

Semantics and Verification of Software

Lecture 24: Provably Correct Implementation I (Abstract Machine & Compiler)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

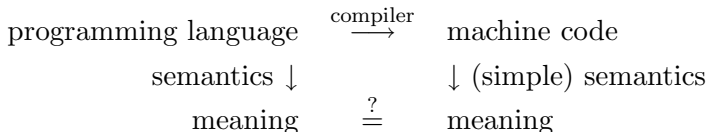
RWTH Aachen University

`noll@cs.rwth-aachen.de`

`http://www-i2.informatik.rwth-aachen.de/i2/svsw08/`

Winter semester 2008/09

- 1 Introduction
- 2 The Abstract Machine
- 3 Properties of AM
- 4 The Compiler



To do:

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

- 1 Introduction
- 2 The Abstract Machine
- 3 Properties of AM
- 4 The Compiler

Definition 24.1

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**
 - $\sigma \in \Sigma := (Var \rightarrow \mathbb{Z})$ is the **(storage) state**

(thus $Cnf = Code \times Stk \times \Sigma$)

- **final configurations** of the form $\langle \varepsilon, e, \sigma \rangle$
- **code sequences** and **instructions**:

$$d ::= \varepsilon \mid i : d$$

$$i ::= \text{PUSH}(z) \mid \text{ADD} \mid \text{MULT} \mid \text{SUB} \mid \\ \text{TRUE} \mid \text{FALSE} \mid \text{EQ} \mid \text{GT} \mid \text{AND} \mid \text{OR} \mid \text{NEG} \mid \\ \text{LOAD}(x) \mid \text{STORE}(x) \mid \text{NOOP} \mid \text{BRANCH}(d, d) \mid \text{LOOP}(d, d)$$

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

Definition 24.2

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 **JUMP(l)** and **JUMPFALSE(l)** ($l \in \mathbb{N}$)
- **Registers** for storing intermediate values (in place of evaluation stack e)

Definition 24.3

- A **terminating computation** is a finite configuration sequence of the form $\gamma_0, \gamma_1, \dots, \gamma_k$ where
 - $\gamma_0 = \langle d, \varepsilon, \sigma \rangle$
 - $\gamma_{i-1} \triangleright \gamma_i$ for each $i \in k$ ($k \in \mathbb{N}$)
 - there is no γ such that $\gamma_k \triangleright \gamma$
- A **looping computation** is an infinite configuration sequence of the form $\gamma_0, \gamma_1, \gamma_2, \dots$ where
 - $\gamma_0 = \langle d, \varepsilon, \sigma \rangle$
 - $\gamma_i \triangleright \gamma_{i+1}$ for each $i \in \mathbb{N}$
 - there is no γ such that $\gamma_k \triangleright \gamma$

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 24.4

For $d := \text{PUSH}(1) : \text{LOAD}(\mathbf{x}) : \text{ADD} : \text{STORE}(\mathbf{x})$ and $\sigma(\mathbf{x}) = 3$:

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

▷ $\langle \text{LOAD}(\mathbf{x}) : \text{ADD} : \text{STORE}(\mathbf{x}), 1, \sigma \rangle$

▷ $\langle \text{ADD} : \text{STORE}(\mathbf{x}), 3 : 1, \sigma \rangle$

▷ $\langle \text{STORE}(\mathbf{x}), 4, \sigma \rangle$

▷ $\langle \varepsilon, \varepsilon, \sigma[\mathbf{x} \mapsto 4] \rangle$

Example 24.5

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

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

- 1 Introduction
- 2 The Abstract Machine
- 3 Properties of AM
- 4 The Compiler

Application: Computation sequences (Def. 24.3)

- Definition:
- for each $\gamma \in Cnf$, γ is a **computation sequence** (of length 0)
 - whenever $\gamma_0, \gamma_1, \dots, \gamma_k$ is a computation sequence and $\gamma_k \triangleright \gamma_{k+1}$, then $\gamma_0, \gamma_1, \dots, \gamma_k, \gamma_{k+1}$ is a **computation sequence** (of length $k + 1$)

Induction base: property holds for all computation sequences of length 0

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

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

Lemma 24.6

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

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 sequence
(on the board) □

Another Property: Determinism

Lemma 24.7

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 24.8

The *semantics of an instruction sequence* is given by the mapping

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

defined by

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

- 1 Introduction
- 2 The Abstract Machine
- 3 Properties of AM
- 4 The Compiler

Repetition: Syntax of WHILE Programs

Definition (Syntax of WHILE (Def. 1.2))

The **syntax of WHILE programs** is defined by the following context-free grammar:

$$a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$$
$$b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$$
$$c ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \in Cmd$$

