Programming Languages and Compiler Design

Provably Correct Implementation

Yliès Falcone, Jean-Claude Fernandez

Master of Sciences in Informatics at Grenoble (MoSIG)
University of Grenoble-Alpes
(Université Joseph Fourier, Grenoble INP)

Academic Year 2014 - 2015

Outline - Provably Correct Implementation

Provably correct Implementation/Code Generation

Using an operational semantics to argue about the correctness of its implementation.

We will see:

- how to define an operational semantics for an abstract machine: a machine with an evaluation stack;
- how to specify a code generator for such a machine (translation functions on the syntax of language **While**);
- how to use the source and target language semantics to prove that the code generation is correct.

Correctness

- Translate the program into code.
- Execute the code on the abstract machine.

$\rightarrow$We get the *"same result"*.

Outline - Provably Correct Implementation

Abstract machine AM: short overview

Machine AM is defined by a transition system.

Configurations are 3-tuples of the form (c, s, m):

- c: an *instruction list* $instr_1, \ldots, instr_n$
  $\hookrightarrow$ the remaining **code** to execute
- s: an evaluation *stack*
  $\hookrightarrow$ used to evaluate expressions
- m: a *storage*, i.e., a memory content

Transition relation $\triangleright$:

$$(c, s, m) \triangleright (c', s', m')$$

Remarks

- AM has no registers.
- Every internal computations is performed in/using the stack.

The instruction set: description

Instruction	**Effect**
push-n, True, False	push constant n,**tt**,**ff**
fetch(x) store(x)	push current value of x pop and assign the top of stack to x
add sub,mult,and,le,equal,neg	replace the 2 top-most stack elements by their sum similar
branch(c_1,c_2)	if the top of the stack is **tt** execute c_1 if it is **ff** then execute c_2 else deadlock
noop	skip
loop(c_1,c_2)	execute c_1, then, if the top of stack is **tt**, execute c_2 followed by loop(c_1,c_2) if it's **ff** then noop

Refining the ingredients

A target program is a word on the instruction alphabet.

Instruction list: $c \in \mathbf{Code}$

Code denotes the syntactic category of program instructions:

$$\begin{array}{rl} \mathit{inst} ::= & \text{push-n} \mid \text{add} \mid \text{sub} \mid \text{mult} \\ & \mid \text{True} \mid \text{False} \mid \text{and} \mid \text{le} \mid \text{equal} \mid \text{neg} \\ & \mid \text{branch}(c, c) \mid \text{loop}(c, c) \mid \text{noop} \\ c \in \mathbf{Code} ::= & \epsilon \mid \mathit{inst} \cdot c \end{array}$$

Evaluation stack: $s \in \mathbf{Stack}$

- Used to evaluate arithmetic and Boolean expressions.
- A list of values: $\mathbf{Stack} = (\mathbb{Z} \cup \mathbb{B})^*$.

Storage m

- Represents the memory content, i.e., value of variables: a *state*.
- A function from the variables to $\mathbb{Z}$: $\mathbf{State} = \mathbf{Var} \overset{part.}{\to} \mathbb{Z}$.

Semantics of instructions: an operational semantics

A configuration of AM is (c, s, m) where:

- $c \in \mathbf{Code}$ is a target program,
- $s \in \mathbf{Stack}$ is a stack content, i.e., a word on $\mathbb{Z} \cup \mathbb{B}$,
- $m \in \mathbf{State}$ is the memory content.

Final configurations are of the form (ϵ, s, m).

Relation $\triangleright$ is inductively defined:

$$
\begin{aligned}
(\text{push-n} \cdot c, s, m) &\triangleright (c, \mathcal{N}[n] \cdot s, m) \\
(\text{True} \cdot c, s, m) &\triangleright (c, \mathbf{tt} \cdot s, m) \\
(\text{False} \cdot c, s, m) &\triangleright (c, \mathbf{ff} \cdot s, m) \\
(\text{fetch}(x) \cdot c, s, m) &\triangleright (c, m(x) \cdot s, m) \\
(\text{store}(x) \cdot c, v \cdot s, m) &\triangleright (c, s, m[x \mapsto v]) \quad \text{if } v \in \mathbb{Z}
\end{aligned}
$$

