Provably correct implementation

William Steingartner

william.steingartner@tuke.sk

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia

Semantics of programming languages

Spring 2022/2023

Theme 05 – Provably correct implementation

Correct implementation

Structural operational semantics is useful when **implementing language**.

A **correct implementation** of program consists of:

- the definition of **abstract machine**,
- the definition of the **meaning** of the abstract machine instructions by an operational semantics,
- the definition of **translation functions** that will map expressions and statements in the *Jane* language into sequences of such instructions.

The correctness result will then state that if we translate a program into code, and execute the code on the abstract machine, then we get the same result as we specified by the semantic functions $\mathscr{S}_{ns}$ and $\mathscr{S}_{os}$.

The **definition of abstract machine** consists of:

- the **configuration** of abstract machine,
- the definition of **instructions** and their:
    - **syntax** definition, and
    - **operational semantics** of instructions.

Abstract machine configuration

The abstract machine AM has **configuration** of the form:

$$\langle c, st, s \rangle$$

where

- c is a **code**, i.e. the sequence of instructions to be executed,
- st is the **evaluation stack**, and
- s is the **storage**, expressed by state $s \in \mathbf{State}$ and is used to hold the values of variables.

We use the evaluation stack to evaluate arithmetic and Boolean expressions. Formally, **stacks** are enclosed into semantic domain

$$st \in \mathbf{Stack} = (\mathbf{Z} \cup \mathbf{B})^*$$

and each element is a stack - finite sequence of numbers and/or truth values, e.g.

$$v_1 : v_2 : t_1 : v_3 : \ldots : v_n$$

where v_i are numeric values and t_j are truth values, v_1 is on the top of stack.

Syntax of instructions

We shall define two syntactic domains:

- $ins \in \mathbf{Instr}$ – for **instructions**; and
- $c \in \mathbf{Code}$ – for **code**, sequences of instructions.

The instructions of **AM** are given by the abstract syntax:

$$\begin{aligned} ins ::= {} & \texttt{PUSH-}n \mid \texttt{ADD} \mid \texttt{MULT} \mid \texttt{SUB} \mid \texttt{TRUE} \mid \texttt{FALSE} \\ & \mid \texttt{EQ} \mid \texttt{LE} \mid \texttt{AND} \mid \texttt{NEG} \mid \texttt{FETCH-}x \mid \texttt{STORE-}x \\ & \mid \texttt{EMPTYOP} \mid \texttt{BRANCH}(c, c) \mid \texttt{LOOP}(c, c) \end{aligned}$$

$$c \;::= \varepsilon \mid ins : c$$

where ε is the empty sequence.

Semantics of instructions

The semantics of the instructions of the abstract machine is given by an operational semantics. **Transition relation** specifies how to execute the instructions:

$$\langle c, st, s\rangle \Longrightarrow\!\!\!> \langle c', st', s'\rangle$$

Final configuration has a form

$$\langle \varepsilon, \varepsilon, s\rangle$$

which means that all instructions were executed on **AM** and the resulting state is s.

Semantics of instructions

Instruction $\texttt{PUSH-}n$ pushes a constant value n onto the stack.

$$\langle \texttt{PUSH-}n : c, st, s\rangle \Rrightarrow \langle c, \mathscr{N}[\![n]\!] : st, s\rangle \quad (1_{\mathrm{AM}})$$

Instructions $\texttt{ADD}$, $\texttt{MULT}$ and $\texttt{SUB}$ assume, that on the top of stack two numeric values $v_1, v_2 \in \mathbf{Z}$ are pushed. They are removed by instruction, arithmetic operation is evaluated and the result is pushed onto the stack.

$$\langle \texttt{ADD} : c, v_1 : v_2 : st, s\rangle \Rrightarrow \langle c, (v_1 \oplus v_2) : st, s\rangle \quad (2_{\mathrm{AM}})$$

$$\langle \texttt{MULT} : c, v_1 : v_2 : st, s\rangle \Rrightarrow \langle c, (v_1 \otimes v_2) : st, s\rangle \quad (3_{\mathrm{AM}})$$

$$\langle \texttt{SUB} : c, v_1 : v_2 : st, s\rangle \Rrightarrow \langle c, (v_1 \ominus v_2) : st, s\rangle \quad (4_{\mathrm{AM}})$$

