CSc 520

Principles of Programming Languages

52: Semantics — Operational Semantics

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

Operational Semantics

- In this lecture we will describe a method for the formal specification of programming languages, known as structural operational semantics.
- Operation semantics describes how a computation is performed.
- We've already seen some operational semantic methods:
  1. A metacircular interpreter for Scheme,
  2. The β-reduction rule with the normal order reduction strategy for λ-calculus.

Operational Semantics. . .

- An operational semantics operates on a sequence of states.
- We start with an initial state.
- The program maps the initial state into a new state, and so on, until a final state is reached, the result of the program.

Operational Semantics — λ-calculus

- A configuration consists of the lambda expression that's left to reduce.
- The transition function applies β and δ-reductions according to our evaluation strategy.
- The evaluation terminates when a configuration is in normal form.

Structural Operational Semantics

- Operational semantics specifies the semantics of a language in terms of how it would be executed on an abstract machine.
- In Structural Operational Semantics (SOS), definitions are given by inference rules.
- SOS turns the abstract machine into a system of logical inferences.

Inference Rules

Inference Rules

- Inference rules:

$$\frac{\text{premise}_1 \qquad \text{premise}_1 \qquad \cdots \qquad \text{premise}_n}{\text{conclusion}}$$

The conclusion follows from the premises.

- Sometimes we add a condition under which the rule is applicable:

$$\frac{\text{premise}_1 \qquad \text{premise}_1 \qquad \cdots \qquad \text{premise}_n}{\text{conclusion}} \quad \text{Condition}$$

Inference Rules. . .

- Sometimes there are no premises (this is called an axiom:

$$\frac{}{\text{conclusion}}$$

in which case we omit the line:

$$\text{conclusion}$$

Inference Rules — Examples

- Example from Natural Deduction (logical properties of conjunction (and)):

$$\frac{p \wedge q}{p} \qquad \frac{p \wedge q}{q} \qquad \frac{p \quad q}{p \wedge q}$$

- Example from Natural Deduction (introduction of universal quantifier):

$$\frac{P(a)}{\forall x P(x)}$$ a does not occur in P(x) or in any assumption on which P(a) depends

Abstract Syntax of Wren

Abstract Syntax

- The operational semantics operates on a abstract syntax of the language.
- The abstract syntax ignores the surface syntax and only specifies the “essential” parts of each language construct.
- We can use any convenient way to define the abstract syntax, EBNF, for example.
- We use inference rules to specify the abstract syntax.
- The next slide gives the types of objects that we use to construct the abstract syntax from.

Syntactic Categories

$\mathrm{n} \in \mathrm{Num}$	=	Set of numerals
$\mathrm{b} \in \{\mathbf{true}, \mathbf{false}\}$	=	Set of Boolean values
$\mathrm{id} \in \mathrm{Id}$	=	Set of integer identifiers
$\mathrm{bid} \in \mathrm{Bid}$	=	Set of Boolean identifiers
$\mathrm{iop} \in \mathrm{Iop}$	=	$\{+, -, *, /\}$
$\mathrm{rop} \in \mathrm{Rop}$	=	$\{<, \leq, =, \geq, >, <>\}$
$\mathrm{bop} \in \mathrm{Bop}$	=	$\{\mathbf{and}, \mathbf{or}\}$
$\mathrm{ie} \in \mathrm{Iexp}$	=	Set of integer expressions
$\mathrm{be} \in \mathrm{Bexp}$	=	Set of Boolean expressions
$\mathrm{c} \in \mathrm{Cmd}$	=	Set of commands

Abstract Syntax — Integers

- Objects from Num may serve as integer expressions:

$$n : \text{iexp} \quad n \in \text{Num}$$

Note that this rule is an abbreviated form of

$$\frac{}{n : \text{iexp}} \quad n \in \text{Num}$$

- Integer identifiers may serve as integer expressions:

$$\text{id} : \text{iexp} \quad \text{id} \in \text{Id}$$

Abstract Syntax — Integers...

