Semantics and Verification of Software

Lecture 12: Provably Correct Implementation I (Abstract Machine)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH AACHEN UNIVERSITY

noll@cs.rwth-aachen.de

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

Summer Semester 2013

Outline

Compiler Correctness

$$\begin{array}{rcl} \text{programming language} & \xrightarrow{\text{compiler}} & \text{machine code} \\ \text{semantics} \downarrow & & \downarrow \text{(simple) semantics} \\ \text{meaning} & \stackrel{?}{=} & \text{meaning} \end{array}$$

Compiler Correctness

$$\begin{array}{rcl} \text{programming language} & \xrightarrow{\text{compiler}} & \text{machine code} \\ \text{semantics} \downarrow & & \downarrow \text{(simple) semantics} \\ \text{meaning} & \stackrel{?}{=} & \text{meaning} \end{array}$$

To do:

1. Definition of abstract machine
2. Definition (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

Outline

The Abstract Machine

Definition 12.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in \mathit{Cnf}$ where
  - $d \in \mathit{Code}$ is the sequence of instructions (code) to be executed
  - $e \in \mathit{Stk} := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (\mathit{Var} \to \mathbb{Z})$ is the (storage) state

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

The Abstract Machine

Definition 12.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in Cnf$ where
  - $d \in Code$ is the sequence of instructions (code) to be executed
  - $e \in Stk := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (Var \to \mathbb{Z})$ is the (storage) state

  (thus $Cnf = Code \times Stk \times \Sigma$)
- initial configurations of the form $\langle d, \varepsilon, \sigma \rangle$

The Abstract Machine

Definition 12.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in \mathit{Cnf}$ where
  - $d \in \mathit{Code}$ is the sequence of instructions (code) to be executed
  - $e \in \mathit{Stk} := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (\mathit{Var} \to \mathbb{Z})$ is the (storage) state

  (thus $\mathit{Cnf} = \mathit{Code} \times \mathit{Stk} \times \Sigma$)
- initial configurations of the form $\langle d, \varepsilon, \sigma \rangle$
- final configurations of the form $\langle \varepsilon, e, \sigma \rangle$

The Abstract Machine

Definition 12.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in Cnf$ where
  - $d \in Code$ is the sequence of instructions (code) to be executed
  - $e \in Stk := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (Var \to \mathbb{Z})$ is the (storage) state

  (thus $Cnf = Code \times Stk \times \Sigma$)
- initial configurations of the form $\langle d, \varepsilon, \sigma \rangle$
- final configurations of the form $\langle \varepsilon, e, \sigma \rangle$
- code sequences d and instructions i:

$$\begin{aligned} d &::= \varepsilon \mid i : d \\ i &::= \texttt{PUSH}(z) \mid \texttt{ADD} \mid \texttt{MULT} \mid \texttt{SUB} \mid \\ &\quad\; \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{EQ} \mid \texttt{GT} \mid \texttt{AND} \mid \texttt{OR} \mid \texttt{NEG} \mid \\ &\quad\; \texttt{LOAD}(x) \mid \texttt{STORE}(x) \mid \texttt{NOOP} \mid \texttt{BRANCH}(d, d) \mid \texttt{LOOP}(d, d) \end{aligned}$$

  (where $z \in \mathbb{Z}$ and $x \in Var$)

Semantics of AM-Code

Definition 12.2 (Transition relation of AM)

