4. Operational Semantics

David Pereira José Proença Eduardo Tovar

FVOCA 2021/2022

Formal Verification of Critical Applications

CISTER – ISEP
Porto, Portugal

https://cister-labs.github.io/fvoca2122

Why look into formal semantics

Programming languages, syntax, and semantics

When we first look into a programming language...

... at first, we typically look into its syntax, but:

- it just deals with correctly formed sentences;
- syntax is not concerned with soundness;
- thus, t**he programmer may get lost if it becomes necessary to check that the specified program actually computes the intended operation**

Programming languages, syntax, and semantics

Important note!

It is important that the semantics is **formal**, **systematic** and **verifiable** so that:

- the user has access to an unambiguous description of the effect of a program
- a starting point for a correct implementation
- a basis for program analysis and synthesis, i.e., **transformation**, **optimisation**, and **verification**

Another important note!

actually showing a program to be correct is more work than writing the program itself, but it is not a task to be neglected

States and types of operational semantics

States in formal semantics

The meaning of programs

- Semantics of programming languages deals with the meaning of programs that execute on a computer, **runnig in memory** and **using various resources**;
- to **express execution correctly**, we need also to **consider the status of the memory during execution**.

State of a program

- The **state of the memory** is fundamental in all definitions of semantics
- we will consider only **programs that compute through variables**, hence **we are not interested in the contents of actual physical addresses**
- will **abstract from the actual memory** and focus on the **representation of the values stored in variables**

How we will represent states

We will use the notation

$$[x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n],$$

where $x_i \in \mathcal{V}$ are variable names (identifiers) and $v_i \in \mathcal{D}$ represent values that are assigned to the variables (in the scope of this class we will be considering mostly integeres and Booleans)

Types of Operational Semantics

Classification of Operational Semantics

Operational Semantics

Focus on how the effect of a computation is produced: it is an abstraction of machine execution in that it expresses the meaning of a program - running on a machine in a specific state - by returning its result, the output.

Denotational Semantics

In this approach, the meaning of a program is a function, that maps the state of the machine before execution to the state after execution.

Axiomatic Semantics

With this semantics, the properties of the effect of executing the constructs are expressed as assertions

We will focus on (specific) operational semantics and axiomatic semantics!

Example with Operational Semantics

Looking at the differences in pratice

In the next slides we will look into the differences of the approaches with the following toy example

$$z := x; x := y; y := z$$

and assuming the following existing assignment of variables to values:

- $x \mapsto 5$
- $y \mapsto 7$
- $z \mapsto 0$

Example with Operational Semantics

We will be using the notation $\langle p, s \rangle$, where p denotes the program code, and s the state, i.e., the mapping of values to variables names. Also, for now, lets assume that:

- to execute a sequence of statements, typically separated by ;, execute individual statements from left to right;
- to execute an assignment $x := v$, first calculate the value of v and then assign it to x.

$$\begin{aligned}
\langle z := x; x := y; y := z, [x \mapsto 5, y \mapsto 7, z \mapsto 0] \rangle \Rightarrow \\
\langle x := y; y := z, [x \mapsto 5, y \mapsto 7, z \mapsto 5] \rangle \Rightarrow \\
\langle y := z, [x \mapsto 7, y \mapsto 7, z \mapsto 5] \rangle \Rightarrow \\
\langle \epsilon, [x \mapsto 7, y \mapsto 5, z \mapsto 5] \rangle
\end{aligned}$$

Example with Denotational Semantics

We will be using a mathematical function with the type $State \rightarrow State$ to evaluate the code, denoted $S[p]$, considering:

- executing a sequence of statements amounts at function composition, i.e., $S[p_1; p_2] = S[p_1] \circ S[p_2]$
- executing an assignment $x := v$ returns the current state where the mapping of the variable x is updated as follows: $S[x := v](s, y) = s(y)$ if $x \neq y$, or v otherwise, where s is the function representing the state and y a variable being evaluated under the S interpretation.

