Operational Semantics
CS 6371: Advanced Programming Languages

Kevin W. Hamlen

January 30, 2024
Let’s create a new language: Simple IMPerative Language (SIMPL)

Backus-Naur Form (BNF)
- Invented by John Backus and Peter Naur (inventors of ALGOL-60 and later FORTRAN)
- Notation for expressing context-free grammars (CFGs)

Convention: Teletype font for symbols vs. mathematical font for mathematical operators
- +, -, and * are symbols from your keyboard that have no particular mathematical meaning
- +, -, and * are the mathematical operators for addition, subtraction, and multiplication
- (To make things easier for our OCaml implementation, we will define them to be 31-bit integer addition, subtraction, and multiplication operators, which is how OCaml performs those operations natively.)

### Syntax of SIMPL

|.commands| $c ::= \text{skip} \mid c_1 ; c_2 \mid v ::= a \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$
|boolean expressions| $b ::= \text{true} \mid \text{false} \mid a_1 \leq a_2 \mid b_1 \&\& b_2 \mid b_1 \mid b_2 \mid \neg b$
|arithmetic expressions| $a ::= n \mid v \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$
|variable names| $v$
|integer constants| $n$
Syntax vs. Semantics

- Syntactic definition imparts **no meaning** (semantics) of programs
  - Symbol “+” might not mean addition.
  - (Can you think of a language where it does not?)
- Elements of CFGs (e.g., defined by BNF) are *abstract syntax trees* (ASTs)
  - Use parentheses to describe AST’s structure
  - Example: \( x := (x + 1) \times 2; \text{skip}; y := x \)

![Abstract Syntax Tree Example]

- Parser transforms symbol stream into AST
  - Uses various precedence and associativity rules to auto-insert parenthhs
  - I’ll assume you know how that works (use a parser generator or take compilers/automata theory class).
  - When I write a program, it denotes an already-parsed AST.
CFG elements (e.g., programs) are finite but unbounded.

- **Finite:** The number of nodes in the AST equals a natural number (not infinity).
- **Unbounded:** For every program $c$ there exists a larger program $c'$.

The *set of all programs* is countably infinite.

- Countably infinite $=$ set has cardinality equal to the set of all natural numbers

But each individual program is finite.

- An infinite-sized program is actually a syntax error, because the CFG has no infinite-sized members.
Operational Semantics

- Operational semantics - mathematically define the meanings of programs in terms of the operation of an abstract machine

Stores

A store (machine state) in SIMPL is a partial function from variable names to integers:

\[ \Sigma = \{ v \mapsto \mathbb{Z} \} \]

\[ \sigma \in \Sigma \]

- (Partial) functions can be written as sets of input-output pairs:
  - Example: \( \sigma = \{(x, 8), (y, -10), (z, 0)\} \)
  - Not every set of input-output pairs is a function though, so be careful.
  - Non-function: \( \{(x, 8), (x, 10)\} \)
## Configurations and Judgments

### Configurations

A **configuration** is a command or expression paired with a store:

- command configurations: \(\langle c, \sigma \rangle\)
- arithmetic configurations: \(\langle a, \sigma \rangle\)
- boolean configurations: \(\langle b, \sigma \rangle\)

### Judgments

A **judgment** declares that a configuration **converges to** a store or value:

- command judgments: \(\langle c, \sigma \rangle \downarrow \sigma'\) (\(\sigma' \in \Sigma\))
- arithmetic judgments: \(\langle a, \sigma \rangle \downarrow n\) (\(n \in \mathbb{Z}\))
- boolean judgments: \(\langle b, \sigma \rangle \downarrow p\) (\(p \in \{T, F\}\))

“Converges to” (\(\downarrow\)) informally means “terminates and returns a value of ...”
We now have formalisms for talking about program behaviors (judgments), but we haven't defined which judgments are “true”.

Insight: Judgments are like mathematical propositions, but for a new math (computation).