Semantics of instructions

Instructions TRUE and FALSE push the constants $\mathbf{tt}$ and $\mathbf{ff}$, respectively, onto the stack:

$$\langle \texttt{TRUE} : c, st, s\rangle \Longrightarrow\!\!\!> \langle c, \mathbf{tt} : st, s\rangle \quad (5_{\mathrm{AM}})$$

$$\langle \texttt{FALSE} : c, st, s\rangle \Longrightarrow\!\!\!> \langle c, \mathbf{ff} : st, s\rangle \quad (6_{\mathrm{AM}})$$

Instruction EQ assumes two numeric values $v_1, v_2 \in \mathbf{Z}$ on the top of stack. They are removed from stack and the result of the relation $=$ (whether the values are equal) is pushed onto the stack:

$$\langle \texttt{EQ} : c, v_1 : v_2 : st, s\rangle \Longrightarrow\!\!\!> \langle c, (v_1 = v_2) : st, s\rangle \quad (7_{\mathrm{AM}})$$

Instruction LE assumes two numeric values $v_1, v_2 \in \mathbf{Z}$ on the top of stack. They are removed from stack and the result of the relation $\leq$ is pushed onto the stack:

$$\langle \texttt{LE} : c, v_1 : v_2 : st, s\rangle \Longrightarrow\!\!\!> \langle c, (v_1 \leq v_2) : st, s\rangle \quad (8_{\mathrm{AM}})$$

Semantics of instructions

Instruction AND assumes two Boolean values $t_1, t_2 \in \mathbf{B}$ on the top of stack. They are removed from stack and the result of conjunction $\wedge$ (value $\mathbf{tt}$ or $\mathbf{ff}$) is pushed onto the stack:

$$\langle \mathtt{AND} : c, t_1 : t_2 : st, s\rangle \Longrightarrow \begin{cases} \langle c, \mathbf{tt} : st, s\rangle, & \text{if } t_1 = \mathbf{tt} \text{ and } t_2 = \mathbf{tt}, \\ \langle c, \mathbf{ff} : st, s\rangle, & \text{if } t_1 = \mathbf{ff} \text{ or } t_2 = \mathbf{ff}. \end{cases} \quad (9_{\mathrm{AM}})$$

Instruction NEG assumes one Boolean value $t \in \mathbf{B}$ on the top of stack. This value is removed from the stack and their logical negation is pushed onto the stack:

$$\langle \mathtt{NEG} : c, t : st, s\rangle \Longrightarrow \begin{cases} \langle c, \mathbf{tt} : st, s\rangle, & \text{if } t = \mathbf{ff} \\ \langle c, \mathbf{ff} : st, s\rangle, & \text{if } t = \mathbf{tt} \end{cases} \quad (10_{\mathrm{AM}})$$

Semantics of instructions

Instruction $\texttt{FETCH-}x$ pushes the value bound to x onto the stack – a value in actual state $s\ x$:

$$\langle \texttt{FETCH-}x : c, st, s\rangle \Rrightarrow \langle c, (s\ x) : st, s\rangle \quad (11_{\mathrm{AM}})$$

Instruction $\texttt{STORE-}x$ pops the topmost element off the stack and updates the storage so that the popped value is bound to x:

$$\langle \texttt{STORE-}x : c, v : st, s\rangle \;\Rrightarrow\; \langle c, st, s[x \mapsto v]\rangle \quad (12_{\mathrm{AM}})$$

Instruction $\texttt{EMPTYOP}$ is an empty instruction, it changes neither the state nor the stack:

$$\langle \texttt{EMPTYOP} : c, st, s\rangle \;\Rrightarrow\; \langle c, st, s\rangle \quad (13_{\mathrm{AM}})$$

Semantics of instructions

Instruction $\texttt{BRANCH}(c_1, c_2)$ changes the flow control.

