



# Semantics and Verification of Software

Winter Semester 2017/18

Lecture 4: Operational Semantics of WHILE III  
(Summary & Application to Compiler Correctness)

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<http://moves.rwth-aachen.de/teaching/ws-1718/sv-sw/>

# Recap: Execution of Statements

## Execution of Statements

### Remember:

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

### 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:

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

# Recap: Execution of Statements

---

## Determinism of Execution Relation

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**

# Functional of the Operational Semantics

---

## 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.  $c = \text{while true do skip end}$ ; see Corollary 3.4)

# Functional of the Operational Semantics

---

## 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  $c_1, c_2 \in \mathit{Cmd}$  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

# Functional of the Operational Semantics

---

## “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 end`  $\sim$  `if b then c; while b do c end else skip end.`

Proof.

on the board



# Summary: Operational Semantics

---

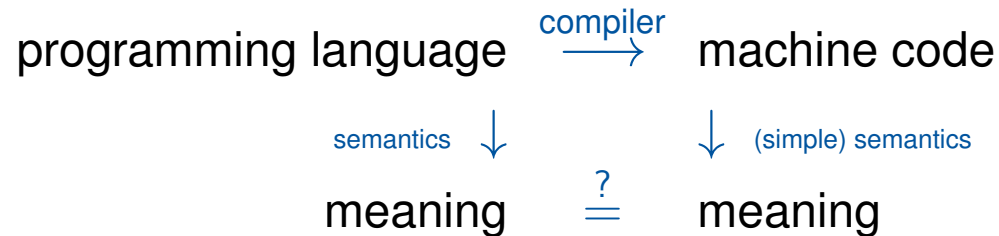
## Summary: Operational Semantics

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

# Application: Compiler Correctness

---

## Compiler Correctness



### To do:

1. Definition of **abstract machine**
2. Definition of (operational) **semantics of machine instructions**
3. Definition of **translation** WHILE  $\rightarrow$  machine code (“compiler”)
4. **Proof:** semantics of generated machine code = semantics of original source code



# The Abstract Machine

## The Abstract Machine

### Definition 4.4 (Abstract machine)

The **abstract machine (AM)** is given by

- **programs**  $P \in \text{Code}$  and **instructions**  $p$ :

$$P ::= p^*$$

$$p ::= \text{PUSH}(z) \mid \text{PUSH}(t) \mid \text{ADD} \mid \text{SUB} \mid \text{MULT} \mid \text{EQ} \mid \text{GT} \mid \text{NOT} \mid \text{AND} \mid \text{OR} \mid \\ \text{LOAD}(x) \mid \text{STO}(x) \mid \text{JMP}(k) \mid \text{JMPF}(k)$$

(where  $z, k \in \mathbb{Z}$ ,  $t \in \mathbb{B}$ , and  $x \in \text{Var}$ )

- **configurations** of the form  $\langle pc, e, \sigma \rangle \in \text{Cnf}$  where
  - $pc \in \mathbb{Z}$  is the **program counter** (i.e., address of next instruction to be executed)
  - $e \in \text{Stk} := (\mathbb{Z} \cup \mathbb{B})^*$  is the **evaluation stack** (top right)
  - $\sigma \in \Sigma := (\text{Var} \rightarrow \mathbb{Z})$  is the **(storage) state**(thus  $\text{Cnf} = \mathbb{Z} \times \text{Stk} \times \Sigma$ )
- **initial configurations** of the form  $\langle 0, \varepsilon, \sigma \rangle$
- **final configurations** of the form  $\langle |P|, e, \sigma \rangle$

# The Abstract Machine

## Semantics of AM-Code I

### Definition 4.5 (Transition relation of AM)

For  $P = p_0; \dots; p_{n-1} \in \text{Code}$  and  $0 \leq pc < n$ , the **transition relation**  $\triangleright \subseteq \text{Cnf} \times \text{Cnf}$  is given by