How do we define “truth” in propositional logic? (Laws or Inference Rules)

Example: Law of Modus Ponens

\[
\begin{array}{c}
p \\
p \Rightarrow q \\
q
\end{array}
\]

Each rule written as a “fraction” with zero or more hypotheses on top, and a conclusion on the bottom

Free variables in rules are universally quantified

Rules can be nested to form tree-shaped derivations (proofs) of truth:

\[
\begin{array}{c}
p \\
p \Rightarrow q \\
q \\
q \Rightarrow r \\
r
\end{array}
\]

Solution: We need logical axioms that define computations in SIMPL!
Rule #1: Skip

Inference Rule (Axiom) for skip

\[
\langle \text{skip}, \sigma \rangle \Downarrow \sigma \tag{1}
\]

- no hypotheses = axiom
- True for every \( \sigma \) (i.e., \( \sigma \) is universally quantified)
Warning: Misnamed Variables

The following rule is *completely different*!

**A different (wrong) rule for skip**

\[
\langle \text{skip}, \sigma \rangle \Downarrow \sigma^{(1)}
\]

- What’s the difference?
- What does this rule effectively say skip does?
Rule #2: Sequence

Inference Rule for ;

\[
\frac{\ ? \quad \langle c_1 ; c_2 , \sigma \rangle \Downarrow \sigma' }{(2)}
\]
Rule #2: Sequence

Inference Rule for \( ; \)

\[
\frac{\langle c_1, \sigma \rangle \Downarrow \sigma_2}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma'} \tag{2}
\]
Rule #2: Sequence

Inference Rule for ;

\[
\frac{\langle c_1, \sigma \rangle \Downarrow \sigma_2 \quad \langle c_2, \sigma_2 \rangle \Downarrow \sigma'}{\langle c_1 ; c_2, \sigma \rangle \Downarrow \sigma'}^{(2)}
\]
Example Derivation

\[ \langle \text{skip}; (\text{skip}; \text{skip}), \sigma \rangle \Downarrow \ ? \]
Example Derivation

\[
\begin{align*}
\langle \text{skip}, \sigma \rangle \downarrow \text{??} & \quad \langle \text{skip}; \text{skip}, \text{??} \rangle \downarrow \text{??} \\
\hline
\langle \text{skip};(\text{skip};\text{skip}), \sigma \rangle \downarrow \text{??}^{(2)}
\end{align*}
\]
Example Derivation

\[
\begin{align*}
\langle \text{skip}, \sigma \rangle \Downarrow \sigma & \quad (1) \\
\langle \text{skip};\text{skip}, \sigma \rangle \Downarrow ? & \quad (2) \\
\langle \text{skip};(\text{skip};\text{skip}), \sigma \rangle \Downarrow ? &
\end{align*}
\]
Example Derivation

\[
\begin{align*}
\langle \text{skip}, \sigma \rangle & \downarrow \sigma & \langle \text{skip}, \sigma \rangle & \downarrow \text{??} & \langle \text{skip}, \text{??} \rangle & \downarrow \text{??} \\
\langle \text{skip}, \sigma \rangle & \downarrow \sigma & \langle \text{skip}; \text{skip}, \sigma \rangle & \downarrow \text{??} & \langle \text{skip}; (\text{skip}; \text{skip}), \sigma \rangle & \downarrow \text{??}
\end{align*}
\]
Example Derivation

\[
\frac{\langle \text{skip}, \sigma \rangle \Downarrow \sigma}{\langle \text{skip}, \sigma \rangle \Downarrow \sigma}^{(1)} \quad \frac{\langle \text{skip}, \sigma \rangle \Downarrow \sigma}{\langle \text{skip} ; \text{skip}, \sigma \rangle \Downarrow ?}^{(2)}
\]

\[
\frac{\langle \text{skip} ; \text{skip} ; \text{skip}, \sigma \rangle \Downarrow ?}{\langle \text{skip} ; (\text{skip} ; \text{skip}), \sigma \rangle \Downarrow ?}^{(2)}
\]
Example Derivation

\[
\begin{align*}
&\langle \text{skip}, \sigma \rangle \Downarrow \sigma \tag{1} \\
&\langle \text{skip;skip}, \sigma \rangle \Downarrow \sigma \tag{2}
\end{align*}
\]

\[
\frac{\langle \text{skip}, \sigma \rangle \Downarrow \sigma \quad \langle \text{skip}, \sigma \rangle \Downarrow \sigma \tag{1}}{\langle \text{skip;(skip;skip)}, \sigma \rangle \Downarrow \sigma \tag{2}}
\]
Building Derivations

Work bottom-up, left-to-right (usually).

