Large-step Operational Semantics

Large-step Judgments

A large-step judgment declares that a configuration converges to a store or value:

- **command judgments**  \[ \langle c, \sigma \rangle \Downarrow \sigma' \quad (\sigma' \in \Sigma) \]
- **arithmetic judgments**  \[ \langle a, \sigma \rangle \Downarrow n \quad (n \in \mathbb{Z}) \]
- **boolean judgments**  \[ \langle b, \sigma \rangle \Downarrow p \quad (p \in \{T, F\}) \]

where “converges to” (\(\Downarrow\)) means “terminates and returns a value of ...”

- **Advantages:**
  - relatively simple to reason about (few inference rules)
  - good when code correctness means returning the correct result

- **Disadvantages:**
  - mostly cannot prove things about non-terminating programs
  - insufficient when code correctness depends on what the program does as it executes (e.g., side-effects)
Small-step Operational Semantics

Alternative: Small-step Operational Semantics

Small-step Judgments

A small-step judgment declares that a configuration **steps to** a new configuration:

- **command judgments**
  \[ \langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle \]
- **arithmetic judgments**
  \[ \langle a, \sigma \rangle \rightarrow_1 \langle a', \sigma' \rangle \]
- **boolean judgments**
  \[ \langle b, \sigma \rangle \rightarrow_1 \langle b', \sigma' \rangle \]

where “steps to” \( \rightarrow_1 \) means “keeps executing in this next configuration.”

- **Advantages:**
  - can prove things about non-terminating code
  - can prove things about all machine states realized by a computation

- **Disadvantage:**
  - more complex (more rules)
  - harder to reason about terminating programs (more induction)
Small-step Rule for skip

\[ \langle \text{skip}, \sigma \rangle \rightarrow_1 \langle ?, ? \rangle \]
Small-step Rule for \textit{skip}

\[
\langle \text{skip}, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma \rangle
\]

This is incorrect. Why? What does this rule actually say?
Small-step Rule for \textit{skip}

Need a way to say that \textit{skip} has no “next configuration.” It’s done.

Solution: No rule for \textit{skip}!

Sometimes write: $\langle \text{skip}, \sigma \rangle \not\rightarrow_1$

\textbf{Definition (final configuration):} $\langle \text{skip}, \sigma \rangle$, $\langle n, \sigma \rangle$, $\langle \text{true}, \sigma \rangle$ and $\langle \text{false}, \sigma \rangle$ are final configurations for all $\sigma \in \Sigma$. 
Small-step Rule for Sequence

\[
\begin{align*}
\langle c_1 ; c_2, \sigma \rangle & \rightarrow_1 \langle ?, ? \rangle \\
\end{align*}
\]
Small-step Rule for Sequence

\[
\langle c_1, \sigma \rangle \rightarrow_1 \langle c'_1, \sigma' \rangle \\
\langle c_1 ; c_2, \sigma \rangle \rightarrow_1 \langle ?, ? \rangle
\]
Small-step Rule for Sequence

\[ \langle c_1, \sigma \rangle \rightarrow_1 \langle c'_1, \sigma' \rangle \]

\[ \langle c_1; c_2, \sigma \rangle \rightarrow_1 \langle c'_1; c_2, \sigma' \rangle \]
Small-step Rule for Sequence

\[
\langle c_1, \sigma \rangle \rightarrow_1 \langle c'_1, \sigma' \rangle \\
\langle c_1 ; c_2, \sigma \rangle \rightarrow_1 \langle c'_1 ; c_2, \sigma' \rangle \tag{S1}
\]

But how do we ever execute \( c_2 \)?

Need some way to say, “If \( c_1 \) can’t take any more steps, then work on \( c_2 \).”

“can’t take any more steps” = “final configuration”
Solution: Two rules

\[
\begin{align*}
\langle c_1, \sigma \rangle \xrightarrow{1} \langle c'_1, \sigma' \rangle \\
\langle c_1; c_2, \sigma \rangle \xrightarrow{1} \langle c'_1; c_2, \sigma' \rangle \\
\langle \text{skip}; c_2, \sigma \rangle \xrightarrow{1} \langle c_2, \sigma \rangle
\end{align*}
\]
Small-step Rule for Assignment

\[ \langle v := a, \sigma \rangle \rightarrow_1 \langle ?, ? \rangle \]
Small-step Rule for Assignment

\[
\langle v := a, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma[v \mapsto a] \rangle
\]
Small-step Rule for Assignment

\[ \langle v := a, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma[v \mapsto a] \rangle \]

This is type-incorrect because \( a \) is not necessarily an integer.
Small-step Rule for Assignment

\[
\begin{align*}
\langle a, \sigma \rangle & \rightarrow_1 \langle n, \sigma \rangle \\
\langle v := a, \sigma \rangle & \rightarrow_1 \langle \text{skip}, \sigma[v \mapsto n] \rangle
\end{align*}
\]

Still wrong: What if \( a \) takes many steps to finally yield an answer \( n \)?

