

Semantics and Verification of Software

Lecture 4: Operational vs. Denotational Semantics of WHILE

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)



noll@cs.rwth-aachen.de

<http://www-i2.informatik.rwth-aachen.de/i2/svsw13/>

Summer Semester 2013

- 1 Recapitulation: Execution of Statements
- 2 Functional of the Operational Semantics
- 3 Summary: Operational Semantics
- 4 The Denotational Approach
- 5 Denotational Semantics of Expressions

Execution of Statements

Remember:

$c ::= \text{skip} \mid x := a \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \in \text{Cmd}$

Definition (Execution relation for statements)

For $c \in \text{Cmd}$ and $\sigma, \sigma' \in \Sigma$, the **execution relation** $\langle c, \sigma \rangle \rightarrow \sigma'$ is defined by the following rules:

$$\begin{array}{lcl} \text{(skip)} \frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma} & \text{(asgn)} \frac{\langle a, \sigma \rangle \rightarrow z}{\langle x := a, \sigma \rangle \rightarrow \sigma[x \mapsto z]} \\ \text{(seq)} \frac{\langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma''} & \text{(if-t)} \frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle c_1, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \sigma'} \\ \text{(if-f)} \frac{\langle b, \sigma \rangle \rightarrow \text{false} \quad \langle c_2, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \sigma'} & \text{(wh-f)} \frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma} \\ \text{(wh-t)} \frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma' \quad \langle \text{while } b \text{ do } c, \sigma' \rangle \rightarrow \sigma''}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma''} \end{array}$$

Determinism of Execution Relation I

This operational semantics is well defined in the following sense:

Theorem

*The execution relation for statements is **deterministic**, i.e., whenever $c \in \text{Cmd}$ and $\sigma, \sigma', \sigma'' \in \Sigma$ such that $\langle c, \sigma \rangle \rightarrow \sigma'$ and $\langle c, \sigma \rangle \rightarrow \sigma''$, then $\sigma' = \sigma''$.*

- How to prove this theorem?
- Idea:
 - employ corresponding result for **expressions** (Lemma 3.6)
 - use **induction on the syntactic structure** of c ↴
- Instead: **structural induction on derivation trees**

- 1 Recapitulation: Execution of Statements
- 2 Functional of the Operational Semantics
- 3 Summary: Operational Semantics
- 4 The Denotational Approach
- 5 Denotational Semantics of Expressions

The determinism of the execution relation (Theorem 3.5) justifies the following definition:

Definition 4.1 (Operational functional)

The **functional of the operational semantics**,

$$\mathcal{D}[\![\cdot]\!] : \mathit{Cmd} \rightarrow (\Sigma \dashrightarrow \Sigma),$$

assigns to every statement $c \in \mathit{Cmd}$ a partial state transformation $\mathcal{D}[\![c]\!] : \Sigma \dashrightarrow \Sigma$, which is defined as follows:

$$\mathcal{D}[\![c]\!]\sigma := \begin{cases} \sigma' & \text{if } \langle c, \sigma \rangle \rightarrow \sigma' \text{ for some } \sigma' \in \Sigma \\ \text{undefined} & \text{otherwise} \end{cases}$$

Functional of the Operational Semantics

The determinism of the execution relation (Theorem 3.5) justifies the following definition:

Definition 4.1 (Operational functional)

The **functional of the operational semantics**,

$$\mathcal{D}[\![\cdot]\!] : \text{Cmd} \rightarrow (\Sigma \dashrightarrow \Sigma),$$

assigns to every statement $c \in \text{Cmd}$ a partial state transformation $\mathcal{D}[\![c]\!] : \Sigma \dashrightarrow \Sigma$, which is defined as follows:

$$\mathcal{D}[\![c]\!]\sigma := \begin{cases} \sigma' & \text{if } \langle c, \sigma \rangle \rightarrow \sigma' \text{ for some } \sigma' \in \Sigma \\ \text{undefined} & \text{otherwise} \end{cases}$$