Example with Denotational Semantics

Getting back to the running example, and using the definition of evaluation of program sequence, we know that

$$S[z := x; x := y; y := z] = S[z := x] \circ S[x := y] \circ S[y := z]$$

Proceeding with the evaluation we have:

$$\begin{aligned}
S[z := x; x := y; y := z]([x \mapsto 5, y \mapsto 7, z \mapsto 0]) = \\
S[z := x] \circ S[x := y] \circ S[y := z]([x \mapsto 5, y \mapsto 7, z \mapsto 0]) = \\
S[x := y] \circ S[y := z]([x \mapsto 5, y \mapsto 7, z \mapsto 5]) = \\
S[y := z]([x \mapsto 7, y \mapsto 7, z \mapsto 5]) = \\
[x \mapsto 5, y \mapsto 7, z \mapsto 5]
\end{aligned}$$

Note: For denotational semantics, the meaning of a program depends only on the program itself. No state information is needed to establish a meaning.

Example with Axiomatic Semantics

Partial Correctness

Axiomatic semantics deals with partial correctness, i.e., it proves the correctness of a program p with respect to its **pre-** and **post-conditions**. The usual representation is as follows:

$$\{Pre\}\, p\, \{Post\}$$

which in the case of the running example can be instantiated to

$$\{x = n \wedge y = m\}\, z := x; x := y; y := z\, \{x = m \wedge y = n\}$$

that expresses that if the assigned values of x and y are, at the start of the program, n and m, respectively, then when the program terminates, it must hold that their values have been swapped.

Example with Axiomatic Semantics

Further ahead in this module of FVOCA we will look deeply into axiomatica semantics, typically known as Hoare Logic. For now, lets look into a proof sketch that intuitively shows how the actual proof, with the concrete rules for a well defined syntax of a programming language, could take place

$$(P1)\ \{x = n \wedge y = m\}\, z := x\, \{z = n \wedge y = m\}$$
$$(P2)\ \{z = n \wedge y = m\}\, x := y\, \{z = n \wedge x = m\}$$
$$(P3)\ \{x = n \wedge x = m\}\, z := x; x := y \{z = n \wedge x = m\}$$
$$(P4)\ \{z = n \wedge x = m\}\, y := z\, \{y = n \wedge x = m\}$$
$$(P5)\{x = n \wedge y = m\}\, z := x; x := y; y := z\, \{x = m \wedge y = n\}$$

However, for more complex cases, this type of approach is not so easy to address:

$$\{x = n \wedge y = m\}\, \text{while}(\text{true})\, \text{do skip}\, \{x = m \wedge y = n\}$$

So, what we will be learning in FVOCA?

What will we be learning in this module of FVOCA?

Milestones to be achieved (Phase 1):

- Select a target programming language (abstract syntax)
- Select a semantics for that language (abstract semantics)
- Mathematically prove properties of programs written in the chosen language

Milestones to be achieved (Phase 2):

- How to extend the abstract semantics to allow logical annotations that characterise code
- What logics and proof rules can be used to reason about annotated programs abstract semantics
- Learn how to automate generation of proof obligations and use theorem provers in practice

What will we be learning in this module of FVOCA?

Today we will look once again into operational semantics and, in the practical class, start to design and program our own FVOCA interpreters! Focus will be on operational semantics, namely structural operational semantics (we will dive into this type of semantics still in this class!)

A Simple Imperative Language

While^{Int} – Syntax

$$
\begin{array}{lll}
x & \in & \text{Identifiers} \\
n & \in & \text{Numerals} \\
B & ::= & \text{true} \mid \text{false} \mid B \wedge B \mid B \vee B \mid \neg B \mid E < E \mid E = E \qquad \text{(boolean-expr)} \\
E & ::= & n \mid x \mid E + E \mid E * E \mid E - E \qquad \text{(int-expr)} \\
C & ::= & \text{skip} \mid C; C \mid x := E \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \qquad \text{(command)}
\end{array}
$$

Assume operators to be left associative.
Use '{' and '}' to clarify precedence when necessary.