The transition relation $\triangleright \subseteq Cnf \times Cnf$ is given by

$$
\begin{aligned}
\langle \texttt{PUSH(}z\texttt{)} : d, e, \sigma \rangle &\triangleright \langle d, z : e, \sigma \rangle \\
\langle \texttt{ADD} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 + z_2) : e, \sigma \rangle \\
\langle \texttt{MULT} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 * z_2) : e, \sigma \rangle \\
\langle \texttt{SUB} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 - z_2) : e, \sigma \rangle \\
\langle \texttt{TRUE} : d, e, \sigma \rangle &\triangleright \langle d, \mathsf{true} : e, \sigma \rangle \\
\langle \texttt{FALSE} : d, e, \sigma \rangle &\triangleright \langle d, \mathsf{false} : e, \sigma \rangle \\
\langle \texttt{EQ} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 = z_2) : e, \sigma \rangle \\
\langle \texttt{GT} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 > z_2) : e, \sigma \rangle \\
\langle \texttt{AND} : d, t_1 : t_2 : e, \sigma \rangle &\triangleright \langle d, (t_1 \wedge t_2) : e, \sigma \rangle \\
\langle \texttt{OR} : d, t_1 : t_2 : e, \sigma \rangle &\triangleright \langle d, (t_1 \vee t_2) : e, \sigma \rangle \\
\langle \texttt{NEG} : d, t : e, \sigma \rangle &\triangleright \langle d, \neg t : e, \sigma \rangle \\
\langle \texttt{LOAD(}x\texttt{)} : d, e, \sigma \rangle &\triangleright \langle d, \sigma(x) : e, \sigma \rangle \\
\langle \texttt{STORE(}x\texttt{)} : d, z : e, \sigma \rangle &\triangleright \langle d, e, \sigma[x \mapsto z] \rangle \\
\langle \texttt{NOOP} : d, e, \sigma \rangle &\triangleright \langle d, e, \sigma \rangle \\
\langle \texttt{BRANCH(}d_{\mathsf{true}}\texttt{,}d_{\mathsf{false}}\texttt{)} : d, t : e, \sigma \rangle &\triangleright \langle d_t : d, e, \sigma \rangle \\
\langle \texttt{LOOP(}d_1\texttt{,}d_2\texttt{)} : d, e, \sigma \rangle &\triangleright \langle d_1 : \texttt{BRANCH(}d_2 : \texttt{LOOP(}d_1\texttt{,}d_2\texttt{),NOOP)} : d, e, \sigma \rangle
\end{aligned}
$$

Alternative Choices

Remark: more traditional machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle d, e, m \rangle$ with memory $m \in \mathbb{Z}^*$
  - $\texttt{LOAD}(x)/\texttt{STORE}(x)$ replaced by $\texttt{GET}(n)/\texttt{PUT}(n)$ (where $n \in \mathbb{N}$)

Alternative Choices

Remark: more traditional machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle d, e, m \rangle$ with memory $m \in \mathbb{Z}^*$
  - $\texttt{LOAD}(x)/\texttt{STORE}(x)$ replaced by $\texttt{GET}(n)/\texttt{PUT}(n)$ (where $n \in \mathbb{N}$)
- BRANCH and LOOP instruction replaced by code addresses (labels) and jumping instructions
  - configurations $\langle pc, d, e, m \rangle$ with program counter $pc \in \mathbb{N}$
  - BRANCH and LOOP implemented by control flow, using $\texttt{JUMP}(l)$ and $\texttt{JUMPFALSE}(l)$ ($l \in \mathbb{N}$)

Alternative Choices

Remark: more traditional machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle d, e, m\rangle$ with memory $m \in \mathbb{Z}^*$
  - `LOAD(`x`)`/`STORE(`x`)` replaced by `GET(`n`)`/`PUT(`n`)` (where $n \in \mathbb{N}$)
- `BRANCH` and `LOOP` instruction replaced by code addresses (labels) and jumping instructions
  - configurations $\langle pc, d, e, m\rangle$ with program counter $pc \in \mathbb{N}$
  - `BRANCH` and `LOOP` implemented by control flow, using `JUMP(`l`)` and `JUMPFALSE(`l`)` ($l \in \mathbb{N}$)
- Registers for storing intermediate values (in place of evaluation stack e)

Terminating and Looping Computations

Definition 12.3 (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

Definition 12.3 (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

Definition 12.3 (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

Definition 12.3 (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: a terminating computation may end in a final configuration ($\langle \varepsilon, e, \sigma \rangle$) or in a stuck configuration (e.g., $\langle \mathtt{ADD}, 1, \sigma \rangle$)

A Terminating Computation

Example 12.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$\langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle$$

A Terminating Computation

Example 12.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$\langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle$$

A Terminating Computation

Example 12.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$\begin{aligned} & \langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\ \rhd \quad & \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\ \rhd \quad & \langle \texttt{ADD:STORE(x)}, 3 : 1, \sigma \rangle \end{aligned}$$