Semantics of instructions (2)

$$
\begin{aligned}
(\mathsf{add} \cdot c, v_1 \cdot v_2 \cdot s, m) &\triangleright (c, (v_1 + v_2) \cdot s, m) \quad \text{if } v_1, v_2 \in \mathbb{Z} \\
(\mathsf{sub} \cdot c, v_1 \cdot v_2 \cdot s, m) &\triangleright (c, (v_1 - v_2) \cdot s, m) \quad \text{if } v_1, v_2 \in \mathbb{Z} \\
(\mathsf{mult} \cdot c, v_1 \cdot v_2 \cdot s, m) &\triangleright (c, (v_1 * v_2) \cdot s, m) \quad \text{if } v_1, v_2 \in \mathbb{Z}
\end{aligned}
$$

$$
\begin{aligned}
(\mathsf{le} \cdot c, v_1 \cdot v_2 \cdot s, m) &\triangleright (c, (v_1 \leq v_2) \cdot s, m) \quad \text{if } v_1, v_2 \in \mathbb{Z} \\
(\mathsf{equal} \cdot c, v_1 \cdot v_2 \cdot s, m) &\triangleright (c, (v_1 = v_2) \cdot s, m) \quad \text{if } v_1, v_2 \in \mathbb{Z} \\
(\mathsf{and} \cdot c, b_1 \cdot b_2 \cdot s, m) &\triangleright (c, (b_1 \wedge b_2) \cdot s, m) \quad \text{if } b_1, b_2 \in \mathbb{B}
\end{aligned}
$$

$$
(\mathsf{neg} \cdot c, b \cdot s, m) \triangleright (c, (\neg b) \cdot s, m) \quad \text{if } b \in \mathbb{B}
$$

$$
\begin{aligned}
(\mathsf{branch}(c_1, c_2) \cdot c, \mathbf{tt} \cdot s, m) &\triangleright (c_1 \cdot c, s, m) \\
(\mathsf{branch}(c_1, c_2) \cdot c, \mathbf{ff} \cdot s, m) &\triangleright (c_2 \cdot c, s, m)
\end{aligned}
$$

$$
\begin{aligned}
(\mathsf{noop} \cdot c, s, m) &\triangleright (c, s, m) \\
(\mathsf{loop}(c_1, c_2) \cdot c, s, m) &\triangleright \\
&\qquad (c_1 \cdot \mathsf{branch}(c_2 \cdot \mathsf{loop}(c_1, c_2), \mathsf{noop}) \cdot c, s, m)
\end{aligned}
$$

About the semantics of the Abstract Machine

This is "close" to a *structural* operational semantics

- execution is done "step by step", and
- semantics defines the execution of individual instructions.

Definition (Computation sequence)

Given $c \in \textbf{Code}$, $m \in \textbf{State}$, a computation sequence for c on σ is either:

- a *finite* sequence $\gamma_0, \gamma_1, \ldots, \gamma_k$ of configurations s.t.
  - $\gamma_0 = (c, \epsilon, m)$
  - $\forall i \in [0, k[: \gamma_i \triangleright \gamma_{i+1}$
  - $\nexists \gamma : \gamma_k \triangleright \gamma$
- an *infinite* sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$ of configurations s.t.
  - $\gamma_0 = (c, \epsilon, m)$
  - $\forall i \geq 0 : \gamma_i \triangleright \gamma_{i+1}$

About the semantics of the Abstract Machine

Terminology:

A computation sequence may be either

- **terminating** iff it is finite
- **looping** iff it is infinite

A terminating computation sequence may end

- in a *terminal configuration* (i.e., with an empty code component)
- in a *stuck configuration* (i.e., for which there is no derivation)

Example

- terminating computation: $(\mathsf{noop}, \epsilon, m) \triangleright (\epsilon, \epsilon, m)$
- looping computation:
  $(\mathsf{loop}(\mathsf{True}, \mathsf{noop}), \epsilon, m) \triangleright^* (\mathsf{loop}(\mathsf{True}, \mathsf{noop}), \epsilon, m) \triangleright^* \ldots$
- terminal configuration: $(\epsilon, \mathbf{n1} \cdot \mathbf{b1} \cdot \mathbf{b2}, m)$
- stuck configuration: $(\mathsf{add}, \epsilon, m)$

Some exercises

Exercise: computing an execution

Compute the execution of $\text{push-1} \cdot \text{fetch}(x) \cdot \text{add} \cdot \text{store}(x)$ in $m = [x \mapsto 3]$.