If the topmost of the stack is the value $\texttt{tt}$ (that is some Boolean expression has been evaluated to true) then the stack is popped and c_1 is to be executed next. Otherwise, if the topmost element of the stack is $\mathbf{ff}$ then it will be popped and c_2 will be executed next:

$$\langle \texttt{BRANCH}(c_1, c_2) : c, t : st, s\rangle \Longrightarrow\!\!\!\!> \begin{cases} \langle c_1 : c, st, s\rangle, & \text{if } t = \mathbf{tt} \\ \langle c_2 : c, st, s\rangle, & \text{if } t = \mathbf{ff} \end{cases} \quad (14_{\mathrm{AM}})$$

Semantics of instructions

A looping construct such as the `while`-construct can be implemented using the instruction $\mathtt{LOOP}(c_1, c_2)$. The semantics of this instruction is is defined by rewriting it to a combination of other constructs including the `BRANCH`-instruction and itself:

$$\langle \mathtt{LOOP}(c_1, c_2) : c, st, s\rangle$$

$$\Longrightarrow \langle c_1 : \mathtt{BRANCH}(c_2 : \mathtt{LOOP}(c_1, c_2), \mathtt{EMPTYOP}) : c, st, s\rangle \qquad (15_{\mathrm{AM}})$$

We emphasize that the assumptions for every instruction are necessary for the continuation of **AM** work.

If the **AM** is expecting value(s) on the top of stack and

- on the top of stack the values are missing, or
- the type of values does not correspond

then the execution of abstract machine **stops**.

Computation sequence

The **execution of program** on **AM** is expressed by computation sequence. Given a sequence c of instructions and a storage s, a **computation sequence** for c and s is either:

- a **finite sequence** of configurations

  $$\alpha_0, \alpha_1, \dots, \alpha_n$$

  satisfying

  $$\alpha_0 = \langle c, \varepsilon, s \rangle \qquad \alpha_i \Rrightarrow \alpha_{i+1} \qquad \alpha_n = \langle \varepsilon, st, s' \rangle,$$

  for $0 \leq i < k$, $k \geq 0$ and where there is no α such that $\alpha_n \Rrightarrow \alpha$. This sequence **terminates**.
- a **finite sequence** of configurations as above but

  $$\alpha_n = \langle c, st, s \rangle.$$

  Here the execution of program is **stopped**.
- an **infinite sequence** of configurations

  $$\alpha_0, \alpha_1, \dots$$

  We say that the sequence is looping – it is infinite.

Example

We consider the following program for abstract machine:

$$\texttt{PUSH-}1 : \texttt{FETCH-}x : \texttt{ADD} : \texttt{STORE-}x$$

with an initial state $s\ x = \mathbf{2}$.

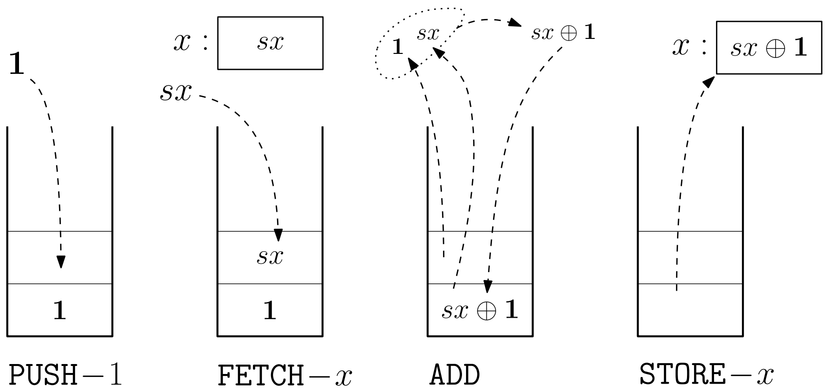
A computation sequence of this program is:

$$\begin{aligned}
\alpha_0 &= \langle \texttt{PUSH-}1 : \texttt{FETCH-}x : \texttt{ADD} : \texttt{STORE-}x, \varepsilon, s\rangle \implies\!\!\!> \\
\alpha_1 &= \langle \texttt{FETCH-}x : \texttt{ADD} : \texttt{STORE-}x, \mathbf{1}, s\rangle \implies\!\!\!> \\
\alpha_2 &= \langle \texttt{ADD} : \texttt{STORE-}x, \mathbf{2} : \mathbf{1}, s\rangle \implies\!\!\!> \\
\alpha_3 &= \langle \texttt{STORE-}x, \mathbf{3}, s\rangle \implies\!\!\!> \\
\alpha_4 &= \langle \varepsilon, \varepsilon, s[x \mapsto \mathbf{3}]\rangle
\end{aligned}$$

Computation sequence terminates and the resulting state is $s\ x = \mathbf{3}$.

$\square$

Semantics of instructions – Example



Properties of computation sequence

We denote

- $\langle c, st, s\rangle \Rrightarrow^k \langle c', st', s'\rangle$ computation sequence of length k,
- $\langle c, st, s\rangle \Rrightarrow^* \langle c', st', s'\rangle$ finite computation sequence.

Lemma 1: Let $c_1, c_2, c' \in \mathbf{Code}$ be codes, $st_1, st_2, st' \in \mathbf{Stack}$ are stacks and $s, s' \in \mathbf{State}$ are states of **AM**. If

$$\langle c_1, st_1, s\rangle \Rrightarrow^k \langle c', st', s'\rangle$$

then

$$\langle c_1 : c_2, st_1 : st_2, s\rangle \Rrightarrow^k \langle c' : c_2, st' : st_2, s'\rangle$$

$\square$

This means that we can extend the code component as well as the stack component without changing the behavior of the machine.

Properties of computation sequence

Lemma 2: Let $c_1, c_2 \in \mathbf{Code}$ be codes, $st, st', st'' \in \mathbf{Stack}$ are stack and $s, s', s'' \in \mathbf{State}$ are states of **AM**. If for some natural k

$$\langle c_1 : c_2, st, s\rangle \Rrightarrow^{k} \langle \varepsilon, st'', s''\rangle$$

then exists configuration $\langle \varepsilon, st', s'\rangle$ and natural numbers k_1, k_2 with $k_1 + k_2 = k$ such that

$$\langle c_1, st, s\rangle \Rrightarrow^{k_1} \langle \varepsilon, st', s'\rangle \quad \text{and}$$

$$\langle c_2, st', s'\rangle \Rrightarrow^{k_2} \langle \varepsilon, st'', s''\rangle$$

□

This means that the execution of a composite sequence of instructions can be split into two pieces.

Properties of computation sequence

Lemma 3: Semantics of abstract machine is **deterministic** if for all choices of $\alpha, \alpha', \alpha''$:

$$\text{if} \quad \alpha \Longrightarrow\!\!\!> \alpha' \quad \text{and} \quad \alpha \Longrightarrow\!\!\!> \alpha'', \ \text{then} \quad \alpha' = \alpha''.$$

$\square$

We shall define the **meaning** of a sequence of instructions as a (partial) function from $\mathbf{State}$ to $\mathbf{State}$:

$$\mathscr{M}[\![\, c \,]\!]s = \begin{cases} s', & \text{if} \quad \langle c, \varepsilon, s \rangle \Longrightarrow\!\!\!>^* \langle \varepsilon, st, s' \rangle, \\ \bot, & \text{otherwise.} \end{cases}$$

Specification of the translation

A **translation** of *Jane* into instructions of **AM** – a **code generating** is defined:

- by **translation functions**,
- for each syntactic domain we define **one** translation function,
- and we define it for **all** alternatives in the given production rule.

