Semantics and Verification of Software

Summer Semester 2015

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/ss-15/sv-sw/

Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

Outline of Lecture 4

Execution of Statements

Remember:

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

Definition (Execution relation for statements)

For $c \in \mathit{Cmd}$ and $\sigma, \sigma' \in \Sigma$, the execution relation $\langle c, \sigma \rangle \to \sigma'$ is defined by:

$$\text{(skip)}\ \frac{}{\langle \texttt{skip}, \sigma \rangle \to \sigma} \qquad \text{(asgn)}\ \frac{\langle a, \sigma \rangle \to z}{\langle x := a, \sigma \rangle \to \sigma[x \mapsto z]}$$

$$\text{(seq)}\ \frac{\langle c_1, \sigma \rangle \to \sigma' \quad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1 ; c_2, \sigma \rangle \to \sigma''} \qquad \text{(if-t)}\ \frac{\langle b, \sigma \rangle \to \mathsf{true} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}, \sigma \rangle \to \sigma'}$$

$$\text{(if-f)}\ \frac{\langle b, \sigma \rangle \to \mathsf{false} \quad \langle c_2, \sigma \rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}, \sigma \rangle \to \sigma'} \qquad \text{(wh-f)}\ \frac{\langle b, \sigma \rangle \to \mathsf{false}}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}, \sigma \rangle \to \sigma}$$

$$\text{(wh-t)}\ \frac{\langle b, \sigma \rangle \to \mathsf{true} \quad \langle c, \sigma \rangle \to \sigma' \quad \langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}, \sigma' \rangle \to \sigma''}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}, \sigma \rangle \to \sigma''}$$

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 $c \in Cmd$ and $\sigma, \sigma', \sigma'' \in \Sigma$ such that $\langle c, \sigma \rangle \to \sigma'$ and $\langle c, \sigma \rangle \to \sigma''$, 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 $\lightning$
- Instead: structural induction on derivation trees

Outline of Lecture 4

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,

$$\mathfrak{O}[\![.]\!] : \mathit{Cmd} \to (\Sigma \dashrightarrow \Sigma),$$

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

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

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,

$$\mathfrak{O}[\![.]\!] : \mathit{Cmd} \to (\Sigma \dashrightarrow \Sigma),$$

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

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

Remark: $\mathfrak{O}[\![c]\!]\sigma$ can indeed be undefined (consider e.g. $c = $ `while true do skip end`; 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

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 $c_1, c_2 \in Cmd$ are called (operationally) equivalent (notation: $c_1 \sim c_2$) iff

$$\mathfrak{O}[\![c_1]\!] = \mathfrak{O}[\![c_2]\!].$$

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 $c_1, c_2 \in Cmd$ are called (operationally) equivalent (notation: $c_1 \sim c_2$) iff

$$\mathfrak{O}[\![c_1]\!] = \mathfrak{O}[\![c_2]\!].$$

Thus:

- $c_1 \sim c_2$ iff $\mathfrak{O}[\![c_1]\!]\sigma = \mathfrak{O}[\![c_2]\!]\sigma$ for every $\sigma \in \Sigma$
- In particular, $\mathfrak{O}[\![c_1]\!]\sigma$ is undefined iff $\mathfrak{O}[\![c_2]\!]\sigma$ 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 $c \in Cmd$,

$$\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end} \sim \texttt{if}\ b\ \texttt{then}\ c\,\texttt{;}\,\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ \texttt{else}\ \texttt{skip}\ \texttt{end}.$$

"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 $c \in Cmd$,

$$\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end} \sim \texttt{if}\ b\ \texttt{then}\ c\texttt{;while}\ b\ \texttt{do}\ c\ \texttt{end}\ \texttt{else}\ \texttt{skip}\ \texttt{end}.$$

Proof.

on the board □

Outline of Lecture 4

Summary: Operational Semantics

- Formalized by evaluation/execution relations

Summary: Operational Semantics

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

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

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

Outline of Lecture 4

Application: Compiler Correctness

Compiler Correctness

$$\begin{tikzcd} \text{programming language} \arrow[r, "\text{compiler}"] \arrow[d, "\text{semantics}"'] & \text{machine code} \arrow[d, "\text{(simple) semantics}"] \\ \text{meaning} \arrow[r, phantom, "\stackrel{?}{=}"] & \text{meaning} \end{tikzcd}$$

Application: Compiler Correctness

Compiler Correctness

$$
\begin{tikzcd}
\text{programming language} \arrow[r, "\text{compiler}"] \arrow[d, "\text{semantics}"'] & \text{machine code} \arrow[d, "\text{(simple) semantics}"] \\
\text{meaning} \arrow[r, phantom, "\stackrel{?}{=}"] & \text{meaning}
\end{tikzcd}
$$

To do:

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

Outline of Lecture 4

The Abstract Machine

Definition 4.4 (Abstract machine)