- Integer unary/binary/relational expressions:

$$\frac{\text{ie}_1 : \text{iexp} \qquad \text{ie}_2 : \text{iexp}}{\text{ie}_1 \text{ iop } \text{ie}_2 : \text{iexp}} \quad \text{iop} \in \text{Iop}$$

$$\frac{\text{ie}_1 : \text{iexp} \qquad \text{ie}_2 : \text{iexp}}{\text{ie}_1 \text{ rop } \text{ie}_2 : \text{bexp}} \quad \text{rop} \in \text{Rop} \qquad \frac{\text{ie} : \text{iexp}}{-\ \text{ie} : \text{iexp}}$$

- Remember that

$$\begin{aligned} \text{iop} \in \text{Iop} &= \{+, -, *, /\} \\ \text{rop} \in \text{Rop} &= \{<, \leq, =, \geq, >, <>\} \\ \text{ie} \in \text{Iexp} &= \text{Set of integer expressions} \end{aligned}$$

Abstract Syntax — Booleans

- Objects from Num may serve as boolean expressions:

$$b : \text{bexp} \quad b \in \{\textbf{true}, \textbf{false}\}$$

- Boolean identifiers may serve as boolean expressions:

$$\text{bid} : \text{bexp} \quad \text{bid} \in \text{Bid}$$

- Boolean binary expressions:

$$\frac{\text{be}_1 : \text{bexp} \qquad \text{be}_2 : \text{bexp}}{\text{be}_1 \text{ bop } \text{be}_2 : \text{bexp}} \quad \text{bop} \in \text{Bop} \qquad \frac{\text{be} : \text{bexp}}{\textbf{not}\ \text{be} : \text{bexp}}$$

Abstract Syntax — Commands

$$\textbf{skip} : \text{cmd} \qquad \frac{\text{ie} : \text{iexp}}{\text{id} \ := \ \text{ie} : \text{cmd}} \quad \text{id} \in \text{Id}$$

$$\frac{\text{be} : \text{bexp}}{\text{bid} \ := \ \text{be} : \text{cmd}} \quad \text{bid} \in \text{Bid} \qquad \textbf{read}\ \text{id} : \text{cmd} \quad \text{id} \in \text{Id}$$

$$\frac{\text{ie} : \text{iexp}}{\textbf{write}\ \text{ie} : \text{cmd}}$$

Abstract Syntax — Commands. . .

$$\frac{\text{be} : \text{bexp} \qquad c : \text{cmd}}{\textbf{if}\ \text{be}\ \textbf{then}\ c : \text{cmd}} \qquad \frac{\text{be} : \text{bexp} \qquad c_1 : \text{cmd} \qquad c_2 : \text{cmd}}{\textbf{if}\ \text{be}\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \text{cmd}}$$

$$\frac{c_1 : \text{cmd} \qquad c_2 : \text{cmd}}{c_1\ ;\ c_2 : \text{cmd}} \qquad \frac{\text{be} : \text{bexp} \qquad c : \text{cmd}}{\textbf{while}\ \text{be}\ \textbf{do}\ c : \text{cmd}}$$

Abstract Syntax — Example

- We can show that the following is a valid Wren command by deriving

$$x := 5\ ;\ \textbf{while}\ \textbf{not}(x = 0)\ \textbf{do}\ x := x - 1\ ;\ \textbf{write}\ x : \text{cmd}$$

$$\frac{\dfrac{\dfrac{x : \text{iexp} \qquad 0 : \text{iexp}}{x = 0 : \text{bexp}}}{\textbf{not}(x = 0) : \text{bexp}} \qquad \dfrac{\dfrac{x : \text{iexp} \qquad 1 : \text{iexp}}{x - 1 : \text{iexp}}}{x := x - 1 : \text{cmd}}}{\textbf{while}\ \textbf{not}(x = 0)\ \textbf{do}\ x := x - 1 : \text{cmd}}$$

- The next slide shows the complete derivation.