Natural Semantics - a quick overview

Natural Semantics

A type of operational semantics

Natural semantics, aka Big-Step Semantics, are operational semantics that directly give the results of the code under evaluation. They don't consider intermediate steps and therefore are not suited to programming languages with constructs that need fine-grained analysis, e.g., concurrency. or concurrency

Mathematical view of Natural Semantics

To make a proper evaluation under the context of natural semantics, we consider a relation $\rightsquigarrow$ that maps a configuration of the program $\langle p, s \rangle$ into its final result, which must be a member of the domain (in the case presented here, either Boolean values or integers). The evaluation of a variable v in a state s is denoted $s(v)$.

Natural Semantics – booleans and integers

$$\frac{}{\langle x, s\rangle \rightsquigarrow s(x)}\ (\text{var}) \qquad \frac{}{\langle \text{true}, s\rangle \rightsquigarrow \text{true}}\ (\text{true}) \qquad \frac{}{\langle \text{false}, s\rangle \rightsquigarrow \text{false}}\ (\text{false}) \qquad \frac{}{\langle n, s\rangle \rightsquigarrow n}\ (\text{int})$$

$$\frac{\langle B, s\rangle \rightsquigarrow \text{false}}{\langle B \wedge B', s\rangle \rightsquigarrow \text{false}} \qquad \frac{\langle B, s\rangle \rightsquigarrow \text{true} \quad \langle B', s\rangle \rightsquigarrow s'}{\langle B \wedge B', s\rangle \rightsquigarrow s'}$$

$$\frac{\langle B, s\rangle \rightsquigarrow \text{true}}{\langle B \vee B', s\rangle \rightsquigarrow \text{true}} \qquad \frac{\langle B, s\rangle \rightsquigarrow \text{false} \quad \langle B', s\rangle \rightsquigarrow s'}{\langle B \vee B', s\rangle \rightsquigarrow s'}$$

$$\frac{\langle E, s\rangle \rightsquigarrow n \quad \langle E', s\rangle \rightsquigarrow n'}{\langle E \odot E', s\rangle \rightsquigarrow n \odot n'} \quad \text{where } \odot \in \{+, -, *\}$$

Natural Semantics – commands

$$\frac{}{\langle \text{skip}, s\rangle \rightsquigarrow s}\ \text{(skip)}$$

$$\frac{\langle E, s\rangle \rightsquigarrow n}{\langle x{:=}E, s\rangle \rightsquigarrow s[x \mapsto n]}\ \text{(assign)}$$

$$\frac{\langle C_1, s\rangle \rightsquigarrow s' \quad \langle C_2, s'\rangle \rightsquigarrow s''}{\langle C_1; C_2\ , s\rangle \rightsquigarrow s''}\ \text{(seq)}$$

$$\frac{\langle B, s\rangle \rightsquigarrow \text{true} \quad \langle C_t, s\rangle \rightsquigarrow s'}{\langle \text{if } B \text{ then } C_t \text{ else } C_f\ , s\rangle \rightsquigarrow s'}\ \text{(if-then)}$$

$$\frac{\langle B, s\rangle \rightsquigarrow \text{false} \quad \langle C_f, s\rangle \rightsquigarrow s'}{\langle \text{if } B \text{ then } C_t \text{ else } C_f\ , s\rangle \rightsquigarrow s'}\ \text{(if-else)}$$

$$\frac{\langle B, s\rangle \rightsquigarrow \text{false}}{\langle \text{while } B \text{ do } C\ , s\rangle \rightsquigarrow s'}\ \text{(while-false)}$$

$$\frac{\langle B, s\rangle \rightsquigarrow \text{true} \quad \langle C, s\rangle \rightsquigarrow s' \quad \langle \text{while } B \text{ do } C, s'\rangle \rightsquigarrow s''}{\langle \text{while } B \text{ do } C\ , s\rangle \rightsquigarrow s''}\ \text{(while-true)}$$

Structural Operational Semantics

Structural Operational Semantics (SOS)

Focus of SOS

Structural Operational Semantics have as main focus the individual steps of the execution of the program.

Differences wrt. Natural Semantics

Like Natural Semantics, the steps are defined as transitions, but the right hand-side of the transition may not be the final state, but rather some intermediate step of computaion.