Specification of the translation

Translation of **arithmetic expressions** is defined by function

$$\mathscr{TE} : \mathbf{Expr} \to \mathbf{Code}$$

for all syntactic forms of arithmetic expressions in *Jane*:

$$\mathscr{TE}[\![\, n \,]\!] = \texttt{PUSH-}n$$

$$\mathscr{TE}[\![\, x \,]\!] = \texttt{FETCH-}x$$

$$\mathscr{TE}[\![\, e_1 + e_2 \,]\!] = \mathscr{TE}[\![\, e_2 \,]\!] : \mathscr{TE}[\![\, e_1 \,]\!] : \texttt{ADD}$$

$$\mathscr{TE}[\![\, e_1 * e_2 \,]\!] = \mathscr{TE}[\![\, e_2 \,]\!] : \mathscr{TE}[\![\, e_1 \,]\!] : \texttt{MULT}$$

$$\mathscr{TE}[\![\, e_1 - e_2 \,]\!] = \mathscr{TE}[\![\, e_2 \,]\!] : \mathscr{TE}[\![\, e_1 \,]\!] : \texttt{SUB}$$

Note: Code generated for binary expressions consists of the code for the *right* argument followed by that for the *left* argument and finally the appropriate instruction for the operator.

Specification of the translation

Translation of **Boolean expressions** is defined by function:

$$\mathscr{TB} : \mathbf{Bexpr} \rightarrow \mathbf{Code}$$

for all syntactic forms of Boolean expressions in *Jane*:

$$
\begin{aligned}
\mathscr{TB}[\![\, \texttt{true} \,]\!] &= \quad \texttt{TRUE} \\
\mathscr{TB}[\![\, \texttt{false} \,]\!] &= \quad \texttt{FALSE} \\
\mathscr{TB}[\![\, e_1 = e_2 \,]\!] &= \quad \mathscr{TE}[\![\, e_2 \,]\!] : \mathscr{TE}[\![\, e_1 \,]\!] : \texttt{EQ} \\
\mathscr{TB}[\![\, e_1 \leq e_2 \,]\!] &= \quad \mathscr{TE}[\![\, e_2 \,]\!] : \mathscr{TE}[\![\, e_1 \,]\!] : \texttt{LE} \\
\mathscr{TB}[\![\, \neg b \,]\!] &= \quad \mathscr{TB}[\![\, b \,]\!] : \texttt{NEG} \\
\mathscr{TB}[\![\, b_1 \wedge b_2 \,]\!] &= \quad \mathscr{TB}[\![\, b_2 \,]\!] : \mathscr{TB}[\![\, b_1 \,]\!] : \texttt{AND}
\end{aligned}
$$

Example 1

Consider an arithmetic expression $x * (x - 1)$ and generate a code for **AM**:

$$\mathscr{TE}[\![x * (x-1)]\!] = \mathscr{TE}[\![x - 1]\!] : \mathscr{TE}[\![x]\!] : \mathtt{MULT}$$

$$= \mathscr{TE}[\![x - 1]\!] : \mathtt{FETCH}\text{-}x : \mathtt{MULT}$$

$$= \mathscr{TE}[\![1]\!] : \mathscr{TE}[\![x]\!] : \mathtt{SUB} : \mathtt{FETCH}\text{-}x : \mathtt{MULT}$$

$$= \mathtt{PUSH}\text{-}1 : \mathtt{FETCH}\text{-}x : \mathtt{SUB} : \mathtt{FETCH}\text{-}x : \mathtt{MULT}$$

$\square$

Specification of the translation

Translation of **statements** is defined by the function:

$$\mathscr{TS} : \mathbf{Statm} \to \mathbf{Code}$$

for all syntactic forms of statements in *Jane*:

$$\mathscr{TS}[\![\, x{:=}e \,]\!] = \mathscr{TE}[\![\, e \,]\!] : \texttt{STORE-}x$$

$$\mathscr{TS}[\![\, \texttt{skip} \,]\!] = \texttt{EMPTYOP}$$

$$\mathscr{TS}[\![\, S_1; S_2 \,]\!] = \mathscr{TS}[\![\, S_1 \,]\!] : \mathscr{TS}[\![\, S_2 \,]\!]$$

$$\mathscr{TS}[\![\, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \,]\!] = \mathscr{TB}[\![\, b \,]\!] : \texttt{BRANCH}(\mathscr{TS}[\![\, S_1 \,]\!], \mathscr{TS}[\![\, S_2 \,]\!])$$

$$\mathscr{TS}[\![\, \texttt{while}\ b\ \texttt{do}\ S \,]\!] = \texttt{LOOP}(\mathscr{TB}[\![\, b \,]\!], \mathscr{TS}[\![\, S \,]\!])$$