Abstract Syntax — Example. . .

$$\frac{\dfrac{5 : \text{iexp}}{x := 5 : \text{cmd}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{x : \text{iexp} \qquad 0 : \text{iexp}}{x = 0 : \text{bexp}}}{\textbf{not}(x = 0) : \text{bexp}} \qquad \dfrac{\dfrac{x : \text{iexp} \qquad 1 : \text{iexp}}{x - 1 : \text{iexp}}}{x := x - 1 : \text{cmd}}}{\textbf{while}\ \textbf{not}(x = 0)\ \textbf{do}\ x := x - 1 : \text{cmd}} \qquad \dfrac{x : \text{iexp}}{\textbf{write}\ x : \text{cmd}}}{\textbf{while}\ \textbf{not}(x = 0)\ \textbf{do}\ x := x - 1\ ;\ \textbf{write}\ x : \text{cmd}}}{x := 5\ ;\ \textbf{while}\ \textbf{not}(x = 0)\ \textbf{do}\ x := x - 1\ ;\ \textbf{write}\ x : \text{cmd}}$$

Semantics of Wren Expressions

Semantics of Wren Expressions

- SOS applies sequences of transformations to the abstract syntax of a program. Each step produces a new configuration.
- Three things can happen:
  1. We arrive at a normal form configuration where no more transformations can be applied. This normal form is the meaning of the program.
  2. Computation gets stuck in a configuration from which no more transformations are possible. For example, the program might try to divide-by-zero.
  3. We find ourselves in a non-terminating sequence of configurations.

Store Abstract Data Type

- Wren expressions can contain identifiers.
- We model the set of identifiers available to an expression using a store abstraction.
- The store is a set of pairs

$$\mathrm{sto} = \{x \mapsto 3, y \mapsto 5, p \mapsto \mathbf{true}\}$$

  mapping identifiers to their bound values.
- The domain of the store is the set of defined variables:

$$\mathrm{dom(sto)} = \{x, y, z\}$$

Store Abstract Data Type…

- $\mathit{emptySto}$ is a store where all identifiers are undefined.
- $\mathit{updateSto}(\mathrm{sto}, \mathrm{id}, n)$ extends sto with a new binding $\mathrm{id} \mapsto n$.
- $\mathit{applySto}(\mathrm{sto}, \mathrm{id})$ returns the value associated with id.
- For example, the store

$$\{x \mapsto 3, y \mapsto 5, p \mapsto \mathbf{true}\}$$

  maps three variables to their respective values.
- It would be created and queried like this:

$$\mathit{updateSto}(\mathit{updateSto}(\mathit{updateSto}(\mathit{emptySto}, p, \mathbf{true}), y, 5), x, 3)$$

$$\mathit{applySto}(\{x \mapsto 3, y \mapsto 5, p \mapsto \mathbf{true}\}, p) \Rightarrow \mathbf{true}$$

Configurations

- In our inference system a configuration looks like this:

$$\langle \mathit{expression}, \mathit{store} \rangle$$

- $\mathit{expression}$ is the expression we're examining.
- store is the context (set of identifiers) in which it should be evaluated.
- In Wren, a final configuration (a normal form) is an integer (n) or boolean (b) constant value:

$$\langle n, \mathrm{sto} \rangle \text{ or } \langle b, \mathrm{sto} \rangle$$

- Note that (unlike C, for example) Wren expressions are pure (they have no side effects), and hence they never alter the store.

Inference System

- Our inference system is given in the next few slides.
- There is one rule for each abstract syntax form that is not in normal form.
- $\longrightarrow$ represents a transition from one configuration to the next.
- For example, the rule

$$\langle n_1 \; + \; n_2, \mathrm{sto}\rangle \longrightarrow \langle \mathit{compute}(+, n_1, n_2), \mathrm{sto}\rangle$$

takes the configuration $\langle n_1 \; + \; n_2, \mathrm{sto}\rangle$ (where n_1 and n_2 are two integer values) into a new configuration consisting of their sum.

