AAA616: Program Analysis

Lecture 2 — Operational Semantics

Hakjoo Oh
2018 Spring

Plan

- Review: Inductive definition, inference rules, grammar
- Operational semantics of **While**
- Operational semantics of **Fun**
- Basic concepts of programming languages

cf) Imperative vs. Functional Languages

Statement and expressions:

- A statement *does something*.
- An expression *evaluates to a value*.

Programming languages can be classified into

- statement-oriented: C, C++, Java, Python, JavaScript, etc
  - often called "imperative languages"
- expression-oriented: ML, Haskell, Scala, Lisp, etc
  - often called "functional languages"

cf) Static vs. Dynamic Languages

Programming languages are classified into:

- *Statically typed languages*: type checking is done at compile-time.
  - type errors are detected before program executions
  - C, C++, Java, ML, Scala, etc
- *Dynamically typed languages*: type checking is done at run-time.
  - type errors are detected during program executions
  - Python, JavaScript, Ruby, Lisp, etc

Statically typed languages are further classified into:

- *Type-safe languages* guarantee that compiled programs do not have type errors at run-time.
  - All type errors are detected at compile time.
  - Compiled programs do not stuck.
  - ML, Haskell, Scala
- *Unsafe languages* do not provide such a guarantee.
  - Some type errors remain at run-time.
  - C, C++

Review: Inductive Definition

Inductive definition is widely used in the study of programming languages:

- Syntax
- Semantics

Induction is a technique for formally defining a set:

- The set is defined in terms of itself.
- The only way of defining an infinite set by a finite means.

Example

Definition (Top-Down)

A natural number n is in S if and only if

1. $n = 0$, or
2. $n - 3 \in S$.

Definition (Bottom-Up)

S is the *smallest* set such that $S \subseteq \mathbb{N}$ and S satisfies the following two conditions:

1. $0 \in S$, and
2. if $n \in S$, then $n + 3 \in S$.

Rules of Inference

$$\frac{A}{B}$$

- A: hypothesis (antecedent)
- B: conclusion (consequent)
- "if A is true then B is also true".
- $\overline{B}$: axiom.

Defining a Set by Rules of Inferences

Definition

$$\overline{\mathbf{0} \in S}$$

$$\frac{n \in S}{(n + \mathbf{3}) \in S}$$

Interpret the rules as follows:

"A natural number n is in S iff $n \in S$ can be derived from the axiom by applying the inference rules finitely many times"

ex) $\mathbf{3} \in S$ because

$$\frac{\overline{\mathbf{0} \in S}\ \text{the axiom}}{\mathbf{3} \in S}\ \text{the second rule}$$

Note that this interpretation enforces that S is the smallest set closed under the inference rules.

Natural Numbers

The set of natural numbers:

$$\mathbb{N} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots\}$$

is inductively defined:

$$\frac{}{\mathbf{0}} \qquad \frac{n}{n+\mathbf{1}}$$

The inference rules can be expressed by a grammar:

$$n \to \mathbf{0} \mid n + \mathbf{1}$$

Interpretation:

- 0 is a natural number.
- If n is a natural number then so is $n + \mathbf{1}$.

Strings

The set of strings over alphabet $\{\mathtt{a},\ldots,\mathtt{z}\}$, e.g., ϵ, a, b, . . . , z, aa, ab, . . . , az, ba, . . . az, aaa, . . . , zzz, and so on. Inference rules:

$$\overline{\epsilon} \qquad \frac{\alpha}{\mathtt{a}\alpha} \qquad \frac{\alpha}{\mathtt{b}\alpha} \quad \cdots \qquad \frac{\alpha}{\mathtt{z}\alpha}$$

or simply,

$$\overline{\epsilon} \qquad \frac{\alpha}{x\alpha}\ x \in \{\mathtt{a},\ldots,\mathtt{z}\}$$

In grammar:

$$\begin{array}{rcl} \alpha & \to & \epsilon \\ & \mid & x\alpha \quad (x \in \{\mathtt{a},\ldots,\mathtt{z}\}) \end{array}$$

Expressions

Expression examples: $\mathbf{2}$, $-\mathbf{2}$, $\mathbf{1}+\mathbf{2}$, $\mathbf{1}+(\mathbf{2}*(-\mathbf{3}))$, etc.
Inference rules:

$$\frac{}{n}\; n\in\mathbb{Z} \qquad \frac{e}{-e} \qquad \frac{e_1 \quad e_2}{e_1+e_2} \qquad \frac{e_1 \quad e_2}{e_1*e_2} \qquad \frac{e}{(e)}$$

