

Semantics and Verification of Software

Lecture 12: Provably Correct Implementation I (Abstract Machine)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

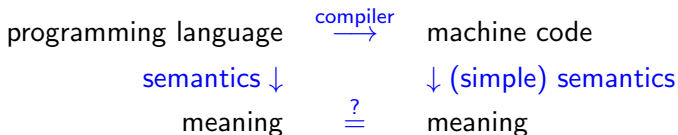


noll@cs.rwth-aachen.de

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

Summer Semester 2013

- 1 Compiler Correctness
- 2 The Abstract Machine
- 3 Properties of AM



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

- 1 Compiler Correctness
- 2 The Abstract Machine
- 3 Properties of AM

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)
 - $\sigma \in \Sigma := (Var \rightarrow \mathbb{Z})$ 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 :
 $d ::= \varepsilon \mid i : d$
 $i ::= PUSH(z) \mid ADD \mid MULT \mid SUB \mid$
 $TRUE \mid FALSE \mid EQ \mid GT \mid AND \mid OR \mid NEG \mid$
 $LOAD(x) \mid STORE(x) \mid NOOP \mid BRANCH(d, d) \mid LOOP(d, d)$

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

Definition 12.2 (Transition relation of AM)

The **transition relation** $\triangleright \subseteq \text{Cnf} \times \text{Cnf}$ is given by

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

Remark: more traditional machine architectures

- **Variables referenced by address** (and not by name)
 - configurations $\langle d, e, m \rangle$ with **memory** $m \in \mathbb{Z}^*$
 - $\text{LOAD}(x)/\text{STORE}(x)$ replaced by $\text{GET}(n)/\text{PUT}(n)$ (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 $\text{JUMP}(l)$ and $\text{JUMPFALSE}(l)$ ($l \in \mathbb{N}$)
- **Registers** for storing intermediate values
(in place of evaluation stack e)

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

Example 12.4

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

- $\langle \text{PUSH}(1) : \text{LOAD}(x) : \text{ADD} : \text{STORE}(x), \varepsilon, \sigma \rangle$
 - ▷ $\langle \text{LOAD}(x) : \text{ADD} : \text{STORE}(x), 1, \sigma \rangle$
 - ▷ $\langle \text{ADD} : \text{STORE}(x), 3 : 1, \sigma \rangle$
 - ▷ $\langle \text{STORE}(x), 4, \sigma \rangle$
 - ▷ $\langle \varepsilon, \varepsilon, \sigma[x \mapsto 4] \rangle$

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

Example 12.5

The following computation loops:

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

- ▷ $\langle \text{TRUE}:\text{BRANCH}(\text{NOOP}:\text{LOOP}(\text{TRUE}, \text{NOOP}), \text{NOOP}), \varepsilon, \sigma \rangle$
- ▷ $\langle \text{BRANCH}(\text{NOOP}:\text{LOOP}(\text{TRUE}, \text{NOOP}), \text{NOOP}), \text{true}, \sigma \rangle$
- ▷ $\langle \text{NOOP}:\text{LOOP}(\text{TRUE}, \text{NOOP}), \varepsilon, \sigma \rangle$
- ▷ $\langle \text{LOOP}(\text{TRUE}, \text{NOOP}), \varepsilon, \sigma \rangle$
- ▷ ...

Remark: implements statement `while true do skip`

- 1 Compiler Correctness
- 2 The Abstract Machine
- 3 Properties of AM

A New Inductive Principle

Application: Finite computations (Def. 12.3)

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 $k + 1$

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 $d_2 \in \text{Code}$ and $e_2 \in \text{Stk}$.

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 $\gamma, \gamma', \gamma'' \in \text{Cnf}$,
 $\gamma \triangleright \gamma'$ and $\gamma \triangleright \gamma''$ 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

$$\mathfrak{M}[\![\cdot]\!] : \text{Code} \rightarrow (\Sigma \dashrightarrow \Sigma),$$

defined by

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