Remark: $\mathcal{D}[\![c]\!]\sigma$ can indeed be undefined
(consider e.g. $c = \text{while true do skip}$; see Corollary 3.4)

Equivalence of Statements

Underlying principle: two (syntactic) objects are considered (semantically) **equivalent** if they have the same “meaning”

- finite automata: $A_1 \sim A_2$ iff $L(A_1) = L(A_2)$
- context-free grammars: $G_1 \sim G_2$ iff $L(G_1) = L(G_2)$
- Turing machines: $T_1 \sim T_2$ iff both compute same function

Equivalence of Statements

Underlying principle: two (syntactic) objects are considered (semantically) **equivalent** if they have the same “meaning”

- finite automata: $A_1 \sim A_2$ iff $L(A_1) = L(A_2)$
- context-free grammars: $G_1 \sim G_2$ iff $L(G_1) = L(G_2)$
- Turing machines: $T_1 \sim T_2$ iff both compute same function

Definition 4.2 (Operational equivalence)

Two statements $c_1, c_2 \in \text{Cmd}$ are called **(operationally) equivalent** (notation: $c_1 \sim c_2$) iff

$$\mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!].$$

Equivalence of Statements

Underlying principle: two (syntactic) objects are considered (semantically) **equivalent** if they have the same “meaning”

- finite automata: $A_1 \sim A_2$ iff $L(A_1) = L(A_2)$
- context-free grammars: $G_1 \sim G_2$ iff $L(G_1) = L(G_2)$
- Turing machines: $T_1 \sim T_2$ iff both compute same function

Definition 4.2 (Operational equivalence)

Two statements $c_1, c_2 \in \text{Cmd}$ are called **(operationally) equivalent** (notation: $c_1 \sim c_2$) iff

$$\mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!].$$

Thus:

- $c_1 \sim c_2$ iff $\mathcal{D}[\![c_1]\!]\sigma = \mathcal{D}[\![c_2]\!]\sigma$ for every $\sigma \in \Sigma$
- In particular, $\mathcal{D}[\![c_1]\!]\sigma$ is undefined iff $\mathcal{D}[\![c_2]\!]\sigma$ is undefined

“Unwinding” of Loops

Simple application of statement equivalence: test of execution condition in a `while` loop can be represented by an `if` statement

Lemma 4.3

For every $b \in BExp$ and $c \in Cmd$,

$while\ b\ do\ c \sim if\ b\ then\ (c; while\ b\ do\ c)\ else\ skip.$

“Unwinding” of Loops

Simple application of statement equivalence: test of execution condition in a `while` loop can be represented by an `if` statement

Lemma 4.3

For every $b \in BExp$ and $c \in Cmd$,

$\text{while } b \text{ do } c \sim \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip.}$

Proof.

on the board □

- 1 Recapitulation: Execution of Statements
- 2 Functional of the Operational Semantics
- 3 Summary: Operational Semantics**
- 4 The Denotational Approach
- 5 Denotational Semantics of Expressions

- Formalized by **evaluation/execution relations**

Summary: Operational Semantics

- Formalized by **evaluation/execution relations**
- Inductively defined by **derivation trees** using **structural operational rules**

- Formalized by **evaluation/execution relations**
- Inductively defined by **derivation trees** using **structural operational rules**
- Enables proofs about operational behavior of programs using **structural induction** on derivation trees

Summary: Operational Semantics

- Formalized by **evaluation/execution relations**
- Inductively defined by **derivation trees** using **structural operational rules**
- Enables proofs about operational behavior of programs using **structural induction** on derivation trees
- **Semantic functional** characterizes complete input/output behavior of programs

- 1 Recapitulation: Execution of Statements
- 2 Functional of the Operational Semantics
- 3 Summary: Operational Semantics
- 4 The Denotational Approach**
- 5 Denotational Semantics of Expressions