In grammar:

$$\begin{array}{rcl} e & \rightarrow & n \quad (n\in\mathbb{Z}) \\ & | & -e \\ & | & e+e \\ & | & e*e \\ & | & (e) \end{array}$$

Example:

$$\dfrac{\dfrac{}{\mathbf{1}} \quad \dfrac{\dfrac{\dfrac{}{\mathbf{2}} \quad \dfrac{\dfrac{\dfrac{}{\mathbf{3}}}{-\mathbf{3}}}{(-\mathbf{3})}}{\mathbf{2}*(-\mathbf{3})}}{(\mathbf{2}*(-\mathbf{3}))}}{\mathbf{1}+(\mathbf{2}*(-\mathbf{3}))}$$

Syntax vs. Semantics

A programming language is defined with syntax and semantics.

- The syntax is concerned with the grammatical structure of programs.
  - Context-free grammar
- The semantics is concerned with the meaning of grammatically correct programs.
  - Operational semantics: The meaning is specified by the computation steps executed on a machine. It is of intrest how it is obtained.
  - Denotational semantics: The meaning is modelled by mathematical objects that represent the effect of executing the program. It is of interest the effect, not how it is obtained.

The **While** Language

n will range over numerals, **Num**
x will range over variables, **Var**
a will range over arithmetic expressions, **Aexp**
b will range over boolean expressions, **Bexp**
c, S will range over statements, **Stm**

$$
\begin{array}{rcl}
a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & \rightarrow & x := a \mid \texttt{skip} \mid c_1 ; c_2 \mid \texttt{if}\ b\ c_1\ c_2 \mid \texttt{while}\ b\ c
\end{array}
$$

Semantics of Arithmetic Expressions

- The meaning of an expression depends on the values bound to the variables that occur in the expression, e.g., $x + \mathbf{3}$.
- A state is a function from variables to values:

$$\textsf{\textbf{State}} = \mathbf{Var} \to \mathbb{Z}$$

- The meaning of arithmetic expressions is a function:

$$\mathcal{A} : \mathbf{Aexp} \to \textsf{\textbf{State}} \to \mathbb{Z}$$

$$\begin{array}{rcl}
\mathcal{A}[\![a]\!] & : & \textsf{\textbf{State}} \to \mathbb{Z} \\
\mathcal{A}[\![n]\!](s) & = & n \\
\mathcal{A}[\![x]\!](s) & = & s(x) \\
\mathcal{A}[\![a_1 + a_2]\!](s) & = & \mathcal{A}[\![a_1]\!](s) + \mathcal{A}[\![a_2]\!](s) \\
\mathcal{A}[\![a_1 \star a_2]\!](s) & = & \mathcal{A}[\![a_1]\!](s) \times \mathcal{A}[\![a_2]\!](s) \\
\mathcal{A}[\![a_1 - a_2]\!](s) & = & \mathcal{A}[\![a_1]\!](s) - \mathcal{A}[\![a_2]\!](s)
\end{array}$$

Semantics of Boolean Expressions

- The meaning of boolean expressions is a function:

$$\mathcal{B} : \text{Bexp} \to \textbf{State} \to \textbf{T}$$

where $\mathbf{T} = \{\mathit{true}, \mathit{false}\}$.

$$\begin{array}{rcl}
\mathcal{B}[\![b]\!] & : & \textbf{State} \to \textbf{T} \\
\mathcal{B}[\![\texttt{true}]\!](s) & = & \mathit{true} \\
\mathcal{B}[\![\texttt{false}]\!](s) & = & \mathit{false} \\
\mathcal{B}[\![a_1 = a_2]\!](s) & = & \mathcal{A}[\![a_1]\!](s) = \mathcal{A}[\![a_2]\!](s) \\
\mathcal{B}[\![a_1 \leq a_2]\!](s) & = & \mathcal{A}[\![a_1]\!](s) \leq \mathcal{A}[\![a_2]\!](s) \\
\mathcal{B}[\![\neg b]\!](s) & = & \mathcal{B}[\![b]\!](s) = \mathit{false} \\
\mathcal{B}[\![b_1 \wedge b_2]\!](s) & = & \mathcal{B}[\![b_1]\!](s) \wedge \mathcal{B}[\![b_2]\!](s)
\end{array}$$

Free Variables

The free variables of an arithmetic expression a are defined to be the set of variables occurring in it:

$$\begin{array}{rcl} FV(n) & = & \emptyset \\ FV(x) & = & \{x\} \\ FV(a_1 + a_2) & = & FV(a_1) \cup FV(a_2) \\ FV(a_1 \star a_2) & = & FV(a_1) \cup FV(a_2) \\ FV(a_1 - a_2) & = & FV(a_1) \cup FV(a_2) \end{array}$$

Exercise) Define free variables of boolean expressions.