Don’t confuse large-step and small-step semantics!
Solution: Again, two rules

\[
\begin{align*}
\langle a, \sigma \rangle & \rightarrow_1 \langle a', \sigma' \rangle \\
\langle v := a, \sigma \rangle & \rightarrow_1 \langle v := a', \sigma' \rangle \\
\langle v := n, \sigma \rangle & \rightarrow_1 \langle \text{skip}, \sigma[v \mapsto n] \rangle
\end{align*}
\]
Small-step Rules for Conditionals

For conditionals we need three rules:

\[
\langle b, \sigma \rangle \rightarrow_1 \langle b', \sigma' \rangle \quad (S5)
\]

\[
\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow_1 \langle \text{if } b' \text{ then } c_1 \text{ else } c_2, \sigma' \rangle \quad (S6)
\]

\[
\langle \text{if true then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle \quad (S6)
\]

\[
\langle \text{if false then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow_1 \langle c_2, \sigma \rangle \quad (S7)
\]
For while-loop we’ll use the same trick from large-step semantics:

\[
\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_1 \langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else } \text{skip}, \sigma \rangle^{(S8)}
\]
This completes the small-step rules for commands.

Also need rules for arithmetic and boolean judgments.
- Nothing particularly surprising, but requires lots of rules (26 total)
- See online lecture notes for full list.
- Exercise: See if you can figure them out on your own, then check the notes.
Definition (total relation): A relation $R$ is total if every domain element $x$ is related to a range element $y$ (i.e., $\forall x, \exists y, x \, R \, y$).

Question: Is $\downarrow$ a total relation?

In other words, is there any program $c$ and store $\sigma$ for which there is no $\sigma'$ satisfying $\langle c, \sigma \rangle \, \downarrow \, \sigma'$?
Totality

Definition (total relation): A relation $\mathcal{R}$ is total if every domain element $x$ is related to a range element $y$ (i.e., $\forall x, \exists y, x \mathcal{R} y$).

Question: Is $\downarrow$ a total relation?

In other words, is there any program $c$ and store $\sigma$ for which there is no $\sigma'$ satisfying $\langle c, \sigma \rangle \downarrow \sigma'$?

Answer: $\downarrow$ is not total. For example, if we choose $c = \text{while true do skip}$ (and any $\sigma$), there is no $\sigma'$ satisfying $\langle c, \sigma \rangle \downarrow \sigma'$.

Follow-up question: What about aside from infinite loops?
Definition (total relation): A relation $\mathcal{R}$ is total if every domain element $x$ is related to a range element $y$ (i.e., $\forall x, \exists y, x \mathcal{R} y$).

Question: Is $\downarrow$ a total relation?

Answer: $\downarrow$ is not total. For example, if we choose $c = \text{while true do skip}$ (and any $\sigma$), there is no $\sigma'$ satisfying $\langle c, \sigma \rangle \downarrow \sigma'$.

Follow-up question: What about aside from infinite loops?

Answer: We could also choose $c = (x := y)$ and any $\sigma$ such that $y \notin \sigma^{-}$.

Two cases of non-totality:

1. infinite loops (limitation of large-step semantics)
2. uninitialized reads (intentionally implementation-defined)
Definition (ambiguity): A derivation system is said to be **ambiguous** if there exists a judgment having multiple distinct derivations.

**Question:** Are our large-step semantics ambiguous?

In other words, is there some judgment $\langle c, \sigma \rangle \Downarrow \sigma'$ that is derivable in two different ways?
**Definition (ambiguity):** A derivation system is said to be **ambiguous** if there exists a judgment having multiple distinct derivations.

**Question:** Are our large-step semantics ambiguous?

In other words, is there some judgment $\langle c, \sigma \rangle \Downarrow \sigma'$ that is derivable in two different ways?

**Answer:** No. For every judgment that’s derivable, there’s only one way to derive it.
Ambiguity

Derivation systems for real languages usually have ambiguity (and that’s okay because it gives implementors choices).

Example: Adding these rules makes our system ambiguous.

\[
\begin{align*}
\langle b_1, \sigma \rangle & \Downarrow F \\
\langle b_1 \&\& b_2, \sigma \rangle & \Downarrow F \\
\langle b_2, \sigma \rangle & \Downarrow F \\
\langle b_1 \&\& b_2, \sigma \rangle & \Downarrow F
\end{align*}
\]
Definition (deterministic): A relation \( R \) is deterministic (also called a function) if every domain element \( x \) is related to at most one range element \( y \) (i.e., \( \forall x, \forall y_1, y_2, (x \ R \ y_1) \land (x \ R \ y_2) \Rightarrow y_1 = y_2 \)).

Question: Is \( \downarrow \) deterministic?
Determinism

**Definition (deterministic):** A relation $\mathcal{R}$ is deterministic (also called a function) if every domain element $x$ is related to at most one range element $y$ (i.e., $\forall x, \forall y_1, y_2, (x \mathcal{R} y_1) \land (x \mathcal{R} y_2) \Rightarrow y_1 = y_2$).

**Question:** Is $\downarrow$ deterministic?

**Answer:** Yes.
Determinism

Our system would become non-deterministic if we added something like this:

\[ a ::= \cdots | \text{rand} \]

\[
\begin{align*}
  n \in \mathbb{Z} \\
  \langle \text{rand}, \sigma \rangle \Downarrow n
\end{align*}
\]