- Identify (by number) which rule you’re using to the right of the bar.
- Instantiate rule variables consistently and uniformly at each rule use.
  - If $\sigma = \sigma_1$ in this rule instance, then every $\sigma$ appearing in the rule must be replaced with $\sigma_1$.
  - Treat rule literally, not what you expect/want it to say!

Derivations and infinity

- No infinite-sized derivations! (Each derivation must have strictly finite height.)
- The set of all derivations is countably infinite.
Rule #3: Assignment

Inference Rule for :=

\[\langle v := a, \sigma \rangle \Downarrow ?^{(3)}\]
Warning: Type-inconsistent Rules

First attempt at assignment rule:

Malformed (wrong) Rule for :=

\[ \langle v := a, \sigma \rangle \downarrow \sigma[v \mapsto a] \quad (3) \]

Notation (functional update):

\[ f[x \mapsto y] = (f - \{(x, z) \mid (x, z) \in f\}) \cup \{(x, y)\} \]

But the rule above is not a mathematically sensible definition. Why?
Rule #3: Assignment

Correct formulation of assignment rule:

Inference Rule for :=

\[
\frac{\langle a, \sigma \rangle \downarrow n}{\langle v := a, \sigma \rangle \downarrow \sigma[v \mapsto n]}^{(3)}
\]

Notation (functional update):

\[
f[x \mapsto y] = (f - \{(x, z) \mid (x, z) \in f\}) \cup \{(x, y)\}
\]
Rule #4: Conditional

Inference Rule for if-then-else

\[
\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \downarrow ?^{(4)}
\]
Solution: Multiple rules per syntactic form are perfectly valid and often useful.

Inference Rules for if-then-else

\[
\begin{align*}
\langle b, \sigma \rangle \Downarrow T & \quad \langle c_1, \sigma \rangle \Downarrow \sigma' \quad (4) \\
\langle b, \sigma \rangle \Downarrow F & \quad \langle c_2, \sigma \rangle \Downarrow \sigma' \quad (5) \\
\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle & \Downarrow \sigma'
\end{align*}
\]
Rule #6: While-loop

Inference Rule for while-loop

\[
\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \text{?}
\]

This is a tough one.
Rule #6: While-loop

The false part is easy, but what about the true part?

Inference Rules for while-loop

\[
\frac{\langle b, \sigma \rangle \Downarrow F}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma}
\]

\[
\frac{\langle b, \sigma \rangle \Downarrow T \quad ?}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow ?}
\]
Rule #6: While-loop

The false part is easy, but what about the true part?

Inference Rules for while-loop

\[
\begin{align*}
\langle b, \sigma \rangle \Downarrow F \\
\hline \\
\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma \\
\langle b, \sigma \rangle \Downarrow T & \quad \langle c, \sigma \rangle \Downarrow \sigma_2 \\
\hline \\
\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \&
\end{align*}
\]
Rule #6: While-loop

Idea: What about using the entire while-loop recursively?

Inference Rules for while-loop

\[
\begin{align*}
\langle b, \sigma \rangle & \Downarrow F \\
\langle \text{while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma \\
\langle b, \sigma \rangle & \Downarrow T \\
\langle c, \sigma \rangle & \Downarrow \sigma_2 \\
\langle \text{while } b \text{ do } c, \sigma_2 \rangle & \Downarrow \sigma' \\
\langle \text{while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma'
\end{align*}
\]

Danger: Is this rule circular?
Warning: Circular Rules

Warning: It is easy to create valid yet pointless rules using recursion.

Example of a valid yet pointless inference rule

\[
\frac{\langle \text{while } b \text{ do } c, \sigma \rangle \downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \downarrow \sigma'}
\]

- This inference rule is valid and sound.
- But it isn’t useful. (Recall that derivations are finite.)
Rule #6: While-loop

Is this rule pointless?