Property of Free Variables

Only the free variables influence the value of an expression.

Lemma

Let s *and* s' *be two states satisfying that* $s(x) = s'(x)$ *for all* $x \in FV(a)$*. Then,* $\mathcal{A}[\![a]\!](s) = \mathcal{A}[\![a]\!](s')$*.*

Lemma

Let s *and* s' *be two states satisfying that* $s(x) = s'(x)$ *for all* $x \in FV(b)$*. Then,* $\mathcal{B}[\![b]\!](s) = \mathcal{B}[\![b]\!](s')$*.*

Substitution

- $a[y \mapsto a_0]$: the arithmetic expression that is obtained by replacing each occurrence of y in a by a_0.

$$
\begin{array}{rcl}
n[y \mapsto a_0] & = & n \\
x[y \mapsto a_0] & = & \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
(a_1 + a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\
(a_1 \star a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) \star (a_2[y \mapsto a_0]) \\
(a_1 - a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0])
\end{array}
$$

- $s[y \mapsto v]$: the state s except that the value bound to y is v.

$$
(s[y \mapsto v])(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}
$$

The two concepts of substitutions are related:

Lemma

$\mathcal{A}[\![a[y \mapsto a_0]]\!](s) = \mathcal{A}[\![a]\!](s[y \mapsto \mathcal{A}[\![a_0]\!](s)])$ *for all states* s.

Operational Semantics

Operational semantics is concerned about how to execute programs and not merely what the execution results are.

- *Big-step operational semantics* describes how the overall results of executions are obtained.
- *Small-step operational semantics* describes how the individual steps of the computations take place.

In both kinds, the semantics is specified by a transition system $(\mathbb{S}, \rightarrow)$ where $\mathbb{S}$ is the set of states (configurations) with two types:

- $\langle S, s \rangle$: a nonterminal state (i.e. the statement S is to be executed from the state s)
- s: a terminal state

The transition relation $(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}$ describes how the execution takes place. The difference between the two approaches are in the definitions of transition relation.

Big-step Operational Semantics

The transition relation specifies the relationship between the initial state and the final state:

$$\langle S, s\rangle \to s'$$

Transition relation is defined with inference rules of the form: A rule has the general form

$$\frac{\langle S_1, s_1\rangle \to s'_1, \ldots, \langle S_n, s_n\rangle \to s'_n}{\langle S, s\rangle \to s'} \text{ if } \cdots$$

- $S_1, \ldots, S_n$ are statements that constitute S.
- A rule has a number of premises and one conclusion.
- A rule may also have a number of conditions that have to be fulfilled whenever the rule is applied.
- Rules without premises are called axioms.

Big-step Operational Semantics for **While**

$$\frac{}{\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!](s)]}$$

$$\frac{}{\langle \texttt{skip}, s\rangle \to s}$$

$$\frac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

$$\frac{\langle S_1, s\rangle \to s'}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \to s'} \ \text{if } \mathcal{B}[\![b]\!](s) = \textbf{true}$$

$$\frac{\langle S_2, s\rangle \to s'}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \to s'} \ \text{if } \mathcal{B}[\![b]\!](s) = \textbf{false}$$

$$\frac{\langle S, s\rangle \to s' \qquad \langle \texttt{while}\ b\ S, s'\rangle \to s''}{\langle \texttt{while}\ b\ S, s\rangle \to s''} \ \text{if } \mathcal{B}[\![b]\!](s) = \textbf{true}$$

$$\frac{}{\langle \texttt{while}\ b\ S, s\rangle \to s} \ \text{if } \mathcal{B}[\![b]\!](s) = \textbf{false}$$

Example

Let s be a state with $s(x) = 3$. Then, we have

$$(\texttt{y:=1; while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}, s) \rightarrow s[y \mapsto 6][x \mapsto 1]$$

Execution Types

We say the execution of a statement S on a state s

- *terminates* if and only if there is a state s' such that $\langle S, s\rangle \to s'$ and
- *loops* if and only if there is no state s' such that $\langle S, s\rangle \to s'$.

We say a statement S always terminates if its execution on a state s terminates for all states s, and always loops if its execution on a state s loops for all states s.