- Primary aspect of a program: its “effect”, i.e., **input/output behavior**

- Primary aspect of a program: its “effect”, i.e., **input/output behavior**
- In operational semantics: **indirect** definition of semantic functional $\mathcal{D}[\cdot] : Cmd \rightarrow (\Sigma \dashrightarrow \Sigma)$ by execution relation

- Primary aspect of a program: its “effect”, i.e., **input/output behavior**
- In operational semantics: **indirect** definition of semantic functional $\mathcal{D}[\![\cdot]\!] : Cmd \rightarrow (\Sigma \dashrightarrow \Sigma)$ by execution relation
- Now: **abstract** from operational details

- Primary aspect of a program: its “effect”, i.e., **input/output behavior**
- In operational semantics: **indirect** definition of semantic functional $\mathcal{D}[\cdot] : Cmd \rightarrow (\Sigma \dashrightarrow \Sigma)$ by execution relation
- Now: **abstract** from operational details
- **Denotational semantics**: direct definition of program effect by induction on its syntactic structure

- 1 Recapitulation: Execution of Statements
- 2 Functional of the Operational Semantics
- 3 Summary: Operational Semantics
- 4 The Denotational Approach
- 5 Denotational Semantics of Expressions

Again: value of an expression determined by current state

Definition 4.4 (Denotational semantics of arithmetic expressions)

The (denotational) semantic functional for arithmetic expressions,

$$\mathcal{A}[\![\cdot]\!] : AExp \rightarrow (\Sigma \rightarrow \mathbb{Z}),$$

is given by:

$$\begin{array}{ll} \mathcal{A}[\![z]\!]\sigma := z & \mathcal{A}[\![a_1 + a_2]\!]\sigma := \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{A}[\![x]\!]\sigma := \sigma(x) & \mathcal{A}[\![a_1 - a_2]\!]\sigma := \mathcal{A}[\![a_1]\!]\sigma - \mathcal{A}[\![a_2]\!]\sigma \\ & \mathcal{A}[\![a_1 * a_2]\!]\sigma := \mathcal{A}[\![a_1]\!]\sigma \cdot \mathcal{A}[\![a_2]\!]\sigma \end{array}$$

Definition 4.5 (Denotational semantics of Boolean expressions)

The (denotational) semantic functional for Boolean expressions,

$$\mathfrak{B}[\cdot] : BExp \rightarrow (\Sigma \rightarrow \mathbb{B}),$$

is given by:

$$\begin{aligned}\mathfrak{B}[\![t]\!]\sigma &:= t \\ \mathfrak{B}[\![a_1 = a_2]\!]\sigma &:= \begin{cases} \text{true} & \text{if } \mathfrak{A}[\![a_1]\!]\sigma = \mathfrak{A}[\![a_2]\!]\sigma \\ \text{false} & \text{otherwise} \end{cases} \\ \mathfrak{B}[\![a_1 > a_2]\!]\sigma &:= \begin{cases} \text{true} & \text{if } \mathfrak{A}[\![a_1]\!]\sigma > \mathfrak{A}[\![a_2]\!]\sigma \\ \text{false} & \text{otherwise} \end{cases} \\ \mathfrak{B}[\![\neg b]\!]\sigma &:= \begin{cases} \text{true} & \text{if } \mathfrak{B}[\![b]\!]\sigma = \text{false} \\ \text{false} & \text{otherwise} \end{cases} \\ \mathfrak{B}[\![b_1 \wedge b_2]\!]\sigma &:= \begin{cases} \text{true} & \text{if } \mathfrak{B}[\![b_1]\!]\sigma = \mathfrak{B}[\![b_2]\!]\sigma = \text{true} \\ \text{false} & \text{otherwise} \end{cases} \\ \mathfrak{B}[\![b_1 \vee b_2]\!]\sigma &:= \begin{cases} \text{false} & \text{if } \mathfrak{B}[\![b_1]\!]\sigma = \mathfrak{B}[\![b_2]\!]\sigma = \text{false} \\ \text{true} & \text{otherwise} \end{cases}\end{aligned}$$