Exercise: computing an execution

Compute the execution of $\text{loop}(\text{True}, \text{noop})$ in any memory m.

Abstracting machine code

Game: what is the function computed by this machine code?

$$
\begin{array}{l}
\mathsf{push\text{-}0} \cdot \mathsf{store}(z) \cdot \mathsf{fetch}(x) \cdot \mathsf{store}(r) \\
\mathsf{loop}(\mathsf{fetch}(r) \cdot \mathsf{fetch}(y) \cdot \mathsf{le}, \\
\qquad \mathsf{fetch}(y) \cdot \mathsf{fetch}(r) \cdot \mathsf{sub} \cdot \mathsf{store}(r) \cdot \\
\qquad \mathsf{push\text{-}1} \cdot \mathsf{fetch}(z) \cdot \mathsf{sub} \cdot \mathsf{store}(z) \\
)
\end{array}
$$

Outline

Proof technique for AM

Semantics of AM is close in spirit to SOS:
$\hookrightarrow$ concerned with execution of the individual steps

Induction on the length of computation sequences

In order to prove a given property Prop for all computation sequences:

- *prove* that Prop holds for all computation sequences of length 0;
- *prove* Prop holds for all other computation sequences:
  - *Assume* Prop holds for all computations of length at most k,
    $\hookrightarrow$ **Induction Hypothesis**
  - *Prove* Prop holds for all computations of length $k+1$.

Some properties of AM

Code and stack contents can be extended

$(c_1, s_1, m_1) \triangleright^k (c_2, s_2, m_2)$ implies $(c_1 \cdot c, s_1 \cdot s, m_1) \triangleright^k (c_2 \cdot c, s_2 \cdot s, m_2)$

Proof.

By induction on k. $\square$

Code can be decomposed and composed

$(c_1 \cdot c_2, s, m) \triangleright^k (\epsilon, s_2, m_2)$

implies

$$\exists k' \in \mathbb{N}, \exists (\epsilon, s', m') \in \textbf{Config} :$$
$$(c_1, s, m) \triangleright^{k'} (\epsilon, s', m') \wedge (c_2, s', m') \triangleright^{k-k'} (\epsilon, s_2, m_2)$$

Proof.

By induction on k. $\square$

Relation $\triangleright$ is deterministic

$(c, s, m) \triangleright (c_1, s_1, m_1) \wedge (c, s, m) \triangleright (c_2, s_2, m_2)$

implies

$(c_1, s_1, m_1) = (c_2, s_2, m_2)$

Proof.

By induction on the length of c. $\square$

Semantics of a target program

We define semantic function (referred to as the execution function):

$$\mathcal{M} : \mathbf{Code} \rightarrow (\mathbf{State} \stackrel{part.}{\rightarrow} \mathbf{State}).$$

$$\mathcal{M}[c]m = \begin{cases} m' & (c, \epsilon, m) \triangleright^* (\epsilon, s, m') \\ \text{undef} & \text{otherwise} \end{cases}$$

Remarks

- It is a well-defined function (because of determinism).
- In the terminal configuration:
  - code component must be empty,
  - stack component is not required to be empty.

Outline - Provably Correct Implementation

Outline - Provably Correct Implementation

Code Generation: the problem

How can we define an automatic and systematic translation from **While** to **Code**?

We define 3 functions:

1. $\mathcal{CA}$: **Aexp** $\rightarrow$ **Code**
2. $\mathcal{CB}$: **Bexp** $\rightarrow$ **Code**
3. $\mathcal{CS}$: **Stm** $\rightarrow$ **Code**

s.t. the generated code "mimics" the semantics of $S_{ns}[\,]$.