Structural Operational Semantics – arithmetic expressions

Expressions are evaluated as functions from variable identifiers onto integers. The rules are:

$$\text{(var)}\quad \frac{s(x) = n}{\langle x, s\rangle \Longrightarrow \langle n, s\rangle} \qquad \text{(add/mul/sub/div-l)}\quad \frac{\langle E_1, s\rangle \Longrightarrow \langle E_1', s\rangle}{\langle E_1 \odot E_2\ ,\ s\rangle \Longrightarrow \langle E_1' \odot E_2, s\rangle}$$

$$\text{(add/mul/sub/div-r)}\quad \frac{\langle E_2\ ,\ s\rangle \Longrightarrow \langle E_2', s\rangle}{\langle n \odot E_2\ ,\ s\rangle \Longrightarrow \langle n \odot E_2', s\rangle} \qquad \text{(add/mul/sub/div)}\quad \frac{n \odot_{\mathbb{I}} m = p \in \mathbb{I}}{\langle n \odot m\ ,\ s\rangle \Longrightarrow \langle p, s\rangle}$$

where $\odot \in \{+, -, *\}$ and $\odot_{\mathbb{I}}$ stands for the concrete operation on the domain of integers.

Structural Operational Semantics – Boolean expressions

Expressions are evaluated as functions from variable identifiers onto integers. The rules are:

$$\frac{\langle B, s\rangle \Longrightarrow \langle B', s\rangle}{\langle \neg B, s\rangle \Longrightarrow \langle \neg B', s\rangle}\ \text{(not-I)} \qquad \frac{b = \text{true}}{\langle \neg b, s\rangle \Longrightarrow \langle \text{false}, s\rangle}\ \text{(not-true)} \qquad \frac{b = \text{false}}{\langle \neg b, s\rangle \Longrightarrow \langle \text{true}, s\rangle}\ \text{(not-false)}$$

$$\frac{\langle B_1, s\rangle \Longrightarrow \langle B_1', s\rangle}{\langle B_1 \odot B_2\ ,\ s\rangle \Longrightarrow \langle B_1' \odot B_2, s\rangle}\ \text{(and/or-I)} \qquad \frac{\langle B_2\ ,\ s\rangle \Longrightarrow \langle B_2', s\rangle}{\langle b \odot B_2\ ,\ s\rangle \Longrightarrow \langle b \odot B_2', s\rangle}\ \text{(and/or-r)}$$

$$\frac{p_1 \odot_{\mathbb{I}} p_2 = p \in \mathbb{B}}{\langle p_1 \odot p_2\ ,\ s\rangle \Longrightarrow \langle p, s\rangle}\ \text{(and/or)}$$

where $\odot \in \{\wedge, \vee\}$ and $\odot_{\mathbb{B}}$ stands for the concrete operation on the domain of integers.

Structural Operational Semantics – commands

(assign-1)

$$\frac{\langle E, s\rangle \Longrightarrow \langle E', s\rangle}{\langle x{:=}E, s\rangle \Longrightarrow \langle x{:=}E', s\rangle}$$

(assign-2)

$$\frac{}{\langle x{:=}n, s\rangle \Longrightarrow \langle \mathsf{skip}, s[x \mapsto n]\rangle}$$

(seq)

$$\frac{\langle C_1, s\rangle \Longrightarrow \langle C_1', s'\rangle}{\langle C_1; C_2\ , \ s\rangle \Longrightarrow \langle C_1'; C_2, s'\rangle}$$

(seq-skip)

$$\frac{}{\langle \mathsf{skip}; C_2\ , \ s\rangle \Longrightarrow \langle C_2, s\rangle}$$

(if-then)

$$\frac{\langle B, s\rangle \rightsquigarrow \mathsf{true}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_t\ \mathsf{else}\ C_f\ ,\ s\rangle \Longrightarrow \langle C_t, s\rangle}$$

(if-else)

$$\frac{\langle B, s\rangle \rightsquigarrow \mathsf{false}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_t\ \mathsf{else}\ C_f\ ,\ s\rangle \Longrightarrow \langle C_f, s\rangle}$$

(while)

$$\frac{}{\langle \mathsf{while}\ B\ \mathsf{do}\ C\ ,\ s\rangle \Longrightarrow \langle \mathsf{if}\ B\ \mathsf{then}\ (C; \mathsf{while}\ B\ \mathsf{do}\ C)\ \mathsf{else}\ \mathsf{skip}, s\rangle}$$