$P \vdash \langle pc, e, \sigma \rangle \triangleright \langle pc + 1, e : z, \sigma \rangle$	if $p_{pc} = \text{PUSH}(z)$
$P \vdash \langle pc, e, \sigma \rangle \triangleright \langle pc + 1, e : t, \sigma \rangle$	if $p_{pc} = \text{PUSH}(t)$
$P \vdash \langle pc, e : z_1 : z_2, \sigma \rangle \triangleright \langle pc + 1, e : (z_1 + z_2), \sigma \rangle$	if $p_{pc} = \text{ADD}$
$P \vdash \langle pc, e : z_1 : z_2, \sigma \rangle \triangleright \langle pc + 1, e : (z_1 - z_2), \sigma \rangle$	if $p_{pc} = \text{SUB}$
$P \vdash \langle pc, e : z_1 : z_2, \sigma \rangle \triangleright \langle pc + 1, e : (z_1 \cdot z_2), \sigma \rangle$	if $p_{pc} = \text{MULT}$
$P \vdash \langle pc, e : z_1 : z_2, \sigma \rangle \triangleright \langle pc + 1, e : (z_1 = z_2), \sigma \rangle$	if $p_{pc} = \text{EQ}$
$P \vdash \langle pc, e : z_1 : z_2, \sigma \rangle \triangleright \langle pc + 1, e : (z_1 > z_2), \sigma \rangle$	if $p_{pc} = \text{GT}$
$P \vdash \langle pc, e : t, \sigma \rangle \triangleright \langle pc + 1, e : (\neg t), \sigma \rangle$	if $p_{pc} = \text{NOT}$
$P \vdash \langle pc, e : t_1 : t_2, \sigma \rangle \triangleright \langle pc + 1, e : (t_1 \wedge t_2), \sigma \rangle$	if $p_{pc} = \text{AND}$
$P \vdash \langle pc, e : t_1 : t_2, \sigma \rangle \triangleright \langle pc + 1, e : (t_1 \vee t_2), \sigma \rangle$	if $p_{pc} = \text{OR}$
$P \vdash \langle pc, e, \sigma \rangle \triangleright \langle pc + 1, e : \sigma(x), \sigma \rangle$	if $p_{pc} = \text{LOAD}(x)$
$P \vdash \langle pc, e : z, \sigma \rangle \triangleright \langle pc + 1, e, \sigma[x \mapsto z] \rangle$	if $p_{pc} = \text{STO}(x)$
$P \vdash \langle pc, e, \sigma \rangle \triangleright \langle pc + k, e, \sigma \rangle$	if $p_{pc} = \text{JMP}(k)$
$P \vdash \langle pc, e : \text{true}, \sigma \rangle \triangleright \langle pc + 1, e, \sigma \rangle$	if $p_{pc} = \text{JMPF}(k)$
$P \vdash \langle pc, e : \text{false}, \sigma \rangle \triangleright \langle pc + k, e, \sigma \rangle$	if $p_{pc} = \text{JMPF}(k)$

# The Abstract Machine

## Semantics of AM-Code II

### Corollary 4.6

▷ *is not total*, i.e., there exists  $\gamma \in \text{Cnf}$  such that

$$\gamma \not\triangleright \gamma'$$

for all  $\gamma' \in \text{Cnf}$

### Proof.

Possible cases:

- $\gamma$  **final** (that is,  $\gamma = \langle |P|, e, \sigma \rangle$ )
- $\gamma$  **stuck**
  - e.g.,  $\gamma = \langle pc, 1, \sigma \rangle$  with  $p_{pc} = \text{ADD}$  or  $p_{pc} = \text{JMPF}(k)$
  - or  $\gamma = \langle pc, e, \sigma \rangle$  with  $pc \notin \{0, \dots, |P|\}$



# The Abstract Machine

---

## Alternative Choices

**Remark:** more realistic machine architectures

- **Variables referenced by address** (and not by name)
  - configurations  $\langle pc, e, \mu \rangle$  with **memory**  $\mu \in (\mathbb{N} \rightarrow \mathbb{Z})$
  - $LOAD(x)/STO(x)$  replaced by  $LOAD(m)/STO(m)$  (where  $m \in \mathbb{N}$ )(requires symbol table for translation)
- **Registers** for storing intermediate values  
(in place of evaluation stack  $e$ ; involves register allocation)

