3. A Not So Formal Introduction to Formal Verification of Program Code

David Pereira José Proença Eduardo Tovar

FVOCA 2021/2022

Formal Verification of Critical Applications

CISTER – ISEP
Porto, Portugal

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

Once Upon a A While Language . . .

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

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 – 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 (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 – expressions

Expressions are evaluated as functions from variable identifiers onto integers (similarly to natural semantics). However, in SOS, all reduction steps are accounted in the reduction rules and consider all intermediate states.

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

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

where $\odot \in \{+, -, *\}$ and $\odot_{\mathbb{I}}$ stands for the concrete operation on the domain of integers. Please note that this semantics is not considering errors, e.g., when faced with a division by 0. That requires changing the notion of state so that it incorporates the concept of "program going wrong".

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 step 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.

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

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.

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!

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.

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:

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

Introduce the following rule in the semantics:

(abort-NS)

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

(abort-SOS)

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

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:

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

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

What can go wrong?

Assuming a well written program

- using an undefined variable
- loops never end
- the result is unexpected
- ...

While^{Int} with assertions – Syntax

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

Assume operators to be left associative.

Use '{' and '}' to clarify precedence when necessary.