Inference Rules for Expressions

- These two rules enforce a left-to-right evaluation strategy for binary expressions.
- I.e., in $E_1 + E_2$, E_1 must be reduced to a constant integer before E_2 can be evaluated.

$$\frac{\langle \mathrm{ie}_1, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{ie}_1', \mathrm{sto}\rangle}{\langle \mathrm{ie}_1 \; \mathrm{iop} \; \mathrm{ie}_2, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{ie}_1' \; \mathrm{iop} \; \mathrm{ie}_2, \mathrm{sto}\rangle}$$

$$\frac{\langle \mathrm{ie}_2, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{ie}_2', \mathrm{sto}\rangle}{\langle n \; \mathrm{iop} \; \mathrm{ie}_2, \mathrm{sto}\rangle \longrightarrow \langle n \; \mathrm{iop} \; \mathrm{ie}_2', \mathrm{sto}\rangle}$$

Inference Rules for Expressions...

- These rules enforce the same evaluation strategy for boolean and relational binary expressions:

$$\frac{\langle \mathrm{ie}_1, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{ie}_1', \mathrm{sto}\rangle}{\langle \mathrm{ie}_1 \; \mathrm{rop} \; \mathrm{ie}_2, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{ie}_1' \; \mathrm{rop} \; \mathrm{ie}_2, \mathrm{sto}\rangle}$$

$$\frac{\langle \mathrm{ie}_2, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{ie}_2', \mathrm{sto}\rangle}{\langle n \; \mathrm{rop} \; \mathrm{ie}_2, \mathrm{sto}\rangle \longrightarrow \langle n \; \mathrm{rop} \; \mathrm{ie}_2', \mathrm{sto}\rangle}$$

$$\frac{\langle \mathrm{be}_1, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{be}_1', \mathrm{sto}\rangle}{\langle \mathrm{be}_1 \; \mathrm{bop} \; \mathrm{be}_2, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{be}_1' \; \mathrm{bop} \; \mathrm{be}_2, \mathrm{sto}\rangle}$$

$$\frac{\langle \mathrm{be}_2, \mathrm{sto}\rangle \longrightarrow \langle \mathrm{be}_2', \mathrm{sto}\rangle}{\langle b \; \mathrm{bop} \; \mathrm{be}_2, \mathrm{sto}\rangle \longrightarrow \langle b \; \mathrm{bop} \; \mathrm{ie}_2', \mathrm{sto}\rangle}$$

Inference Rules for Expressions...

- When both arguments of a binary expression are integers, they can be calculated using a built-in function $\mathit{compute}(\mathrm{op}, n_1, n_2)$:

$$\langle n_1 \; \mathrm{iop} \; n_2, \mathrm{sto}\rangle \longrightarrow \langle \mathit{compute}(\mathrm{iop}, n_1, n_2), \mathrm{sto}\rangle$$

$$(\mathrm{iop} \neq /) \text{ or } (n_2 \neq 0)$$

The computation gets stuck if we try to divide by zero.

- Here's the same rule for boolean and relational expressions:

$$\langle n_1 \; \mathrm{rop} \; n_2, \mathrm{sto}\rangle \longrightarrow \langle \mathit{compute}(\mathrm{rop}, n_1, n_2), \mathrm{sto}\rangle$$

$$\langle b_1 \; \mathrm{bop} \; b_2, \mathrm{sto}\rangle \longrightarrow \langle \mathit{compute}(\mathrm{bop}, b_1, b_2), \mathrm{sto}\rangle$$

Inference Rules for Expressions...

- Rules for boolean **not**:

$$\frac{\langle \text{be}, \text{sto}\rangle \to \langle \text{be}', \text{sto}\rangle}{\langle \textbf{not}(\text{be}), \text{sto}\rangle \to \langle \textbf{not}(\text{be}'), \text{sto}\rangle}$$

$$\langle \textbf{not}(\textbf{true}), \text{sto}\rangle \to \langle \textbf{false}, \text{sto}\rangle$$

$$\langle \textbf{not}(\textbf{false}), \text{sto}\rangle \to \langle \textbf{true}, \text{sto}\rangle$$

Inference Rules for Expressions...

- Rules for looking up identifiers in the store:

$$\langle \text{id}, \text{sto}\rangle \to \langle \textit{applySto}(\text{sto}, \text{id}), \text{sto}\rangle \quad \text{id} \in \text{dom}(\text{sto})$$

$$\langle \text{bid}, \text{sto}\rangle \to \langle \textit{applySto}(\text{sto}, \text{bid}), \text{sto}\rangle \quad \text{bid} \in \text{dom}(\text{sto})$$

Inference Rules for Expressions...