Examples:

- `while true do skip`
- `while ¬(x=1) do (y:=y⋆x; x:=x-1)`

Semantic Equivalence

We say S_1 and S_2 are semantically equivalent, denoted $S_1 \equiv S_2$, if the following is true for all states s and s':

$$\langle S_1, s\rangle \to s' \quad \text{if and only if} \quad \langle S_2, s\rangle \to s'$$

Example:

$$\texttt{while}\ b\ \texttt{do}\ S \equiv \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}$$

Semantic Function for Statements

The semantic function for statements is the partial function:

$$\mathcal{S}_b : \mathrm{Stm} \to (\textbf{State} \hookrightarrow \textbf{State})$$

$$\mathcal{S}_b[\![S]\!](s) = \begin{cases} s' & \text{if } \langle S, s\rangle \to s' \\ \textbf{undef} & \text{otherwise} \end{cases}$$

Examples:

- $\mathcal{S}_b[\![\texttt{y:=1; while } \neg\texttt{(x=1) do (y:=y}\star\texttt{x; x:=x-1)}]\!](s[x \mapsto 3])$
- $\mathcal{S}_b[\![\texttt{while true do skip}]\!](\texttt{s})$

Summary of **While**

The syntax is defined by the grammar:

$$
\begin{array}{rcl}
a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & \rightarrow & x := a \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if}\ b\ c_1\ c_2 \mid \texttt{while}\ b\ c
\end{array}
$$

The semantics is defined by the functions:

$$
\begin{array}{rcl}
\mathcal{A}[\![a]\!] & : & \textbf{State} \rightarrow \mathbb{Z} \\
\mathcal{B}[\![b]\!] & : & \textbf{State} \rightarrow \textbf{T} \\
\mathcal{S}_b[\![c]\!] & : & \textbf{State} \hookrightarrow \textbf{State}
\end{array}
$$

Small-step Operational Semantics

The individual computation steps are described by the transition relation of the form:

$$\langle S, s \rangle \Rightarrow \gamma$$

where γ either is non-terminal state $\langle S', s' \rangle$ or terminal state s'. The transition expresses the first step of the execution of S from state s.

- If $\gamma = \langle S', s' \rangle$, then the execution of S from s is not completed and the remaining computation continues with $\langle S', s' \rangle$.
- If $\gamma = s'$, then the execution of S from s has terminated and the final state is s'.

We say $\langle S, s \rangle$ is stuck if there is no γ such that $\langle S, s \rangle \Rightarrow \gamma$ (no stuck state for **While**).

Small-step Operational Semantics for **While**

$$\frac{}{\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!](s)]}$$

$$\frac{}{\langle \texttt{skip}, s\rangle \Rightarrow s}$$

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

$$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$\frac{}{\langle \texttt{if}\ b\ S_1\ \ S_2, s\rangle \Rightarrow \langle S_1, s\rangle}\ \text{if } \mathcal{B}[\![b]\!](s) = \textbf{true}$$

$$\frac{}{\langle \texttt{if}\ b\ S_1\ \ S_2, s\rangle \Rightarrow \langle S_2, s\rangle}\ \text{if } \mathcal{B}[\![b]\!](s) = \textbf{false}$$

$$\frac{}{\langle \texttt{while}\ b\ S, s\rangle \Rightarrow \langle \texttt{if}\ \ b\ \ \texttt{(}S\texttt{;}\ \ \texttt{while}\ \ b\ \ S\texttt{)}\ \ \texttt{skip}, s\rangle}$$

Derivation Sequence

A *derivation sequence* of a statement S starting in state s is either

- A finite sequence

  $$\gamma_0, \gamma_1, \gamma_2, \cdots, \gamma_k$$

  which is sometimes written

  $$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_k$$

  such that

  $$\gamma_0 = \langle S, s \rangle, \quad \gamma_i \Rightarrow \gamma_{i+1} \text{ for } 0 \leq i \leq k$$

  and γ_k is either a terminal configuration or a stuck configuration.
- An infinite sequence

  $$\gamma_0, \gamma_1, \gamma_2, \cdots$$

  which is sometimes written

  $$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots$$

  consisting of configurations satisfying $\gamma_0 = \langle S, s \rangle$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for $0 \leq i$.

Example

Let s be a state such that $s(x) = 5, s(y) = 7, s(z) = 0$. Consider the statement:

$$(\mathrm{z} := \mathrm{x}; \mathrm{x} := \mathrm{y}); \mathrm{y} := \mathrm{z}$$

Compute the derivation sequence starting in s.

Example: Factorial

Assume that $s(x) = 3$.