Example 2

Consider a simple program $z{:=}0; \texttt{while}\ (y \le x)\ \texttt{do}\ (z{:=}z+1; x{:=}x-y)$ and generate a code for **AM**:

$$\mathscr{TS}[\![\, z{:=}0; \texttt{while}\ (y \le x)\ \texttt{do}\ (z{:=}z+1; x{:=}x-y) \,]\!]$$

$$= \mathscr{TS}[\![\, z{:=}0 \,]\!] : \mathscr{TS}[\![\, \texttt{while}\ (y \le x)\ \texttt{do}\ (z{:=}z+1; x{:=}x-y) \,]\!]$$

$$= \mathscr{TE}[\![\, 0 \,]\!] : \mathtt{STORE}\text{-}z : \mathtt{LOOP}(\mathscr{TB}[\![\, y \le x \,]\!], \mathscr{TS}[\![\, z{:=}z+1; x{:=}x-y \,]\!])$$

$$\begin{aligned} = \ & \mathtt{PUSH}\text{-}0 : \mathtt{STORE}\text{-}z \\ & \quad : \mathtt{LOOP}(\mathscr{TE}[\![\, x \,]\!] : \mathscr{TE}[\![\, y \,]\!] : \mathtt{LE}, \mathscr{TS}[\![\, z{:=}z+1 \,]\!] : \mathscr{TS}[\![\, x{:=}x-y \,]\!]) \end{aligned}$$

$$\begin{aligned} = \ & \mathtt{PUSH}\text{-}0 : \mathtt{STORE}\text{-}z \\ & \quad : \mathtt{LOOP}(\mathtt{FETCH}\text{-}x : \mathtt{FETCH}\text{-}y : \mathtt{LE}, \\ & \qquad \mathtt{PUSH}\text{-}1 : \mathtt{FETCH}\text{-}z : \mathtt{ADD} : \mathtt{STORE}\text{-}z \\ & \qquad : \mathtt{FETCH}\text{-}y : \mathtt{FETCH}\text{-}x : \mathtt{SUB} : \mathtt{STORE}\text{-}x) \end{aligned}$$

$\square$

The semantic function $\mathscr{S}_{AM}$

An **abstract implementation** of statement S is obtained by executing of the following steps:

- translating the statement S into code of **AM** by translation functions, and
- executing the code on **AM** by semantics of instructions.

This can be expressed by the **semantic function** $\mathscr{S}_{AM}$ of abstract machine:

$$\mathscr{S}_{AM} : \mathbf{Statm} \rightarrow (\mathbf{State} \rightharpoonup \mathbf{State})$$

which is defined for statement $S \in \mathbf{Statm}$ as follows:

$$\mathscr{S}_{AM}[\![S]\!] = (\mathscr{T}\mathscr{S} \circ \mathscr{M})[\![S]\!]$$

Correctness of abstract implementation

The **correctness of abstract implementation** amounts to showing that, if we first translate a statement into code for **AM** and then execute code, then we must to obtain the same result as specified by the operational semantics of *Jane*.

Proof will be done in three steps:

- for arithmetic expressions,
- for Boolean expressions,
- for statements.

Correctness of the implementation of arithmetic expressions

Theorem 1: For all arithmetic expressions e we have:

$$\langle \mathscr{T}\mathscr{E}[\![\, e \,]\!], \varepsilon, s \rangle \Rrightarrow^{*} \langle \varepsilon, \mathscr{E}[\![\, e \,]\!]s, s \rangle$$

Proof: The proof is by structural induction on e.