- The meaning of an expression is a sequence of configurations where each step is justified by one of our inference rules:

$$\langle e_1, \text{sto}\rangle \to \langle e_2, \text{sto}\rangle \to \langle e_3, \text{sto}\rangle \to \ldots \to \langle e_n, \text{sto}\rangle$$

- We can add a final rule to make the $\to$ relation transitive:

$$\frac{\langle e_1, \text{sto}\rangle \to \langle e_2, \text{sto}\rangle \qquad \langle e_2, \text{sto}\rangle \to \langle e_3, \text{sto}\rangle}{\langle e_1, \text{sto}\rangle \to \langle e_3, \text{sto}\rangle}$$

Expression Example

- Evaluate the expression

$$x + (y + 6)$$

given the store

$$\text{s} = \{x \mapsto 17, y \mapsto 25\}$$

- We first get:

$$S \equiv \frac{\dfrac{\dfrac{\langle y, \text{s}\rangle \to \langle 25, \text{s}\rangle}{\langle y+6, \text{s}\rangle \to \langle 25+6, \text{s}\rangle} \qquad \langle 25+6, \text{s}\rangle \to \langle 31, \text{s}\rangle}{\langle y+6, \text{s}\rangle \to \langle 31, \text{s}\rangle}}{\langle 17+(y+6), \text{s}\rangle \to \langle 17+31, \text{s}\rangle}$$

Expression Example...

$$S \equiv \cfrac{\cfrac{\cfrac{\langle y, \mathrm{s}\rangle \rightarrow \langle 25, \mathrm{s}\rangle}{\langle y+6, \mathrm{s}\rangle \rightarrow \langle 25+6, \mathrm{s}\rangle} \qquad \langle 25+6, \mathrm{s}\rangle \rightarrow \langle 31, \mathrm{s}\rangle}{\langle y+6, \mathrm{s}\rangle \rightarrow \langle 31, \mathrm{s}\rangle}}{\langle 17+(y+6), \mathrm{s}\rangle \rightarrow \langle 17+31, \mathrm{s}\rangle}$$

- Continuing, we get:

$$\cfrac{\cfrac{\langle x, \mathrm{s}\rangle \rightarrow \langle 17, \mathrm{s}\rangle}{\langle x+(y+6), \mathrm{s}\rangle \rightarrow \langle 17+(y+6), \mathrm{s}\rangle} \qquad S \qquad \langle 17+31, \mathrm{s}\rangle \rightarrow \langle 48, \mathrm{s}\rangle}{\langle x+(y+6), \mathrm{s}\rangle \rightarrow \langle 48, \mathrm{s}\rangle}$$

Semantics of Wren Commands

Configurations

- The assignment and **read** commands in Wren can affect the store.
- The **read** and **write** statements affect the input and output streams.
- Our configuration for commands reflects this:

$$\langle c, \mathrm{st}(\mathrm{in}, \mathrm{out}, \mathrm{sto})\rangle$$

c is a command, sto is the current store, in is the current list of inputs to the program (numbers), and out is the current list of outputs.

- For example,

$$\langle \mathbf{write}\ x, \mathrm{st}([1,2], [\,], \{x \mapsto 13)\}\rangle$$

Configurations...

- A computation proceeds by applying a sequence of inference rules, yielding a sequence of configurations:

$$\langle c_0, \mathrm{st}(\mathrm{in}_0, \mathrm{out}_0, \mathrm{sto}_0)\rangle \rightarrow \langle c_1, \mathrm{st}(\mathrm{in}_1, \mathrm{out}_1, \mathrm{sto}_1)\rangle \rightarrow \ldots$$

- For example,

$$\begin{array}{rl} \langle \mathbf{write}\ x, \mathrm{st}([1,2], [\,], \{x \mapsto 13)\}\rangle & \rightarrow \\ \langle \mathbf{write}\ 13, \mathrm{st}([1,2], [\,], \{x \mapsto 13)\}\rangle & \rightarrow \\ \langle \mathbf{skip}, \mathrm{st}([1,2], [13], \{x \mapsto 13)\}\rangle & \ldots \end{array}$$

Configurations...