The abstract machine (AM) is given by

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

$$\begin{aligned} P &::= p^* \\ p &::= \mathtt{PUSH}(z) \mid \mathtt{PUSH}(t) \mid \mathtt{ADD} \mid \mathtt{SUB} \mid \mathtt{MULT} \mid \\ &\quad\;\; \mathtt{EQ} \mid \mathtt{GT} \mid \mathtt{NOT} \mid \mathtt{AND} \mid \mathtt{OR} \mid \\ &\quad\;\; \mathtt{LOAD}(x) \mid \mathtt{STO}(x) \mid \mathtt{JMP}(k) \mid \mathtt{JMPF}(k) \end{aligned}$$

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

The Abstract Machine

Definition 4.4 (Abstract machine)

The abstract machine (AM) is given by

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

$$\begin{aligned} P &::= p^* \\ p &::= \texttt{PUSH(}z\texttt{)} \mid \texttt{PUSH(}t\texttt{)} \mid \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MULT} \mid \\ & \texttt{EQ} \mid \texttt{GT} \mid \texttt{NOT} \mid \texttt{AND} \mid \texttt{OR} \mid \\ & \texttt{LOAD(}x\texttt{)} \mid \texttt{STO(}x\texttt{)} \mid \texttt{JMP(}k\texttt{)} \mid \texttt{JMPF(}k\texttt{)} \end{aligned}$$

  (where $z, k \in \mathbb{Z}$, $t \in \mathbb{B}$, and $x \in \mathit{Var}$)
- configurations of the form $\langle pc, e, \sigma \rangle \in \mathit{Cnf}$ where
  - $pc \in \mathbb{Z}$ is the program counter (i.e., address of next instruction to be executed)
  - $e \in \mathit{Stk} := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top right)
  - $\sigma \in \Sigma := (\mathit{Var} \to \mathbb{Z})$ is the (storage) state

  (thus $\mathit{Cnf} = \mathbb{Z} \times \mathit{Stk} \times \Sigma$)

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

The Abstract Machine

Definition 4.4 (Abstract machine)

The abstract machine (AM) is given by

- programs $P \in Code$ and instructions p:
$$\begin{aligned} P &::= p^* \\ p &::= \texttt{PUSH}(z) \mid \texttt{PUSH}(t) \mid \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MULT} \mid \\ &\quad\ \texttt{EQ} \mid \texttt{GT} \mid \texttt{NOT} \mid \texttt{AND} \mid \texttt{OR} \mid \\ &\quad\ \texttt{LOAD}(x) \mid \texttt{STO}(x) \mid \texttt{JMP}(k) \mid \texttt{JMPF}(k) \end{aligned}$$
  (where $z, k \in \mathbb{Z}$, $t \in \mathbb{B}$, and $x \in Var$)
- configurations of the form $\langle pc, e, \sigma \rangle \in Cnf$ where
  - $pc \in \mathbb{Z}$ is the program counter (i.e., address of next instruction to be executed)
  - $e \in Stk := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top right)
  - $\sigma \in \Sigma := (Var \to \mathbb{Z})$ is the (storage) state

  (thus $Cnf = \mathbb{Z} \times Stk \times \Sigma$)
- initial configurations of the form $\langle 0, \varepsilon, \sigma \rangle$
- final configurations of the form $\langle |P|, e, \sigma \rangle$

Semantics of AM-Code I

Definition 4.5 (Transition relation of AM)

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

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

Semantics of AM-Code II

Corollary 4.6

$\triangleright$ is not total, i.e., there exists $\gamma \in \mathit{Cnf}$ such that

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

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

Semantics of AM-Code II

Corollary 4.6

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

$$\gamma \not\rhd \gamma'$$

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

Proof.

Possible cases:

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

$\square$

Alternative Choices

Remark: more realistic machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle pc, e, \mu \rangle$ with memory $\mu \in (\mathbb{N} \to \mathbb{Z})$
  - $\texttt{LOAD}(x)$/$\texttt{STO}(x)$ replaced by $\texttt{LOAD}(m)$/$\texttt{STO}(m)$ (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 $\gamma_0, \gamma_1, \ldots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \rhd \gamma_i$ for each $i \in \{1, \ldots, k\}$

Terminating and Looping Computations I

Definition 4.7 (AM computations)

- A finite computation is a finite configuration sequence of the form $\gamma_0, \gamma_1, \ldots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \rhd \gamma_i$ for each $i \in \{1, \ldots, k\}$
- If, in addition, there is no γ such that $\gamma_k \rhd \gamma$, then $\gamma_0, \gamma_1, \ldots, \gamma_k$ is called terminating

Terminating and Looping Computations I

Definition 4.7 (AM computations)

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

Terminating and Looping Computations I

Definition 4.7 (AM computations)