Example with expressions

Let's assume two variables, foo and bar, a state $s : Var \to \mathbb{I}$ such that $s(foo) = 4$ and $s(bar) = 3$. Let's now provide a reasoning for the calculation of $(foo + 2) \times (bar + 1)$

$$\text{(mul-l)}\ \frac{\langle foo + 2, s\rangle \Longrightarrow \langle E_1', s\rangle}{\langle (foo + 2) \times (bar + 1), s\rangle \Longrightarrow \langle E_1' \times (bar + 1), s\rangle}$$

We now have to show that the premise actually holds and thus need to find what E_1' is.

$$\text{(add-l)}\ \frac{\langle foo, s\rangle \Longrightarrow \langle E_1'', s\rangle}{\langle foo + 2, s\rangle \Longrightarrow \langle E_1'' + 2, s\rangle}$$

Now it is enough to apply the rule that maps values to variable identifiers.

$$\text{(var)}\ \frac{s(foo) = 4}{\langle foo, s\rangle \Longrightarrow \langle 4, s\rangle}$$

But we are not yet finished; lets continue in the next slide!

Example with expressions

Now that we know the previous derivations we can proceed with the substitution and make a reduction tep using the (add) rule.

$$\text{(add)}\ \frac{\langle 4+2, s\rangle \Longrightarrow \langle 6, s\rangle}{\langle (4+2)\times(bar+1), s\rangle \Longrightarrow \langle 6\times(bar+1), s\rangle}$$

We now have to show that the premise actually holds and thus need to find what E_1' is.

$$\text{(add-r)}\ \frac{\text{(add-l)}\ \dfrac{\text{(var)}\ \dfrac{s(bar)=3}{\langle bar, s\rangle \Longrightarrow \langle 3, s\rangle}}{\langle bar+1, s\rangle \Longrightarrow \langle E_2', s\rangle}}{\langle 6\times(bar+1), s\rangle \Longrightarrow \langle 6\times E_2'+2, s\rangle}$$

We can now proceed as before, and obtain the desired derivation.

Example

Let us start with a very simple example (and ignore skip command in the application of the transition rules for simplicity.

$$\dfrac{\dfrac{\langle x := 1, s\rangle \Longrightarrow \langle \mathsf{skip}, s[x \mapsto 1]\rangle}{\langle x := 1; y := 2, s[x \mapsto 1]\rangle \Longrightarrow \langle y := 2, s[x \mapsto 1]\rangle}}{\langle x := 1; y := 2; z := 3, s\rangle \Longrightarrow \langle y := 2; z := 3, s[x \mapsto 1]\rangle}$$

Truth to be told, we just need one proof step (using associativity of sequence)[1]!

$$\dfrac{\langle x := 1, s\rangle \Longrightarrow \langle \mathsf{skip}, s[x \mapsto 1]\rangle}{\langle x := 1; y := 2; z := 3, s\rangle \Longrightarrow \langle y := 2; z := 3, s[x \mapsto 1]\rangle}$$

[1] We will learn more about associativity and other properties on upcoming lectures.

Interesting Extensions

While^{Int} – Adding variables declarations and procedure definitions

$$
\begin{array}{rcll}
x & \in & \text{Identifiers} & \\
n & \in & \text{Numerals} & \\
p & \in & \text{Procedure Identifiers} & \\
B & ::= & \text{true} \mid \text{false} \mid B \wedge B \mid B \vee B \mid \neg B \mid E < E \mid E = E & \text{(boolean-expr)} \\
E & ::= & n \mid x \mid E + E \mid E * E \mid E - E & \text{(int-expr)} \\
D_v & ::= & \text{var } x ::= E; D_v? & \text{(var-decl)} \\
D_p & ::= & \text{proc } p \text{ is } C; D_p? & \text{(var-decl)} \\
C & ::= & \text{skip} \mid C; C \mid x := E \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \mid \text{call } p & \text{(command)}
\end{array}
$$

Aborting execution

Add a new keyword to commands:

$$C ::= \mathsf{skip} \mid \ldots \mid \mathsf{abort} \qquad \text{(command)}$$

Introduce the following rule in the semantics:

(abort-NS)

$$\frac{}{\langle \mathsf{abort}, s \rangle \rightsquigarrow \bot}$$

(abort-SOS)

$$\frac{}{\langle \mathsf{abort}, s \rangle \Longrightarrow \langle \mathsf{skip}, \bot \rangle}$$

Non-Determinism

Add a new keyword to commands:

$$C \;::=\; \text{skip} \;\mid\; \ldots \;\mid\; C_1 \text{ or } C_2 \qquad \text{(command)}$$

Introduce the following rule in the semantics:

$$\frac{\langle C_1, s\rangle \Longrightarrow \langle C_1', s'\rangle}{\langle C_1 \text{ or } C_2, s\rangle \Longrightarrow \langle C_1' \text{ or } C_2, s'\rangle}\ \text{(ndet-l)}$$

$$\frac{\langle C_2, s\rangle \Longrightarrow \langle C_2', s'\rangle}{\langle C_1 \text{ or } C_2, s\rangle \Longrightarrow \langle C_1 \text{ or } C_2', s'\rangle}\ \text{(ndet-l)}$$

While^{Int} with assertions – Syntax

$$
\begin{array}{lll}
x & \in & \text{Identifiers} \\
n & \in & \text{Numerals} \\
B & ::= & \text{true} \mid \text{false} \mid B \wedge B \mid B \vee B \mid \neg B \mid E \odot E \qquad \text{(boolean-expr)} \\
E & ::= & n \mid x \mid E + E \mid E * E \mid E - E \qquad \text{(int-expr)} \\
C & ::= & \text{skip} \mid C; C \mid I := E \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \mid \\
 & & \text{assert}(B) \qquad \text{(command)}
\end{array}
$$

Where $\odot \in \{<, >, \leq, \geq, ==\}$.

Structural Operational Semantics of Machine Code

Do we need to stick to IMP?

So far we have been looking into simple imperative languages, similar to the C family of languages (well, in reality, a very reduced version of such languages). But what about other families of languages? Indeed it is possible to define other types of languages, and we will be looking into an abstract assembly language!

Establishing the basis

Assembly-like languages usually consider registers and a stack, in their more simplest form. So, we will be looking into a language that consists of "configurations" of the type

$$\langle c, e, s \rangle$$

such that c is a program, e is the evolution stack, and s is the storage (i.e., it keeps the "variables"). The evaluation stack can be seen as an infinite list of elements in $(\mathbb{Z} \cup \mathbb{B})$.

Syntax

$$
\begin{aligned}
x &\in \text{Identifiers} \\
n &\in \text{Numerals} \\
\mathit{inst} &::= \text{PUSH } n \mid \text{ADD} \mid \text{MULT} \mid \text{SUB} \mid \text{TRUE} \mid \text{FALSE} \mid \\
&\quad\ \text{EQ} \mid \text{LE} \mid \text{AND} \mid \text{NEG} \mid \text{FETCH } x \mid \text{STORE } x \mid \text{NOOP} \\
&\quad\ \text{BRANCH}(c, c) \mid \text{LOOP}(c, c) && \text{(instructions)} \\
c &::= \epsilon \mid \mathit{inst} : c && \text{(code)}
\end{aligned}
$$

Semantics

$$\langle \text{PUSH } n : c, e, s \rangle \Longrightarrow \langle c, n : e, s \rangle$$

$$\langle \text{ADD} : c, n_1 : n_2 : e, s \rangle \Longrightarrow \langle c, (n_1 + n_2) : e, s \rangle$$

$$\langle \text{SUB} : c, n_1 : n_2 : e, s \rangle \Longrightarrow \langle c, (n_1 - n_2) : e, s \rangle$$

$$\langle \text{MULT} : c, n_1 : n_2 : e, s \rangle \Longrightarrow \langle c, (n_1 \times n_2) : e, s \rangle$$

$$\langle \text{TRUE} : c, e, s \rangle \Longrightarrow \langle c, \mathbf{tt} : e, s \rangle$$

$$\langle \text{FALSE} : c, e, s \rangle \Longrightarrow \langle c, \mathbf{ff} : e, s \rangle$$