- The normal form of a computation is

$$\langle \mathbf{skip}, \text{state} \rangle$$

where state is the final result of the computation.

- There are three possible outcomes of a computation:
  1. We reach a final $\langle \mathbf{skip}, \text{state} \rangle$ configuration.
  2. We reach a configuration from which no further transition is possible: a divide-by-zero error or a read from an empty input list.
  3. We enter a non-terminating **while**-loop, resulting in an infinite sequence of configurations:

$$\langle \mathbf{while\ true\ do\ skip}, \text{state} \rangle$$

Command Inference Rules — Assign

- These rules force the expression in an assignment statement to be evaluated:

$$\frac{\langle \text{ie}, \text{sto} \rangle \rightarrow \langle \text{ie}', \text{sto} \rangle}{\langle \text{id} := \ \text{ie}, \text{st}(\text{in}, \text{out}, \text{sto}) \rangle \rightarrow \langle \text{id} := \ \text{ie}', \text{st}(\text{in}, \text{out}, \text{sto}) \rangle}$$

$$\langle \text{id} := \ n, \text{st}(\text{in}, \text{out}, \text{sto}) \rangle \rightarrow \langle \mathbf{skip}, \text{st}(\text{in}, \text{out}, \mathit{updateSto}(\text{sto}, \text{id}, n)) \rangle$$

- In Wren the order of evaluation in an assignment statement doesn't matter (since the left-hand-side must be a pure identifier), but this isn't true in C:

```
A[c++] = A[c++];
```


Command Inference Rules — Assign

- Same as last slide, but for booleans:

$$\frac{\langle \text{be}, \text{sto} \rangle \rightarrow \langle \text{be}', \text{sto} \rangle}{\langle \text{bid} := \ \text{be}, \text{st}(\text{in}, \text{out}, \text{sto}) \rangle \rightarrow \langle \text{bid} := \ \text{be}', \text{st}(\text{in}, \text{out}, \text{sto}) \rangle}$$

$$\langle \text{bid} := \ b, \text{st}(\text{in}, \text{out}, \text{sto}) \rangle \rightarrow \langle \mathbf{skip}, \text{st}(\text{in}, \text{out}, \mathit{updateSto}(\text{sto}, \text{bid}, b)) \rangle$$

Command Inference Rules — if

$$\frac{\langle \text{be}, \text{sto} \rangle \rightarrow \langle \text{be}', \text{sto} \rangle}{\langle \mathbf{if}\ \text{be}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \text{st}(\text{in}, \text{out}, \text{sto}) \rangle \rightarrow \langle \mathbf{if}\ \text{be}'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \text{st}(\text{in}, \text{out}, \text{sto}) \rangle}$$

$$\langle \mathbf{if\ true\ then}\ c_1\ \mathbf{else}\ c_2, \text{state} \rangle \rightarrow \langle c_1, \text{state} \rangle$$

$$\langle \mathbf{if\ false\ then}\ c_1\ \mathbf{else}\ c_2, \text{state} \rangle \rightarrow \langle c_2, \text{state} \rangle$$

- **if-then** is transformed into an **if-then-else** which can be further transformed by the rules above:

$$\langle \mathbf{if}\ \text{be}\ \mathbf{then}\ c, \text{state} \rangle \rightarrow \langle \mathbf{if}\ \text{be}\ \mathbf{then}\ c\ \mathbf{else\ skip}, \text{state} \rangle$$

Command Inference Rules — while

- **while**-statements are defined in terms of themselves:

$$\langle \textbf{while}\ \text{be}\ \textbf{do}\ c, \text{state}\rangle \rightarrow$$
$$\langle \textbf{if}\ \text{be}\ \textbf{then}\ (c\ ;\ \textbf{while}\ \text{be}\ \textbf{do}\ c)\ \textbf{else}\ \textbf{skip}, \text{state}\rangle$$

