return, push, } \{ x \mapsto A[e] \sigma \}, \sigma \rangle \\
\langle \text{return, } \sigma \rangle \rightarrow_1 \text{pop} (\sigma)
\]

- Similar techniques can be used for procedure calls

Abortion

- Statement \text{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 \text{abort} in the structural operational semantics

- \text{abort} and \text{skip} are not semantically equivalent
  - \( \langle \text{abort, } \sigma \rangle \) is the only derivation sequence for \text{abort} starting is \( s \)
  - \( \langle \text{skip, } \sigma \rangle \rightarrow_1 \sigma \) is the only derivation sequence for \text{skip} starting is \( s \)

Abortion: Observations

- \text{abort} and while true do \text{skip} end are not semantically equivalent:
  - \( \langle \text{while true do skip end, } \sigma \rangle \rightarrow_1 \)
  - \( \langle \text{if true then skip; while true do skip end end, } \sigma \rangle \rightarrow_1 \)
  - \( \langle \text{skip; while true do skip end, } \sigma \rangle \rightarrow_1 \)
  - \( \langle \text{while true do skip end, } \sigma \rangle \)

- 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 $x:=1; x:=x+2$ could result in a state in which $x$ has the value 1 or 4.
- Rules:
  \[
  \langle s_1 \parallel s_2, \sigma \rangle \rightarrow_1 \langle s_1', \sigma \rangle
  \]
  \[
  \langle s_1 \parallel s_2, \sigma \rangle \rightarrow_1 \langle s_2', \sigma \rangle
  \]

Parallelism

- For the statement $s_1 \parallel s_2$ both statements $s_1$ and $s_2$ are executed, but execution can be interleaved.
- Rules:
  \[
  \langle s_1, \sigma \rangle \rightarrow_1 \langle s_1', \sigma' \rangle
  \]
  \[
  \langle s_1 \parallel s_2, \sigma \rangle \rightarrow_1 \langle s_1', \parallel s_2, \sigma' \rangle
  \]
  \[
  \langle s_2, \sigma \rangle \rightarrow_1 \langle s_2', \sigma' \rangle
  \]
  \[
  \langle s_1 \parallel s_2, \sigma \rangle \rightarrow_1 \langle s_1 \parallel s_2', \sigma' \rangle
  \]
  \[
  \langle s_2, \sigma \rangle \rightarrow_1 \langle s_1, \parallel s_2', \sigma' \rangle
  \]
  \[
  \langle s_1 \parallel s_2, \sigma \rangle \rightarrow_1 \langle s_1', \parallel s_2, \sigma' \rangle
  \]

Non-determinism: Observations

- There are two derivation sequences:
  \[
  \langle x:=1; x:=x+2, \sigma \rangle \rightarrow_1 \langle x:=1; x:=x+2, \sigma[x \leftarrow 1] \rangle
  \]
  \[
  \langle x:=1; x:=x+2, \sigma \rangle \rightarrow_1 \langle x:=1; x:=x+2, \sigma[x \leftarrow 4] \rangle
  \]
- There are also two derivation sequences for:
  \[
  \langle \text{while true do skip end}; x:=2; x:=x+2, \sigma \rangle
  \]\n  - an infinite derivation sequence leading to $\sigma[x \leftarrow 4]$
  - 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.

Example: Interleaving

- The statement $x:=1; x:=x+2$ 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

\[
\begin{align*}
(x := 1 & \text{par } x := 2; \ x := x + 2, \sigma) & \rightarrow_1 (x := 2; \ x := x + 2, \sigma[x \mapsto 1]) \\
& \rightarrow_1 (x := x + 2, \sigma[x \mapsto 2]) \\
& \rightarrow_1 \sigma[x \mapsto 4] \\
(x := 1 & \text{par } x := 2; \ x := x + 2, \sigma) & \rightarrow_1 (x := 1 \text{par } x := x + 2, \sigma[x \mapsto 2]) \\
& \rightarrow_1 (x := 1, \sigma[x \mapsto 4]) \\
& \rightarrow_1 \sigma[x \mapsto 1] \\
(x := 1 & \text{par } x := 2; \ x := x + 2, \sigma) & \rightarrow_1 (x := 1 \text{par } x := x + 2, \sigma[x \mapsto 2]) \\
& \rightarrow_1 (x := x + 2, \sigma[x \mapsto 1]) \\
& \rightarrow_1 \sigma[x \mapsto 3]
\end{align*}
\]