$$\langle \text{EQ} : c, n_1 : n_2 : e, s \rangle \Longrightarrow \langle c, (n_1 = n_2) : e, s \rangle$$

$$\langle \text{LE} : c, n_1 : n_2 : e, s \rangle \Longrightarrow \langle c, (n_1 \leq n_2) : e, s \rangle$$

$$\langle \text{AND} : c, b_1 : b_2 : e, s \rangle \Longrightarrow \langle c, (b_1 \wedge b_2) : e, s \rangle$$

$$\langle \text{NEG} : c, b : e, s \rangle \Longrightarrow \langle c, (\neg b) : e, s \rangle$$

Semantics

$$\langle \text{FETCH } n : c, e, s \rangle \Longrightarrow \langle c, s(x) : e, s \rangle$$

$$\langle \text{STORE } n : c, e, s \rangle \Longrightarrow \langle c, e, s[x \mapsto n] \rangle$$

$$\langle \text{NOOP} : c, e, s \rangle \Longrightarrow \langle c, e, s \rangle$$

$$\langle \text{BRANCH}(c_1, c_2) : c, b : e, s \rangle \Longrightarrow \langle c_1 : c, e, s \rangle \; \textit{if} \; b = \mathbf{tt}$$

$$\langle \text{BRANCH}(c_1, c_2) : c, b : e, s \rangle \Longrightarrow \langle c_1 : c, e, s \rangle \; \textit{if} \; b = \mathbf{ff}$$

$$\langle \text{LOOP}(c_1, c_2) : c, e, s \rangle \Longrightarrow \langle c_1 : \text{BRANCH}(c_2 : \text{LOOP}(c_1, c_2), \text{NOOP}) : c, e, s \rangle$$

A small example

Lets go through a simple example, where we assume that the value of the variable x is 3.

$$\begin{aligned}
\langle \text{PUSH } 1 : \text{FETCH } x : \text{ADD} : \text{STORE } x, \epsilon, s\rangle &\Longrightarrow \\
\langle \text{FETCH } x : \text{ADD} : \text{STORE } x, 1, s\rangle &\Longrightarrow \\
\langle \text{ADD} : \text{STORE } x, 3 : 1, s\rangle &\Longrightarrow \\
\langle \text{STORE } x, 4, s\rangle &\Longrightarrow \\
\langle \epsilon, \epsilon, s[x \mapsto 4]\rangle &
\end{aligned}$$

Formal Semantics are Fun, but what can we do with them?

Certified Compilation

One particular important application of formal semantics is certified compilation, that is, the process of transforming high-level source code into a machine level instruction set if guaranteed (i.e., mathematically proved), is correct.

Steps

- Define an instruction set and the mathematic meaning of its instructions, i.e., the rules for a formal semantics
- Define a translation function
- Prove that if $\langle C, s\rangle \Longrightarrow_{hlc} \langle \text{skip}, s'\rangle$, then $\langle \mathcal{T}(C), s\rangle \Longrightarrow_{llc} \langle NOOP, s'\rangle$[2]

[2] $NOOP$ means an abstraction of no-operation, which can be represented differently depending on the instruction set.

Suggested exercises for practicing - I

Construct your own command and semantic rules

As a first exercise, I would like to appeal to your criativity and do the following:

- select a command that is not available in our simple imperative language, provide its abstract syntax and either Natural Semantics or Structural Operational Semantics transition rules/steps;
- in the case when the new command's transition rules are defined by translating into combinations of primitive rules, try to provide an alternative formulation that does not use those primitive rules.
- if needed, you can also extend the type of state of a program. Remember that currently on a function mapping variable identifiers to values in $\mathbb{I}$ is defined.

Suggested exercises for practicing - II

Construct your own formal semantics interpreter

As a second exercise, I would like to appeal to your passion for coding do the following:

- select the programming language that most suits you. For simplicity I suggest Python, and highly suggest that you avoid system programming languages such as C.
- find a representation for the abstract syntax of expressions and commands using the facilities of the chosen programming language
- experiment implementing a Natural Semantics interpreter
- by the way, next laboratory language will be dedicated to these to exercises, so don't forget to bring laptops!