Command Inference Rules — Sequencing

- Command sequencing ($c_1; c_2$) is done in two stages. First c_1 is reduced to **skip**:

$$\frac{\langle c_1, \text{state}\rangle \rightarrow \langle c_1', \text{state}'\rangle}{\langle c_1\ ;\ c_2, \text{state}\rangle \rightarrow \langle c_1';\ c_2, \text{state}'\rangle}$$

  then **skip** is discarded:

$$\langle \textbf{skip}\ ;\ c, \text{state}\rangle \rightarrow \langle c, \text{state}\rangle$$

Command Inference Rules — read

- **read** reads a number from the input list and stores it to a variable:

$$\langle \textbf{read}\ \text{id}, \text{st}(\text{in}, \text{out}, \text{sto})\rangle \rightarrow$$
$$\langle \textbf{skip}, \text{st}(\mathit{tail}(\text{in}), \text{out}, \mathit{updateSto}(\text{sto}, \text{id}, \mathit{head}(\text{in})))\rangle$$

$$\text{in} \neq [\,]$$

- Read fails if the input is empty.

Command Inference Rules — write

- **write** evaluates its argument expression to a number

$$\frac{\langle \text{ie}, \text{sto}\rangle \rightarrow \langle \text{ie}', \text{sto}\rangle}{\langle \textbf{write}\ \text{ie}, \text{st}(\text{in}, \text{out}, \text{sto})\rangle \rightarrow \langle \textbf{write}\ \text{ie}', \text{st}(\text{in}, \text{out}, \text{sto})\rangle}$$

  and appends it to the output list:

$$\langle \textbf{write}\ n, \text{st}(\text{in}, \text{out}, \text{sto})\rangle \rightarrow \langle \textbf{skip}, \text{st}(\text{in}, \mathit{affix}(\text{out}, n), \text{sto})\rangle$$

An Example

Input Program

```
x :=  0;
read z;
while z>=0 do
   ((if z>x then x := z);read z);
write x
```

$$c_1 = (x{:=}\ 0)$$
$$c_2 = \textbf{read } z$$
$$c_3 = \textbf{while } z >= 0 \textbf{ do } ((\textbf{if } z > x \textbf{ then } x{:=}\ z); \textbf{read } z)$$
$$c_4 = \textbf{write } x$$
$$c_w = (\textbf{if } z > x \textbf{ then } x{:=}\ z); \textbf{read } z$$
$$\{\} = \textit{emptySto}$$

Transitions

$$\langle x{:=}\ 0, \mathrm{st}([5,8,3,-1],[\,],\{\})\rangle \to \quad \langle \textbf{skip}, \mathrm{st}([5,8,3,-1],[\,],\{x \mapsto 0\})\rangle$$

$$\langle x{:=}\ 0; c_2; c_3; c_4, \mathrm{st}([5,8,3,-1],[\,],\{\})\rangle \to$$
$$\langle \textbf{read } z; c_3; c_4, \mathrm{st}([5,8,3,-1],[\,],\{x \mapsto 0\})\rangle$$

$$\langle \textbf{read } z, \mathrm{st}([5,8,3,-1],[\,],\{x \mapsto 0\})\rangle \to$$
$$\langle \textbf{skip}, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle$$

$$\langle \textbf{read } z; c_3; c_4, \mathrm{st}([5,8,3,-1],[\,],\{x \mapsto 0\})\rangle \to$$
$$\langle (\textbf{while } z \geq 0 \textbf{ do } c_w); c_4, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle$$

$$\langle \textbf{while } z \geq 0 \textbf{ do } c_w, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle \to$$
$$\langle \textbf{if } z \geq 0 \textbf{ then } (c_w; c_3) \textbf{else skip}, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle$$

Transitions. . .

