CSCI 740 - Programming Language Theory

Lecture 16

Semantics for Imperative Programs

Instructor: Hossein Hojjat

October 6, 2017

Big-Step Operational Semantics for λ-calculus

- Configuration is simply a λ-expression: (there is no state)
- Result is a different λ-expression
- Inductive definition: Base case

$$\frac{}{x \Downarrow x}$$

- Inductive definition: recursive cases

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}$$

Call-by-Name Reduction:

$$\frac{e_1 \Downarrow \lambda x.e_1' \qquad e_1'[x \mapsto \alpha(e_2)] \Downarrow e_3}{e_1\ e_2 \Downarrow e_3}$$

Call-by-Value Reduction:

$$\frac{e_1 \Downarrow \lambda x.e_1' \qquad e_2 \Downarrow e_2' \qquad e_1'[x \mapsto \alpha(e_2')] \Downarrow e_3}{e_1\ e_2 \Downarrow e_3}$$

Big-Step Operational Semantics for Imperative Programs

- The same techniques apply to programs with state
- The big difference is that the configuration now includes state

IMP: A Simple Imperative Language

$$e := \; n \mid x \mid e_1 == e_2 \mid \texttt{true} \mid \texttt{false}$$
$$c := \; x := e \mid c_1 \,;\, c_2 \mid \texttt{if}\, e\, \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{skip}$$

- Now we need two types of judgments

expressions result in values	commands change the state
$\langle e, \sigma \rangle \Downarrow n$	$\langle c, \sigma \rangle \Downarrow \sigma'$

- **State:** a function σ from variable names to values

Big-Step Operational Semantics for Expressions

- Rules for expressions are very similar to what we had before

$$\frac{}{\langle n, \sigma \rangle \Downarrow n} \qquad \frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2 \quad n = n_1 + n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow n}$$

- We need a rule to read values from variables

$$\frac{}{\langle x, \sigma \rangle \Downarrow \sigma(x)}$$

Big-Step Operational Semantics for Commands

- Commands mutate the state

$$\frac{\langle e, \sigma \rangle \Downarrow n}{\langle x := e, \sigma \rangle \Downarrow \sigma[x \mapsto n]} \qquad \frac{\langle c_1, \sigma \rangle \Downarrow \sigma'' \qquad \langle c_2, \sigma'' \rangle \Downarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma'}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow \texttt{false} \qquad \langle c_f, \sigma \rangle \Downarrow \sigma'}{\langle \texttt{if } e_1 \texttt{ then } c_t \texttt{ else } c_f, \sigma \rangle \Downarrow \sigma'} \qquad \frac{\langle e_1, \sigma \rangle \Downarrow \texttt{true} \qquad \langle c_t, \sigma \rangle \Downarrow \sigma'}{\langle \texttt{if } e_1 \texttt{ then } c_t \texttt{ else } c_f, \sigma \rangle \Downarrow \sigma'}$$

- What about loops?

Big-Step Operational Semantics for Loops

- The definition for loops must be recursive

$$\frac{\langle e_1, \sigma\rangle \Downarrow \texttt{false}}{\langle \texttt{while}\ e_1\ \texttt{do}\ \ c, \sigma\rangle \Downarrow \sigma}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \texttt{true} \qquad \langle c\ ;\ \texttt{while}\ e_1\ \texttt{do}\ \ c, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{while}\ e_1\ \texttt{do}\ \ c, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \texttt{true} \qquad \langle c, \sigma\rangle \Downarrow \sigma'' \qquad \langle \texttt{while}\ e_1\ \texttt{do}\ \ c, \sigma''\rangle \Downarrow \sigma'}{\langle \texttt{while}\ e_1\ \texttt{do}\ \ c, \sigma\rangle \Downarrow \sigma'}$$

Operational Semantics: Big-Step vs. Small-Step

Big-step Operational Semantics:

- Does not allow us to explicitly observe intermediate execution states
  $\boxed{\langle e, \sigma \rangle \Downarrow n}$ means that in state σ, expression e evaluates to n
- In one, big step, all the way to a result
  $\boxed{\langle c, \sigma \rangle \Downarrow \sigma'}$ after evaluating command c in state σ the new state will be σ'
- Hard to talk about commands that do not terminate
- But we do not have an explanation of how c runs or fails

Small-step Operational Semantics:

- Describes a single step in the evaluation
- Many steps may be needed to get a result
- Contextual semantics is a small-step semantics where the atomic execution step is a rewrite of the program

Contextual Semantics

- We define a relation $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$
- c' is obtained from c via an atomic rewrite step
- Evaluation terminates when the program has been rewritten to a terminal program
  - one from which we cannot make further progress
- For IMP the terminal command is `skip`
- As long as the command is not `skip` we can make further progress
- Some commands neverreduce to `skip`

```
while true do skip
```