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

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
7. Programming Language Semantics

7.4 Denotational semantics

package cells;
public interface Val{};

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

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 $A : Aexp \rightarrow State \rightarrow Val$ and $B : Bexp \rightarrow State \rightarrow Bool$ are denotational definitions

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

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

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

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

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

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

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

Semantic Functions

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

\[
S_{DS} : \text{Stm} \rightarrow (\text{State} \rightarrow \text{State})
\]

- 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

- skip does not modify the state.

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

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

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

- $x := e$ 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.

\[
(f \circ g)\sigma = \begin{cases} 
f(g(\sigma)) & \text{if } g(\sigma) \neq \text{undefined} \text{ and } f(g(\sigma)) \neq \text{undefined} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
**Direct Style Semantics of IMP (cont’d)**

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

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

- 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

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

**Semantics of Loop: Observations**

- Defining the semantics of \( \text{while } b \text{ do } s \text{ end} \) is difficult
- The semantics of \( \text{while } b \text{ do } s \text{ end} \) must be equal to \( \text{if } b \text{ then } s; \text{while } b \text{ do } s \text{ end} \text{ skip end} \)
- This requirement yields:

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

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

**Functionals and Fixed Points**

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

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

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

We write $\text{FIX } F$ to denote the fixed point of the functional $F$:

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

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

Example

Consider the statement

while $x \neq 0$ do skip end

The functional for this loop is defined by

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

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

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

$= \begin{cases} 
  g(\sigma) & \text{if } \sigma(x) \neq 0 \\
  \sigma & \text{if } \sigma(x) = 0 
\end{cases}$

is a fixed point of $F'$

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 $F'$.}

Example (cont’d)

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

The function $g_1(\sigma)$ is a fixed point of $F'$ because

$F'(g_1)\sigma = \text{cond}(B[x \neq 0], g_1 \circ S_{DS}[\text{skip}], \text{id})\sigma$

$= \text{cond}(B[x \neq 0], g_1 \circ \text{id}, \text{id})\sigma$

$= \text{cond}(B[x \neq 0], g_1, \text{id})\sigma$

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

This statement is not a fixed point of $F''$. 

The function $g_2(\sigma)$ is not a fixed point of $F''$ because

$F''(g_2)\sigma = \text{cond}(B[x \neq 0], g_2 \circ S_{DS}[\text{skip}], \text{id})\sigma$

$= \text{cond}(B[x \neq 0], g_2 \circ \text{id}, \text{id})\sigma$

$= \text{cond}(B[x \neq 0], g_2, \text{id})\sigma$

$= \begin{cases} 
  g_2(\sigma) & \text{if } \sigma(x) \neq 0 \\
  \sigma & \text{if } \sigma(x) = 0 
\end{cases}$

This statement is not defined for $\sigma(x) = 0$.
Well-Definedness

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

- The function \( S_{DS}[\text{while } b \text{ do } s \text{ end}] \) is well-defined if \( \text{FIX } F \) defines a \textbf{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
  \[ F'(g) = \begin{cases} g(\sigma) & \text{if } \sigma(x) \neq 0 \\ \sigma & \text{otherwise} \end{cases} \]

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

Fixed point theory for functions

**Strict functions**
Let \( D_\perp = \text{def } D \cup \{ \perp \} \text{ with } \perp \notin D. \)
A function \( f : D_\perp \rightarrow D_\perp \) is called \textit{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 \( f \leq g \text{ if } f(x) = g(x) \text{ or } f(x) = \perp. \)
\( (\mathcal{D}, \leq) \) is a \textit{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 \textit{continuous} iff for every chain \( X \) in \( \mathcal{D} \):
\[ F(\bigvee X) = \bigvee \{ F(x) \mid x \in X \} \]

**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