Definition 24.9

The translation function

$$\mathfrak{T}_a[\cdot] : AExp \rightarrow Code$$

is given by

$$\begin{aligned}\mathfrak{T}_a[z] &:= \text{PUSH}(z) \\ \mathfrak{T}_a[x] &:= \text{LOAD}(x) \\ \mathfrak{T}_a[a_1 + a_2] &:= \mathfrak{T}_a[a_2] : \mathfrak{T}_a[a_1] : \text{ADD} \\ \mathfrak{T}_a[a_1 - a_2] &:= \mathfrak{T}_a[a_2] : \mathfrak{T}_a[a_1] : \text{SUB} \\ \mathfrak{T}_a[a_1 * a_2] &:= \mathfrak{T}_a[a_2] : \mathfrak{T}_a[a_1] : \text{MULT}\end{aligned}$$

Example 24.10

$$\begin{aligned}\mathfrak{T}_a[x + 1] &= \mathfrak{T}_a[1] : \mathfrak{T}_a[x] : \text{ADD} \\ &= \text{PUSH}(1) : \text{LOAD}(x) : \text{ADD}\end{aligned}$$

Definition 24.11

The translation function

$$\mathfrak{T}_b[\cdot] : BExp \rightarrow Code$$

is given by

$$\begin{aligned}\mathfrak{T}_b[\text{true}] &:= \text{TRUE} \\ \mathfrak{T}_b[\text{false}] &:= \text{FALSE} \\ \mathfrak{T}_b[a_1 = a_2] &:= \mathfrak{T}_a[a_2] : \mathfrak{T}_a[a_1] : \text{EQ} \\ \mathfrak{T}_b[a_1 > a_2] &:= \mathfrak{T}_a[a_2] : \mathfrak{T}_a[a_1] : \text{GT} \\ \mathfrak{T}_b[\neg b] &:= \mathfrak{T}_b[b] : \text{NEG} \\ \mathfrak{T}_b[b_1 \wedge a_2] &:= \mathfrak{T}_b[b_2] : \mathfrak{T}_b[b_1] : \text{AND} \\ \mathfrak{T}_b[b_1 \vee a_2] &:= \mathfrak{T}_b[b_2] : \mathfrak{T}_b[b_1] : \text{OR}\end{aligned}$$

Definition 24.12

The translation function $\mathfrak{T}_c[\cdot] : \text{Cmd} \rightarrow \text{Code}$ is given by

$$\begin{aligned}\mathfrak{T}_c[\text{skip}] &:= \text{NOOP} \\ \mathfrak{T}_c[x := a] &:= \mathfrak{T}_a[a] : \text{STORE}(x) \\ \mathfrak{T}_c[c_1; c_2] &:= \mathfrak{T}_c[c_1] : \mathfrak{T}_c[c_2] \\ \mathfrak{T}_c[\text{if } b \text{ then } c_1 \text{ else } c_2] &:= \mathfrak{T}_b[b] : \text{BRANCH}(\mathfrak{T}_c[c_1], \mathfrak{T}_c[c_2]) \\ \mathfrak{T}_c[\text{while } b \text{ do } c] &:= \text{LOOP}(\mathfrak{T}_b[b], \mathfrak{T}_c[c])\end{aligned}$$

Example 24.13 (Factorial program)

$$\begin{aligned}\mathfrak{T}_c[y:=1; \text{while } \neg(x=1) \text{ do } (y:=y*x; x:=x-1)] \\ &= \mathfrak{T}_c[y:=1] : \mathfrak{T}_c[\text{while } \neg(x=1) \text{ do } (y:=y*x; x:=x-1)] \\ &= \mathfrak{T}_a[1] : \text{STORE}(y) : \text{LOOP}(\mathfrak{T}_b[\neg(x=1)], \mathfrak{T}_c[y:=y*x; x:=x-1]) \\ &= \text{PUSH}(1) : \text{STORE}(y) : \text{LOOP}(\mathfrak{T}_b[x=1] : \text{NEG}, \mathfrak{T}_c[y:=y*x] \mathfrak{T}_c[x:=x-1]) \\ &\vdots \\ &= \text{PUSH}(1) : \text{STORE}(y) : \text{LOOP}(\text{PUSH}(1) : \text{LOAD}(x) : \text{EQ} : \text{NEG}, \\ &\quad \text{LOAD}(x) : \text{LOAD}(y) : \text{MULT} : \text{STORE}(y) : \\ &\quad \text{PUSH}(1) : \text{LOAD}(x) : \text{SUB} : \text{STORE}(x))\end{aligned}$$