1. **The case** n:
   From translation function we have $\mathscr{T}\mathscr{E}[\![\, n \,]\!] = \texttt{PUSH-}n$.
   From semantics of instruction it implies $\langle \texttt{PUSH-}n, \varepsilon, s \rangle \Rrightarrow \langle \varepsilon, \mathscr{N}[\![\, n \,]\!], s \rangle$.
   Since

$$\mathscr{E}[\![\, n \,]\!]s = \mathscr{N}[\![\, n \,]\!]$$

   we have completed the proof in this case.

Correctness of the implementation of arithmetic expressions

2. **The case** x (variable):
   From translation function we have $\mathscr{TE}[\![\, x \,]\!] = \texttt{FETCH-}x$.
   From semantics of instruction it implies $\langle \texttt{FETCH-}x, \varepsilon, s\rangle \Longrightarrow\!\!\!> \langle \varepsilon, (s\ x), s\rangle$.
   Since:

$$\mathscr{E}[\![\, x \,]\!]s = s\ x$$

   this is required result.

Correctness of the implementation of arithmetic expressions

3. **The case** $e_1 + e_2$:
   We have $\mathcal{TE}[\![e_1 + e_2]\!] = \mathcal{TE}[\![e_2]\!] : \mathcal{TE}[\![e_1]\!] : \mathtt{ADD}$.
   The **induction hypothesis** (IH) applied to both expressions e_1, e_2 gives that:

$$\langle \mathcal{TE}[\![e_1]\!], \varepsilon, s\rangle \Rightarrow\!\!\!\gg^* \langle \varepsilon, \mathcal{E}[\![e_1]\!]s, s\rangle$$
$$\langle \mathcal{TE}[\![e_2]\!], \varepsilon, s\rangle \Rightarrow\!\!\!\gg^* \langle \varepsilon, \mathcal{E}[\![e_2]\!]s, s\rangle$$

   In both cases all intermediate configurations will have a non-empty evaluation stack. Thus:

$$\begin{array}{ll}
\langle \mathcal{TE}[\![e_2]\!] : \mathcal{TE}[\![e_1]\!] : \mathtt{ADD}, \varepsilon, s\rangle \Rightarrow\!\!\!\gg^* & \text{from IH and Lemma 1} \\
\langle \mathcal{TE}[\![e_1]\!] : \mathtt{ADD}, \mathcal{E}[\![e_2]\!]s, s\rangle \Rightarrow\!\!\!\gg^* & \text{from IH and Lemma 1} \\
\langle \mathtt{ADD}, \mathcal{E}[\![e_1]\!]s : \mathcal{E}[\![e_2]\!]s, s\rangle & \text{from sem. AM} \\
\langle \varepsilon, \mathcal{E}[\![e_1]\!]s \oplus \mathcal{E}[\![e_2]\!]s, s\rangle &
\end{array}$$

   Since

$$\mathcal{E}[\![e_1 + e_2]\!]s = \mathcal{E}[\![e_1]\!]s \oplus \mathcal{E}[\![e_2]\!]s$$

   we have the desired result.

The proof for other cases is analogous.

Correctness of the implementation of Boolean expressions

Theorem 2: For all Boolean expressions b we have:

$$\langle \mathscr{TB}[\![b]\!], \varepsilon, s\rangle \Rightarrow\!\!\!\gg^* \langle \varepsilon, \mathscr{B}[\![b]\!], s\rangle$$

Proof: The proof is analogous to the proof for correctness of arithmetic expressions.

$\square$

Theorem 3: For every statement S of $\mathscr{J}ane$ we have:

$$\mathscr{S}_{ns}[\![S]\!] = \mathscr{S}_{AM}[\![S]\!]$$

Proof: The theorem is proved in two stages:

1. if $\langle S, s\rangle \to s'$ then $\langle \mathscr{TS}[\![S]\!], \varepsilon, s\rangle \Rightarrow\!\!\!\gg^* \langle \varepsilon, \varepsilon, s'\rangle$;
2. if $\langle \mathscr{TS}[\![S]\!], \varepsilon, s\rangle \Rightarrow\!\!\!\gg^k \langle \varepsilon, st, s'\rangle$ then $\langle S, s\rangle \to s'$ and $st = \varepsilon$.