$\langle$`y:=1; while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s\rangle$
$\Rightarrow \langle$`while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s[y \mapsto 1]\rangle$
$\Rightarrow \langle$`if ¬(x=1) then ((y:=y⋆x; x:=x-1);while ¬(x=1) do (y:=y⋆x; x:=x-1))`
`else skip`$, s[y \mapsto 1]\rangle$
$\Rightarrow \langle$`(y:=y⋆x; x:=x-1);while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s[y \mapsto 1]\rangle$
$\Rightarrow \langle$`x:=x-1;while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s[y \mapsto 3]\rangle$
$\Rightarrow \langle$`while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s[y \mapsto 3][x \mapsto 2]\rangle$
$\Rightarrow \langle$`if ¬(x=1) then ((y:=y⋆x; x:=x-1);while ¬(x=1) do (y:=y⋆x; x:=x-1))`
`else skip`$, s[y \mapsto 3][x \mapsto 2]\rangle$
$\Rightarrow \langle$`(y:=y⋆x; x:=x-1);while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s[y \mapsto 3][x \mapsto 2]\rangle$
$\Rightarrow \langle$`x:=x-1;while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s[y \mapsto 6][x \mapsto 2]\rangle$
$\Rightarrow \langle$`while ¬(x=1) do (y:=y⋆x; x:=x-1)`$, s[y \mapsto 6][x \mapsto 1]\rangle$
$\Rightarrow s[y \mapsto 6][x \mapsto 1]$

Other Notations

- We write $\gamma_0 \Rightarrow^i \gamma_i$ to indicate that there are i steps in the execution from γ_0 to γ_i.
- We write $\gamma_0 \Rightarrow^* \gamma_i$ to indicate that there are a finite number of steps.
- We say that the execution of a statement S on a state s terminates if and only if there is a finite derivation sequence starting with $\langle S, s \rangle$.
- The execution loops if and only if there is an infinite derivation sequence starting with $\langle S, s \rangle$.

Semantic Function

The semantic function $\mathcal{S}_s$ for small-step semantics:

$$\mathcal{S}_s : \mathrm{Stm} \to (\textbf{State} \hookrightarrow \textbf{State})$$

$$\mathcal{S}_s[\![S]\!](s) = \begin{cases} s' & \text{if } \langle S, s \rangle \Rightarrow^* s' \\ \textbf{undef} & \end{cases}$$

Summary of **While**

We have defined the operational semantics of **While**.

- *Big-step operational semantics* describes how the overall results of executions are obtained.
- *Small-step operational semantics* describes how the individual steps of the computations take place.

The big-step and small-step operational semantics are equivalent:

Theorem

For every statement S of **While**, *we have* $\mathcal{S}_b[\![S]\!] = \mathcal{S}_s[\![S]\!]$.

Scope and Procedures

Consider the simple expression-oriented language:

$$
\begin{array}{rcl}
P & \rightarrow & E \\
E & \rightarrow & n \\
& | & x \\
& | & E + E \\
& | & E - E \\
& | & \texttt{zero?}\ E \\
& | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ E \\
& | & \texttt{read}
\end{array}
$$

Examples

- `1, 2, x, y`
- `1+(2+3), x+1, x+(y-2)`
- `zero? 1, zero? (2-2), zero? (zero? 3)`
- `if iszero 1 then 2 else 3, if 1 then 2 else 3`
- ```
  let x = read
  in  x + 1
  ```
- ```
  let x = read
  in let y = 2
     in if iszero x then y else x
  ```

Values and Environments

The set of values includes integers and booleans:

$$v \in \boldsymbol{Val} = \mathbb{Z} + \boldsymbol{Bool}$$

and an environment is a function from variables to values:

$$\rho \in \boldsymbol{Env} = \boldsymbol{Var} \to \boldsymbol{Val}$$

Notations:

- $[]$: the empty environment.
- $[x \mapsto v]\rho$ (or $\rho[x \mapsto v]$): the extension of ρ where x is bound to v:

$$([x \mapsto v]\rho)(y) = \begin{cases} v & \text{if } x = y \\ \rho(y) & \text{otherwise} \end{cases}$$

For simplicity, we write $[x_1 \mapsto v_1, x_2 \mapsto v_2]\rho$ for the extension of ρ where x_1 is bound to v_1, x_2 to v_2:

$$[x_1 \mapsto v_1, x_2 \mapsto v_2]\rho = [x_1 \mapsto v_1]([x_2 \mapsto v_2]\rho)$$