Contextual Derivations

A contextual semantics derivation is a sequence (or list) of atomic rewrites:

$$\langle x + (8 - 2), \sigma \rangle \;\rightarrow\; \langle x + (6), \sigma \rangle \;\rightarrow\; \langle 4 + 6, \sigma \rangle \;\rightarrow\; \langle 10, \sigma \rangle$$

$$\sigma(x) = 4$$

Redexes

- A **redex** is a syntactic expression or command that can be reduced (transformed) in one atomic step
- The first step in defining a small-step semantics is to define the redexes

$$
\begin{aligned}
r \ ::= \ & x \\
\mid \ & n_1 + n_2 \\
\mid \ & n_1 == n_2 \\
\mid \ & x := n \\
\mid \ & \texttt{skip} \ ; \ c \\
\mid \ & \texttt{if true then } c_1 \texttt{ else } c_2 \\
\mid \ & \texttt{if false then } c_1 \texttt{ else } c_2 \\
\mid \ & \texttt{while } b \texttt{ do } c
\end{aligned}
$$

Note that $(1+2)+3$ is not a redex, but $1+2$ is

- In λ-calculus : $(\lambda x.v)\ e_2$, $(\lambda x.e_1)\ e_2$

Local Reduction Rules

- One for each redex: $\langle r, \sigma \rangle \to \langle e, \sigma' \rangle$
- Means in state σ, the redex r can be replaced in one step with the expression e

- $\langle x, \sigma \rangle \to \langle \sigma(x), \sigma \rangle$
- $\langle n_1 + n_2, \sigma \rangle \to \langle n, \sigma \rangle$ where $n = n_1$ plus n_2
- $\langle n_1 == n_2, \sigma \rangle \to \langle \texttt{true}, \sigma \rangle$ if $n_1 = n_2$
- $\langle x := n, \sigma \rangle \to \langle \texttt{skip}, \sigma[x := n] \rangle$
- $\langle \texttt{skip} \; ; \; c, \sigma \rangle \to \langle c, \sigma \rangle$
- $\langle \texttt{if true then } c_1 \texttt{ else } c_2, \sigma \rangle \to \langle c_1, \sigma \rangle$
- $\langle \texttt{if false then } c_1 \texttt{ else } c_2, \sigma \rangle \to \langle c_2, \sigma \rangle$
- $\langle \texttt{while } b \texttt{ do } c, \sigma \rangle \to$
  $\langle \texttt{if } b \texttt{ then } c; \texttt{ while } b \texttt{ do } c \texttt{ else skip}, \sigma \rangle$

The Global Reduction Rule

General idea of contextual semantics

- Decompose the current expression into the redex-to-reduce-next and the remaining program
  - The remaining program is called a context
- Reduce the redex r to some other expression e
- The resulting (reduced) expression consists of e with the original context

Example

Context

```
...
x := 2+2
...
```

- Step 1: Find The Redex

Example

Context

...

`x` `:=` $\boxed{\texttt{2+2}\text{(redex)}}$

...

- Step 1: Find The Redex
- Step 2: Reduce The Redex

Example

Context

...

`x := ` $\boxed{\texttt{2+2}\text{(redex)}}$ $\rightarrow$ `4` (reduced)

...

- Step 1: Find The Redex
- Step 2: Reduce The Redex

Example

Context

```
...
x := 4
...
```

- Step 1: Find The Redex
- Step 2: Reduce The Redex
- Step 3: Replace It In The Context

Contextual Analysis

- We use H to range over contexts
- We write $H[r]$ for the expression obtained by placing redex r in context H
- Now we can define a small-step

$$\text{If } \langle r, \sigma \rangle \to \langle e, \sigma' \rangle \text{ then } \langle H[r], \sigma \rangle \to \langle H[e], \sigma' \rangle$$

Contexts

- A context is like an expression (or command) with a hole $\circ$ in the place where the redex goes

Examples:

- To evaluate $(1+3)+2$ we use the redex $1+3$ and the context $\circ + 2$
- To evaluate $\texttt{if } x > 2 \texttt{ then } c_1 \texttt{ else } c_2$ we use
  the redex x and
  the context $\texttt{if } \circ > 2 \texttt{ then } c_1 \texttt{ else } c_2$

Example

Configuration	**Context**	**Redex**
$\langle x := (x+1)+2, [x=2]\rangle$	$x = (\circ + 1) + 2$	x
$\langle x := (2+1)+2, [x=2]\rangle$	$x = \circ + 2$	$2+1$
$\langle x := 3+2, [x=2]\rangle$	$x = \circ$	$3+2$
$\langle x := 5, [x=2]\rangle$	$\circ$	$x := 5$
$\langle \texttt{skip}, [x=5]\rangle$		

More On Contexts

- Contexts are defined by a grammar

$$
\begin{aligned}
H ::= \circ \mid n + H \mid H + e \mid x := H \\
\mid \texttt{if } H \texttt{ then } c_1 \texttt{ else } c_2 \mid H; c
\end{aligned}
$$

- The grammar defines the evaluation order
- Note in $a + b$, a is evaluated before b
- We can define redexes and contexts to
  - define the order of evaluation
  - define short circuit behavior

More On Contexts

- How do we know if our contexts and redexes are well-defined?

Decomposition theorem:

- If c is not `skip`, then there exist unique H and r such that c is $H[r]$
- Exist guarantees progress
- Unique guarantees determinism