A Terminating Computation

Example 12.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$
\begin{aligned}
& \langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\
\rhd\quad & \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\
\rhd\quad & \langle \texttt{ADD:STORE(x)}, 3 : 1, \sigma \rangle \\
\rhd\quad & \langle \texttt{STORE(x)}, 4, \sigma \rangle
\end{aligned}
$$

A Terminating Computation

Example 12.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$
\begin{aligned}
& \langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\
\rhd \quad & \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\
\rhd \quad & \langle \texttt{ADD:STORE(x)}, 3 : 1, \sigma \rangle \\
\rhd \quad & \langle \texttt{STORE(x)}, 4, \sigma \rangle \\
\rhd \quad & \langle \varepsilon, \varepsilon, \sigma[\texttt{x} \mapsto 4] \rangle
\end{aligned}
$$

A Terminating Computation

Example 12.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$
\begin{aligned}
&\langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\
&\quad \rhd \quad \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\
&\quad \rhd \quad \langle \texttt{ADD:STORE(x)}, 3:1, \sigma \rangle \\
&\quad \rhd \quad \langle \texttt{STORE(x)}, 4, \sigma \rangle \\
&\quad \rhd \quad \langle \varepsilon, \varepsilon, \sigma[\texttt{x} \mapsto 4] \rangle
\end{aligned}
$$

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

A Looping Computation

Example 12.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$

A Looping Computation

Example 12.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$

$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle$$

A Looping Computation

Example 12.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle$$
$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma\rangle$$
$$\rhd \quad \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma\rangle$$

A Looping Computation

Example 12.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle$$
$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma\rangle$$
$$\rhd \quad \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma\rangle$$
$$\rhd \quad \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle$$

A Looping Computation

Example 12.5

The following computation loops:

$$\begin{array}{ll} & \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle \\ \rhd & \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle \\ \rhd & \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma \rangle \\ \rhd & \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle \\ \rhd & \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle \end{array}$$

A Looping Computation

Example 12.5

The following computation loops:

$$\begin{array}{ll} & \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle \\ \rhd & \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma\rangle \\ \rhd & \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma\rangle \\ \rhd & \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle \\ \rhd & \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle \\ \rhd & \ldots \end{array}$$

A Looping Computation

Example 12.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma \rangle$$
$$\rhd \quad \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \ldots$$

Remark: implements statement `while true do skip`

Outline

A New Inductive Principle

Application: Finite computations (Def. 12.3)

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 $k + 1$

Application: Extension of Code and Stack

Lemma 12.6

If $\langle d_1, e_1, \sigma \rangle \rhd^* \langle d', e', \sigma' \rangle$, *then*

$$\langle d_1 : d_2, e_1 : e_2, \sigma \rangle \rhd^* \langle d' : d_2, e' : e_2, \sigma' \rangle$$

for every $d_2 \in \mathit{Code}$ *and* $e_2 \in \mathit{Stk}$.

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

Application: Extension of Code and Stack

Lemma 12.6

If $\langle d_1, e_1, \sigma\rangle \triangleright^* \langle d', e', \sigma'\rangle$, *then*

$$\langle d_1 : d_2, e_1 : e_2, \sigma\rangle \triangleright^* \langle d' : d_2, e' : e_2, \sigma'\rangle$$

for every $d_2 \in Code$ *and* $e_2 \in Stk$.

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

Proof.

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

Another Property: Determinism

Lemma 12.7

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 12.7

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

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

Proof.

The successor configuration is determined by the first instruction in the code component, which is unique. □

Another Property: Determinism

Lemma 12.7

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

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

Proof.

The successor configuration is determined by the first instruction in the code component, which is unique. □

Thus the following function is well defined:

Definition 12.8 (Semantics of AM)

The semantics of an instruction sequence is given by the mapping

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

defined by

$$\mathfrak{M}[\![d]\!](\sigma) := \begin{cases} \sigma' & \text{if } \langle d, \varepsilon, \sigma \rangle \rhd^* \langle \varepsilon, e, \sigma' \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$