- A finite computation is a finite configuration sequence of the form $\gamma_0, \gamma_1, \ldots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \rhd \gamma_i$ for each $i \in \{1, \ldots, k\}$
- If, in addition, there is no γ such that $\gamma_k \rhd \gamma$, then $\gamma_0, \gamma_1, \ldots, \gamma_k$ is called terminating
- A looping computation is an infinite configuration sequence of the form $\gamma_0, \gamma_1, \gamma_2, \ldots$ where $\gamma_i \rhd \gamma_{i+1}$ 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 $P := 0{:}\texttt{LOAD(x)};1{:}\texttt{PUSH(1)};\ 2{:}\texttt{ADD};3{:}\texttt{STO(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$\langle 0, \varepsilon, \sigma \rangle$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma \rangle \rhd \langle 1, 3, \sigma \rangle$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, 3, \sigma\rangle \rhd \langle 2, 3:1, \sigma\rangle$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma \rangle \rhd \langle 1, 3, \sigma \rangle \rhd \langle 2, 3 : 1, \sigma \rangle \rhd \langle 3, 4, \sigma \rangle$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, 3, \sigma\rangle \rhd \langle 2, 3:1, \sigma\rangle \rhd \langle 3, 4, \sigma\rangle \rhd \langle 4, \varepsilon, \sigma[\mathtt{x} \mapsto 4]\rangle$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, 3, \sigma\rangle \rhd \langle 2, 3:1, \sigma\rangle \rhd \langle 3, 4, \sigma\rangle \rhd \langle 4, \varepsilon, \sigma[\texttt{x} \mapsto 4]\rangle$$

Remark: implements statement `x := x + 1`

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma \rangle \rhd \langle 1, 3, \sigma \rangle \rhd \langle 2, 3:1, \sigma \rangle \rhd \langle 3, 4, \sigma \rangle \rhd \langle 4, \varepsilon, \sigma[\texttt{x} \mapsto 4] \rangle$$

   Remark: implements statement `x := x + 1`

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

$$\langle 0, \varepsilon, \sigma \rangle$$

Terminating and Looping Computations II

Example 4.8

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

   $$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, 3, \sigma\rangle \rhd \langle 2, 3 : 1, \sigma\rangle \rhd \langle 3, 4, \sigma\rangle \rhd \langle 4, \varepsilon, \sigma[\texttt{x} \mapsto 4]\rangle$$

   Remark: implements statement `x := x + 1`
2. For $P := \texttt{0:PUSH(}\mathsf{true}\texttt{);1:JMPF(2);2:JMP(-2)}$, the following computation loops:

   $$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, \mathsf{true}, \sigma\rangle$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, 3, \sigma\rangle \rhd \langle 2, 3:1, \sigma\rangle \rhd \langle 3, 4, \sigma\rangle \rhd \langle 4, \varepsilon, \sigma[\texttt{x} \mapsto 4]\rangle$$

   Remark: implements statement `x := x + 1`

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, \mathsf{true}, \sigma\rangle \rhd \langle 2, \varepsilon, \sigma\rangle$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, 3, \sigma\rangle \rhd \langle 2, 3:1, \sigma\rangle \rhd \langle 3, 4, \sigma\rangle \rhd \langle 4, \varepsilon, \sigma[\texttt{x} \mapsto 4]\rangle$$

   Remark: implements statement `x := x + 1`

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, \mathsf{true}, \sigma\rangle \rhd \langle 2, \varepsilon, \sigma\rangle \rhd \langle 0, \varepsilon, \sigma\rangle \rhd \ldots$$

Terminating and Looping Computations II

Example 4.8

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, 3, \sigma\rangle \rhd \langle 2, 3:1, \sigma\rangle \rhd \langle 3, 4, \sigma\rangle \rhd \langle 4, \varepsilon, \sigma[\texttt{x} \mapsto 4]\rangle$$

   Remark: implements statement `x := x + 1`

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

$$\langle 0, \varepsilon, \sigma\rangle \rhd \langle 1, \mathsf{true}, \sigma\rangle \rhd \langle 2, \varepsilon, \sigma\rangle \rhd \langle 0, \varepsilon, \sigma\rangle \rhd \dots$$

   Remark: implements statement `while true do skip end`

Outline of Lecture 4

A New Inductive Principle

Application: Finite computations (Def. 4.7)

Definition: a finite computation $\gamma_0, \gamma_1, \ldots, \gamma_k$ 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 $P \vdash \langle pc, e, \sigma\rangle \rhd^* \langle pc', e', \sigma'\rangle$, *then*

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

for all $P_1, P_2 \in Code$ *and* $e_0 \in Stk$.

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

Properties of AM

Application: Extension of Code and Stack

Lemma 4.9

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

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

for all $P_1, P_2 \in Code$ *and* $e_0 \in Stk$.

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 $\gamma, \gamma', \gamma'' \in \mathit{Cnf}$,

$$\gamma \rhd \gamma' \text{ and } \gamma \rhd \gamma'' \text{ imply } \gamma' = \gamma''.$$

Another Property: Determinism

Lemma 4.10

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

$$\gamma \rhd \gamma' \textit{ and } \gamma \rhd \gamma'' \textit{ imply } \gamma' = \gamma''.$$

Proof (Idea).

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

Another Property: Determinism

Lemma 4.10

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

$$\gamma \rhd \gamma' \textit{ and } \gamma \rhd \gamma'' \textit{ imply } \gamma' = \gamma''.$$

Proof (Idea).

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

Thus the following function is well defined:

Definition 4.11 (Semantics of AM)

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

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