## Terminating and Looping Computations I

### Definition 4.7 (AM computations)

- A **finite computation** is a finite configuration sequence of the form  $\gamma_0, \gamma_1, \dots, \gamma_k$  where  $k \in \mathbb{N}$  and  $\gamma_{i-1} \triangleright \gamma_i$  for each  $i \in \{1, \dots, k\}$
- If, in addition, there is no  $\gamma$  such that  $\gamma_k \triangleright \gamma$ , then  $\gamma_0, \gamma_1, \dots, \gamma_k$  is called **terminating**
- A **looping computation** is an infinite configuration sequence of the form  $\gamma_0, \gamma_1, \gamma_2, \dots$  where  $\gamma_i \triangleright \gamma_{i+1}$  for each  $i \in \mathbb{N}$

**Note:** according to (the proof of) Corollary 4.6, a terminating computation may end in a final or in a stuck configuration

## Terminating and Looping Computations II

### Example 4.8

1. For  $P := 0:\text{LOAD}(x); 1:\text{PUSH}(1); 2:\text{ADD}; 3:\text{STO}(x)$  and  $\sigma(x) = 3$ , we obtain the following terminating computation:

$$\langle 0, \varepsilon, \sigma \rangle \triangleright \langle 1, 3, \sigma \rangle \triangleright \langle 2, 3 : 1, \sigma \rangle \triangleright \langle 3, 4, \sigma \rangle \triangleright \langle 4, \varepsilon, \sigma[x \mapsto 4] \rangle$$

**Remark:** implements statement  $x := x + 1$

2. For  $P := 0:\text{PUSH}(\text{true}); 1:\text{JMPF}(2); 2:\text{JMP}(-2)$ , the following computation loops:

$$\langle 0, \varepsilon, \sigma \rangle \triangleright \langle 1, \text{true}, \sigma \rangle \triangleright \langle 2, \varepsilon, \sigma \rangle \triangleright \langle 0, \varepsilon, \sigma \rangle \triangleright \dots$$

**Remark:** implements statement `while true do skip end`

# Properties of AM

---

## A New Inductive Principle

Application: Finite computations (Def. 4.7)

Definition: a finite computation  $\gamma_0, \gamma_1, \dots, \gamma_k$  has length  $k$

Induction base: property holds for all computations of length 0

Induction hypothesis: property holds for all computations of length  $\leq k$

Induction step: property holds for all computations of length  $k + 1$

# Properties of AM

---

## Application: Extension of Code and Stack

### Lemma 4.9

If  $P \vdash \langle pc, e, \sigma \rangle \triangleright^* \langle pc', e', \sigma' \rangle$ , then

$$P_1; P; P_2 \vdash \langle |P_1| + pc, e_0 : e, \sigma \rangle \triangleright^* \langle |P_1| + pc', e_0 : e', \sigma' \rangle$$

for all  $P_1, P_2 \in \text{Code}$  and  $e_0 \in \text{Stk}$ .

**Interpretation:** both the code and the stack component can be extended without actually changing the behaviour of the machine

### Proof.

by induction on the length of the computation (on the board) □



# Properties of AM

## Another Property: Determinism

### Lemma 4.10

The semantics of AM is **deterministic**: for all  $\gamma, \gamma', \gamma'' \in \text{Cnf}$ ,

$$P \vdash \gamma \triangleright \gamma' \text{ and } P \vdash \gamma \triangleright \gamma'' \text{ implies } \gamma' = \gamma''.$$

### Proof (Idea).

- Instruction to be executed is unambiguously given by program counter
- Topmost stack entries and storage state then yield unique successor configuration □

Thus the following function is well defined:

### Definition 4.11 (Semantics of AM)

The **semantics of an AM program** is given by  $\mathfrak{M}[\![\cdot]\!] : \text{Code} \rightarrow (\Sigma \dashrightarrow \Sigma)$  as follows:

$$\mathfrak{M}[\![P]\!]\sigma := \begin{cases} \sigma' & \text{if } P \vdash \langle 0, \varepsilon, \sigma \rangle \triangleright^* \langle |P|, e, \sigma' \rangle \text{ for some } e \in \text{Stk} \\ \text{undefined} & \text{otherwise} \end{cases}$$