Evaluation Rules

$$\boxed{\rho \vdash e \Rightarrow v}$$

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2} \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 - E_2 \Rightarrow n_1 - n_2}$$

$$\frac{}{\rho \vdash \texttt{read} \Rightarrow n} \qquad \frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{zero?}\ E \Rightarrow \textsf{true}} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{zero?}\ E \Rightarrow \textsf{false}}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \textsf{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ E_2\ E_3 \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow \textsf{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ E_2\ E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

cf) Precise Interpretation

- The inference rules define a set S of triples (ρ, e, v). For readability, the triple was written by $\rho \vdash e \Rightarrow v$ in the rules.
- We say an expression e has semantics w.r.t. ρ iff there is a triple $(\rho, e, v) \in S$ for some value v.
- That is, we say an expression e has semantics w.r.t. ρ iff we can derive $\rho \vdash e \Rightarrow v$ for some value v by applying the inference rules.
- We say an initial program e has semantics if $[] \vdash e \Rightarrow v$ for some v.

Procedures

$$
\begin{array}{rcl}
P & \rightarrow & E \\
E & \rightarrow & n \\
& | & x \\
& | & E + E \\
& | & E - E \\
& | & \texttt{zero?}\ E \\
& | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ E \\
& | & \texttt{read} \\
& | & \texttt{proc}\ x\ E \\
& | & E\ E
\end{array}
$$

Example

- ```
  let f = proc (x) (x-11)
  in (f (f 77))
  ```
- `((proc (f) (f (f 77))) (proc (x) (x-11)))`
  ```

Free/Bound Variables of Procedures

- An occurrence of the variable `x` is *bound* when it occurs without definitions in the body of a procedure whose formal parameter is `x`.
- Otherwise, the variable is *free*.
- Examples:
  - `proc (y) (x+y)`
  - `proc (x) (let y = 1 in x + y + z)`
  - `proc (x) (proc (y) (x+y))`
  - `let x = 1 in proc (y) (x+y)`
  - `let x = 1 in proc (y) (x+y+z)`

Static vs. Dynamic Scoping

What is the result of the program?

```
let x = 1
in let f = proc (y) (x+y)
   in let x = 2
      in let g = proc (y) (x+y)
         in (f 1) + (g 1)
```

Two ways to determine free variables of procedures:

- In *static scoping* (*lexical scoping*), the procedure body is evaluated in the environment where the procedure is defined (i.e. procedure-creation environment).
- In *dynamic scoping*, the procedure body is evaluated in the environment where the procedure is called (i.e. calling environment)

Why Static Scoping?

Most modern languages use static scoping. Why?

- Reasoning about programs is much simpler in static scoping.
- In static scoping, renaming bound variables by their lexical definitions does not change the semantics, which is unsafe in dynamic scoping.

```
let x = 1
in let f = proc (y) (x+y)
   in let x = 2
      in let g = proc (y) (x+y)
         in (f 1) + (g 1)
```

- In static scoping, names are resolved at compile-time.
- In dynamic scoping, names are resolved only at runtime.

Semantics of Procedures: Static Scoping

- Domain:

$$\begin{array}{rcl} Val & = & \mathbb{Z} + Bool + Procedure \\ Procedure & = & Var \times E \times Env \\ Env & = & Var \rightarrow Val \end{array}$$

  The procedure value is called *closures*. The procedure is closed in its creation environment.

- Semantics rules:

$$\frac{}{\rho \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \qquad [x \mapsto v]\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

Examples

$$\frac{}{[] \vdash \begin{array}{l}\texttt{let x = 1}\\\texttt{in let f = proc (y) (x+y)}\\\quad\texttt{in let x = 2}\\\qquad\texttt{in (f 3)}\end{array} \Rightarrow 4}$$

Semantics of Procedures: Dynamic Scoping

- Domain:

$$\begin{array}{rcl} \boldsymbol{Val} & = & \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\ \boldsymbol{Procedure} & = & \boldsymbol{Var} \times \boldsymbol{E} \\ \boldsymbol{Env} & = & \boldsymbol{Var} \rightarrow \boldsymbol{Val} \end{array}$$

- Semantics rules:

$$\frac{}{\rho \vdash \texttt{proc}\ x\ E \Rightarrow (x, E)}$$

$$\frac{\rho \vdash E_1 \vdash (x, E) \qquad \rho \vdash E_2 \Rightarrow v \qquad [x \mapsto v]\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

Adding Recursive Procedures

The current language does not support recursive procedures, e.g.,

