Semantics and Verification of Software

Lecture 16: 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/svsw10/

Summer Semester 2010

Outline

Compiler Correctness

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

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

Outline

Definition 16.1 (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
  - $\sigma \in \Sigma := (\mathit{Var} \to \mathbb{Z})$ is the (storage) state

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

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

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

Definition 16.2 (Transition relation of AM)

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

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

Alternative Choices

Remark: more traditional machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle d, e, m \rangle$ with memory $m \in \mathbb{Z}^*$
  - 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 16.3 (AM computations)

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

A Terminating Computation

Example 16.4

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

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

A Looping Computation

Example 16.5

The following computation loops:

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

Outline

A New Inductive Principle

Application: Computation sequences (Def. 16.3)

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

Application: Extension of Code and Stack

Lemma 16.6

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

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

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

Proof.

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

Another Property: Determinism

Lemma 16.7

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

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

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 16.8 (Semantics of AM)

The semantics of an instruction sequence is given by the mapping

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

defined by

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

Outline

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:

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

Translation of Arithmetic Expressions

Definition 16.9 (Translation of arithmetic expressions)

The translation function

$$\mathfrak{T}_a[\![.]\!] : AExp \to Code$$

is given by

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

Example 16.10

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

Definition 16.11 (Translation of Boolean expressions)

The translation function

$$\mathfrak{T}_b[\![.]\!] : \mathit{BExp} \to \mathit{Code}$$

is given by

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

Translation of Statements

Definition 16.12 (Translation of statements)

The translation function $\mathfrak{T}_c[\![.]\!] : \mathit{Cmd} \to \mathit{Code}$ is given by

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

Example 16.13 (Factorial program)

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