To do so:

- we do not distinguish m and σ anymore
- we prove that $\mathcal{CA}$, $\mathcal{CB}$ and $\mathcal{CS}$ verify the following properties:
  1. $(\mathcal{CA}[a], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[a]\sigma, \sigma)$,
  2. $(\mathcal{CB}[b], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[b]\sigma, \sigma)$,
  3. $(\mathcal{CS}[S], \epsilon, \sigma) \triangleright^* (\epsilon, \epsilon, \sigma')$ iff $(S, \sigma) \rightarrow \sigma'$.

Code generation for arithmetical and Boolean expressions

Examples of clauses to define $\mathcal{CA}$:

- $\mathcal{CA}[n] =$ push-n.
- $\mathcal{CA}[x] =$ fetch(x)

Examples of clauses to define $\mathcal{CB}$:

- $\mathcal{CB}[\text{true}] =$ True
- $\mathcal{CB}[\neg b] = \mathcal{CB}[b] \cdot$ neg

Exercise

Give the complete definition of code-generation functions $\mathcal{CA}$ and $\mathcal{CB}$.

Exercise

Calculate the code for:

- arithmetical expressions: $x + 1$, $2 * x$
- Boolean expression $2 * x = 5 * y$

Code generation for statements

Examples of clauses to define $\mathcal{CS}$:

- $\mathcal{CS}[x := a] = \mathcal{CA}[a] \cdot \text{store}(x)$
- $\mathcal{CS}[S_1; S_2] = \mathcal{CS}[S_1] \cdot \mathcal{CS}[S_2]$

Exercise

Complete the definition of code generation function $\mathcal{CS}$.

Example of code generation for a program/statement

Exercise

Give the target code obtained when translating the *factorial program*

$$y := 1; \text{while } \neg(x = 1) \ \text{ do } y := y * x; x := x - 1 \text{ od}$$

Outline - Provably Correct Implementation

Proving the correctness of the code generation ?

Several intermediate steps.

Correctness for arithmetical expressions

$\forall a \in \mathbf{Aexp} : (\mathcal{CA}[a], \epsilon, \sigma) \rhd^* (\epsilon, \mathcal{A}[a]\sigma, \sigma)$

Proof.

By structural induction on $a \in \mathbf{Aexp}$. □

Correctness for Boolean expressions

$\forall b \in \mathbf{Bexp} : (\mathcal{CB}[b], \epsilon, \sigma) \rhd^* (\epsilon, \mathcal{B}[b]\sigma, \sigma)$

Proof.

By structural induction on $b \in \mathbf{Bexp}$. □

Correctness for statements:

$\forall S \in \mathbf{Stm}, \forall \sigma, \sigma' \in \mathbf{State}$:

1. $(S, \sigma) \rightarrow \sigma'$ implies $(\mathcal{CS}[S], \epsilon, \sigma) \rhd^* (\epsilon, \epsilon, \sigma')$
2. $(\mathcal{CS}[S], \epsilon, \sigma) \rhd^k (\epsilon, e, \sigma')$ implies $(S, \sigma) \rightarrow \sigma'$ and $e = \epsilon$

Proof.

1. by induction on the shape of the derivation tree for $(S, \sigma) \rightarrow \sigma'$
2. by induction on k, the length of the computation sequence.

Correctness of code generation

Meaning of a statement on the abstract machine:

$$\begin{array}{lcl} \mathcal{S}_{am} & : & \mathbf{Stm} \rightarrow (\mathbf{State} \overset{part.}{\rightarrow} \mathbf{State}) \\ \mathcal{S}_{am}[S] & = & \mathcal{M} \circ \mathcal{CS}(S) \end{array}$$

Correctness of code generation

For any program $S \in \mathbf{Stm}$:

$$\mathcal{S}_{ns}[S] = \mathcal{M} \circ \mathcal{CS}[S]$$

Outline - Provably Correct Implementation

Summary - Provably Correct Implementation

Summary - Provably Correct Implementation

- Definition of abstract machine AM.
  - (list of) instructions to be executed,
  - (evaluation) stack,
  - memory.
- Translation from **While** to **Code**.
- AM plus the translation function provides a provably-correct implementation of the NOS of **While**.