```
let f = proc (x) (f x)
in (f 1)
```

for which evaluation gets stuck:

$$\frac{[f \mapsto (x, f\ x, [])] \vdash f \Rightarrow (x, f\ x, []) \qquad \dfrac{[x \mapsto 1] \vdash f \Rightarrow ? \qquad [x \mapsto 1] \vdash x \Rightarrow 1}{[x \mapsto 1] \vdash f\ x \Rightarrow ?}}{[f \mapsto (x, f\ x, [])] \vdash (f\ 1) \Rightarrow ?}$$

Two solutions:

- go back to dynamic scoping :-(
- modify the language syntax and semantics for procedure :-)

Recursion is Not Special in Dynamic Scoping

With dynamic scoping, recursive procedures require no special mechanism. Running the program

```
let f = proc (x) (f x)
in (f 1)
```

via dynamic scoping semantics

$$\frac{\rho \vdash E_1 \Rightarrow (x, E) \qquad \rho \vdash E_2 \Rightarrow v \qquad [x \mapsto v]\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

proceeds well:

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{[f \mapsto (x, f\ x), x \mapsto 1] \vdash \texttt{f x} \Rightarrow}}{[f \mapsto (x, f\ x), x \mapsto 1] \vdash \texttt{f x} \Rightarrow}}{[f \mapsto (x, f\ x)] \vdash \texttt{f 1} \Rightarrow}}{[] \vdash \texttt{let f = proc (x) (f x) in (f 1)} \Rightarrow}$$

Adding Recursive Procedures

$$
\begin{array}{rcl}
P & \to & E \\
E & \to & n \\
& | & x \\
& | & E + E \\
& | & E - E \\
& | & \texttt{zero?}\ E \\
& | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ E \\
& | & \texttt{read} \\
& | & \texttt{letrec}\ f(x) = E\ \texttt{in}\ E \\
& | & \texttt{proc}\ x\ E \\
& | & E\ E
\end{array}
$$

Example

```
letrec double(x) =
  if zero?(x) then 0 else ((double (x-1)) + 2)
in (double 1)
```

Semantics of Recursive Procedures

- Domain:

$$\begin{array}{rcl} \mathit{Val} & = & \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} \\ \mathit{Procedure} & = & \mathit{Var} \times E \times \mathit{Env} \\ \mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times E \times \mathit{Env} \\ \mathit{Env} & = & \mathit{Var} \to \mathit{Val} \end{array}$$

- Semantics rules:

$$\frac{[f \mapsto (f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \\ [x \mapsto v, f \mapsto (f, x, E, \rho')]\rho' \vdash E \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

States

- So far, our language only had the values produced by computation.
- But computation also has *effects*: it may change the state of memory.
- We will extend the language to support computational effects:
  - Syntax for creating and using memory locations
  - Semantics for manipulating memory states

Motivating Example

- How can we compute the number of times `f` has been called?

```
let f = proc (x) (x)
in (f (f 1))
```

- Does the following program work?

```
let counter = 0
in let f = proc (x) (let counter = counter + 1
                     in x)
   in let a = (f (f 1))
      in counter
```

- The binding of `counter` is local. We need global *effects*.
- Effects are implemented by introducing *memory* (*store*) and *locations* (*reference*).

Two Approaches

Programming languages support references explicitly or implicitly.

- Languages with explicit references provide a clear account of allocation, dereference, and mutation of memory cells.
  - e.g., OCaml, F#
- In languages with implicit references, references are built-in. References are not explicitly manipulated.
  - e.g., C and Java.

A Language with Explicit References

$$
\begin{array}{rcl}
P & \to & E \\
E & \to & n \mid x \\
& \mid & E + E \mid E - E \\
& \mid & \texttt{zero?}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\
& \mid & \texttt{proc}\ x\ E \mid E\ E \\
& \mid & \texttt{ref}\ E \\
& \mid & !\ E \\
& \mid & E := E \\
& \mid & E; E
\end{array}
$$

- `ref` E allocates a new location, store the value of E in it, and returns it.
- $!\ E$ returns the contents of the location that E refers to.
- $E_1 := E_2$ changes the contents of the location (E_1) by the value of E_2.
- $E_1; E_2$ executes E_1 and then E_2 while accumulating effects.

Example 1

- ```
  let counter = ref 0
  in let f = proc (x) (counter := !counter + 1; !counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a - b)
  ```
- ```
  let f = let counter = ref 0
          in proc (x) (counter := !counter + 1; !counter)
  in let a = (f 0)
     in let b = (f 0)
        in (a - b)
  ```
- ```
  let f = proc (x) (let counter = ref 0
                    in (counter := !counter + 1; !counter))
  in let a = (f 0)
     in let b = (f 0)
        in (a - b)
  ```
  ```