$$\langle z \geq 0, \{x \mapsto 0, z \mapsto 5\}\rangle \to \quad \langle \textbf{true}, \{x \mapsto 0, z \mapsto 5\}\rangle$$

$$\langle \textbf{if } z \geq 0 \textbf{ then } (c_w; c_3) \textbf{else skip}, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle \quad \to$$
$$\langle \textbf{if true then } (c_w; c_3) \textbf{else skip}, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle$$

$$\langle \textbf{if true then } (c_w; c_3) \textbf{else skip}, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle \to$$
$$\langle c_w; c_3, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle$$

$$\langle c_w; c_3, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle =$$
$$\langle (\textbf{if } z > x \textbf{ then } x{:=}\ z); \textbf{read } z; \textbf{while } z \geq 0 \textbf{ do } c_w, \mathrm{st}([8,3,-1],[\,],\cdots\})\rangle$$

$$\langle z > x, \{x \mapsto 0, z \mapsto 5\}\rangle \to \quad \langle \textbf{true}, \{x \mapsto 0, z \mapsto 5\}\rangle$$

$$\langle \textbf{if } z > x \textbf{ then } x{:=}\ z, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle \to$$
$$\langle \textbf{if true then } x{:=}\ z, \mathrm{st}([8,3,-1],[\,],\{x \mapsto 0, z \mapsto 5\})\rangle$$

A Haskel Implementation

Abstract Syntax for Integer Expressions

```
data IOp      = Add | Mul

instance Show  IOp where
  show Add = "+"
  show Mul = "*"

data IntExpr = Iid String |
               Ilit Int |
               Ibin IOp IntExpr IntExpr

instance Show  IntExpr where
  show (Iid s) = s
  show (Ilit i) = show i
  show (Ibin op l r) = "("++show l++")"++" "++show op++
                       " ("++show r++")"
```


Configuration for Expressions

```
data ExprConf = IExpr IntExpr Store |
                BExpr BoolExpr Store

instance Show ExprConf where
   show (IExpr i s) = "<"++show i++","++show s++">"
   show (BExpr i s) = "<"++show i++","++show s++">"
```


Basic Computation

```
icompute :: IOp -> Int -> Int -> Int
icompute Add i1 i2 = i1+i2
icompute Mul i1 i2 = i1*i2
```

Transition Function

```
etransition :: ExprConf -> ExprConf
etransition (IExpr (Iid s) sto) =
    (IExpr (Ilit (intValue (applySto s sto))) sto)

etransition (IExpr (Ibin op (Ilit i1) (Ilit i2)) sto) =
    (IExpr (Ilit (icompute op i1 i2)) sto)

etransition (IExpr (Ibin op (Ilit i1) e2) sto) =
    (IExpr (Ibin op (Ilit i1) (extract c)) sto)
                    where c = etransition (IExpr e2 sto)
                          extract (IExpr e _) = e

etransition (IExpr (Ibin op e1 e2) sto) =
    (IExpr (Ibin op (extract(
               etransition (IExpr e1 sto))) e2) sto)
                    where extract (IExpr e _) = e
```


Top-level Evaluation Function

```
enormal :: ExprConf -> Bool
enormal (IExpr (Ilit _) _) = True
enormal (BExpr (Blit _) _) = True
enormal _ = False

ederive :: ExprConf -> ExprConf
ederive = until enormal etransition
```


Testing

```
test = (IExpr
           (Ibin Add
               (Ilit 1)
               (Ibin Add
                     (Ilit 1)
                     (Ilit 2))) [])

Main> etransition test
<(1) + (3),[]>

Main> etransition (etransition test)
<4,[]>

Main> ederive test
<4,[]>
```


The Store

```
data Value     = IVal Int | BVal Bool
data Map       = M String Value
type Store     = [Map]

emptySto:: Store
emptySto = []

updateSto:: String -> Value -> Store -> Store
updateSto s v sto = (M s v) : sto

applySto:: String -> Store -> Value
applySto s (M t v:r)
   | s==t = v
   | otherwise = applySto s r
```

The Store…

```
Main> (updateSto "a" (IVal 5) emptySto)
[a->5]

Main> updateSto "a" (IVal 77) (updateSto "a" (IVal 5) emptySto)
[a->77,a->5]

Main> updateSto "b" (BVal True) (updateSto "a" (IVal 5) emptySto)
[b->True,a->5]
```


Readings and References

- Read pp. 223–226, 238–264, in *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz, http://www.cs.uiowa.edu/˜slonnegr/plf/Book.

Acknowledgments

- Much of the material in this lecture on Operational Semantics is taken from the book *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz, http://www.cs.uiowa.edu/˜slonnegr/plf/Book.