Inference Rules for while-loop

\[
\begin{align*}
\langle b, \sigma \rangle \Downarrow F & \quad \frac{\langle b, \sigma \rangle \Downarrow F}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma} \\
\langle b, \sigma \rangle \Downarrow T & \quad \langle c, \sigma \rangle \Downarrow \sigma_2 \quad \langle \text{while } b \text{ do } c, \sigma_2 \rangle \Downarrow \sigma' \\
& \quad \frac{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'}
\end{align*}
\]
Rule #6: While-loop

Is this rule pointless? No, this works! But let’s compact it into a single rule...

Inference Rules for while-loop

\[
\begin{align*}
\langle b, \sigma \rangle & \Downarrow F \\
\langle \text{while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma \\
\langle b, \sigma \rangle & \Downarrow T \\
\langle c, \sigma \rangle & \Downarrow \sigma_2 \\
\langle \text{while } b \text{ do } c, \sigma_2 \rangle & \Downarrow \sigma' \\
\langle \text{while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma'
\end{align*}
\]
Rule #6: While-loop

Inference Rule for while-loop

\[
\frac{\langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma \rangle \Downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'}^{(6)}
\]
Rule #6: While-loop

This single rule suffices because with it we can derive:

\[
\begin{align*}
\langle b, \sigma \rangle \Downarrow F & \quad \langle \text{skip}, \sigma \rangle \Downarrow \sigma \quad \text{(1)} \\
\langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma \rangle \Downarrow \sigma & \quad \text{(4)} \\
\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma' & \quad \text{(6)}
\end{align*}
\]

and

\[
\begin{align*}
\langle b, \sigma \rangle \Downarrow T & \quad \langle \text{while } b \text{ do } c, \sigma_2 \rangle \Downarrow \sigma' \quad \text{(2)} \\
\langle c, \sigma \rangle \Downarrow \sigma_2 & \quad \langle \text{while } b \text{ do } c, \sigma_2 \rangle \Downarrow \sigma' \quad \text{(5)} \\
\langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma \rangle \Downarrow \sigma' & \quad \text{(6)} \\
\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma' & \quad \text{(6)}
\end{align*}
\]
Other Rules

- Also need inference rules for arithmetic and boolean judgments
- See reference section of Assignment #2 for full list
- Mostly “obvious” but I’ll mention a few
Symbols vs. Mathematical Operators

Inference Rule for addition

\[
\frac{\langle a_1, \sigma \rangle \downarrow n_1 \quad \langle a_2, \sigma \rangle \downarrow n_2}{\langle a_1 + a_2, \sigma \rangle \downarrow n_1 + n_2}
\]

(15)

Recall: “+” is a symbol from the input stream (no mathematical significance), whereas “+” is the mathematical operator for 31-bit modular integer addition.
Reading the Store

To get variable values, we simply use $\sigma$ as a function.

**Inference Rule for variable-read**

\[
\begin{array}{c}
\langle v, \sigma \rangle \Downarrow \sigma(v) \\
\end{array}
\] (14)

- Rules with no premises are called **axioms**.
- When writing axioms, feel free to omit the fraction line.
Comprehending Inference Rules

Inference Rule for $\langle c_1, c_2, \sigma \rangle \Downarrow \sigma'$

$\langle c_1, \sigma \rangle \Downarrow \sigma_2 \quad \langle c_2, \sigma_2 \rangle \Downarrow \sigma' \\
\frac{\quad \langle c_1; c_2, \sigma \rangle \Downarrow \sigma'}{\quad (2)}$

- Two ways to understand each inference rule:
  1. Implementation recipe: “To compute $c_1; c_2$ on $\sigma$, first (recursively) compute $c_1$ on $\sigma$ to get $\sigma_2$, then (recursively) compute $c_2$ on $\sigma_2$ to get $\sigma'$.”
  2. Logical specification: “To prove that $c_1; c_2$ on $\sigma$ converges to $\sigma'$, it suffices to prove $c_1$ on $\sigma$ converges to some $\sigma_2$, and $c_2$ on $\sigma_2$ converges to $\sigma'$.”

- Big hint: Reading each rule as an implementation recipe essentially solves Assignment #2 for you. Your solution should be a nearly verbatim translation from the rules to code.

- Spanning the semantic gap
  - Rules are definitions, not theorems. So if you get them “wrong”, there’s no proof of wrongness. You’ve merely defined a really strange language.
  - Functional languages minimize the chance for error when mapping the math to code.