Example 2

We can make chains of references:

```
let x = ref (ref 0)
in (!x := 11; !(!x))
```

Semantics

Memory is modeled as a finite map from locations to values:

$$
\begin{array}{rcl}
Val & = & \mathbb{Z} + Bool + Procedure + Loc \\
Procedure & = & Var \times E \times Env \\
\rho \in Env & = & Var \to Val \\
\sigma \in \mathbb{M} & = & Loc \to Val
\end{array}
$$

Semantics rules additionally describe memory effects:

$$\rho, \sigma \vdash E \Rightarrow v, \sigma'$$

Semantics

Existing rules are enriched with memory effects:

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{zero?}\ E \Rightarrow \textsf{true}, \sigma_1} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow n, \sigma_1}{\rho, \sigma_0 \vdash \texttt{zero?}\ E \Rightarrow \textsf{false}, \sigma_1}\ n \neq 0$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \textsf{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ E_2\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \textsf{false}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_3 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ E_2\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto v_1]\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad [x \mapsto v]\rho', \sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}$$

Semantics

Rules for new constructs:

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{ref } E \Rightarrow l, [l \mapsto v]\sigma_1} \quad l \notin \mathsf{dom}(\sigma_1)$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma_0 \vdash \,!\, E \Rightarrow \sigma_1(l), \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash E_1 := E_2 \Rightarrow v, [l \mapsto v]\sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

A Language with Implicit References

$$
\begin{array}{rcl}
P & \rightarrow & E \\
E & \rightarrow & n \mid x \\
 & \mid & E + E \mid E - E \\
 & \mid & \texttt{zero?}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\
 & \mid & \texttt{proc}\ x\ E \mid E\ E \\
 & \mid & \texttt{set}\ x = E \\
 & \mid & E; E
\end{array}
$$

- In this design, every variable denotes a reference and is mutable.
- set $x = E$ changes the contents of x by the value of E.

Examples

Computing the number of times `f` has been called:

- ```
  let counter = 0
  in let f = proc (x) (set counter = counter + 1; counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a-b)
  ```
- ```
  let f = let counter = 0
          in proc (x) (set counter = counter + 1; counter)
  in let a = (f 0)
     in let b = (f 0)
        in (a-b)
  ```
- ```
  let f = proc (x) (let counter = 0
                    in (set counter = counter + 1; counter))
  in let a = (f 0)
     in let b = (f 0)
        in (a-b)
  ```
  ```

Exercise

What is the result of the program?

```
let f = proc (x)
          proc (y)
           (set x = x + 1; x - y)
in ((f 44) 33)
```

Semantics

References are no longer values and every variable denotes a reference:

$$\begin{array}{rcl} \boldsymbol{Val} & = & \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\ \boldsymbol{Procedure} & = & \boldsymbol{Var} \times \boldsymbol{E} \times \boldsymbol{Env} \\ \boldsymbol{\rho} \in \boldsymbol{Env} & = & \boldsymbol{Var} \to \boldsymbol{Loc} \\ \boldsymbol{\sigma} \in \mathbb{M} & = & \boldsymbol{Loc} \to \boldsymbol{Val} \end{array}$$

Semantics

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{zero?}\ E \Rightarrow \textsf{true}, \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \textsf{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ E_2\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho), \sigma} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{set}\ x = E \Rightarrow v, [\rho(x) \mapsto v]\sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto l]\rho, [l \mapsto v_1]\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \notin \textsf{dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \textsf{dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

Summary

- Big-step semantics of **While**
- Small-step semantics of **While**
- Big-step semantics of **Fun**

Homework 1

Define the semantics of the language that combines **While** and **Fun**:

$$
\begin{array}{rcl}
E & \rightarrow & \mathtt{skip} \\
& | & n \mid x \mid \mathtt{true} \mid \mathtt{false} \mid E_1 + E_2 \mid E_1 < E_2 \\
& | & x := E \\
& | & \mathtt{if}\ E_1\ E_2\ E_3 \\
& | & \mathtt{while}\ E_1\ E_2 \\
& | & \mathtt{for}\ x := E_1\ \mathtt{to}\ E_2\ \mathtt{do}\ E_3 \\
& | & \mathtt{let}\ x := E_1\ \mathtt{in}\ E_2 \\
& | & \mathtt{let\ proc}\ f(x) = E_1\ \mathtt{in}\ E_2 \\
& | & f(E) \\
& | & E_1; E_2
\end{array}
$$

Use LaTeX and submit the document via email to TA (Due 3/25 24:00).