Operational Semantics

Mooly Sagiv

Winskel: The Formal Semantics of Programming Languages
Chapter 2

Recap Lambda Calculus

- An abstraction for functional programming
- Other abstractions
  - Pi-calculus [Milner'93] abstracts concurrent computations
    - Cryptographic protocols
    - Molecular biology
    - Business process

Modern Programming Languages

- Imperative
  - Pascal
  - C
- Object Oriented
  - C++, Java, C#
  - Smalltalk
  - Javascript, Scala, Python
- Functional
  - Lisp/Scheme
  - ML/Haskell
- Logic
  - Prolog

Programming Languages

- Syntax
  - Which string is a legal program?
  - Usually defined using context free grammar+ contextual constraints
- Semantics
  - What does a program mean?
  - What is the output of the program on a given run?
  - When does a runtime error occur?
  - A formal definition

Benefits of Formal Semantics

- Programming language design
  - hard- to-define= hard-to-implement=hard-to-use
  - Avoid design mistakes
- Programming language implementation
  - Compiler Correctness
    - Correctness of program optimizations
    - Design of Static Analysis
- Programming language understanding
- Program correctness
- Program equivalence
- Automatic generation of interpreter
- Techniques used in software engineering

Desired Features of PL Semantics

- Tractable
  - as simple as possible without losing the ability to express behavior accurately
- Abstract
  - uncluttered by irrelevant detail
- Computational
  - an accurate abstraction from runtime behavior
- Compositional
  - The meaning of compound language construct is defined using the meaning of subconstructs
  - Supports modular reasoning

Alternative Formal Semantics

- Operational Semantics [Plotkin]
  - The meaning of the program is described "operationally"
  - Structural Operational Semantics
- Denotational Semantics [Strachey, Scott]
  - The meaning of the program is an input/output relation
- Axiomatic Semantics [Floyd, Hoare]
  - The meaning of the program is observed properties
  - Proof rules to show that the program is correct
- Complement each other

Tentative Plan

- A simple programming language IMP
  - Structural operational Semantics of IMP
  - Inductive Definitions
  - Denotational Semantics of IMP
  - Axiomatic Semantics of IMP
  - Non-Determinism and Parallelism
- Advanced programming languages
  - Java byte code
  - Memory Models (Lahav)

Chapter 2

Introduction to Operational Semantics

IMP: A Simple Imperative Language

- numbers N
  - Positive and negative numbers
  - $n, m \in N$
- truth values T={true, false}
- locations Loc
  - $X, Y \in Loc$
- arithmetic Aexp
  - $a \in Aexp$
- boolean expressions Bexp
  - $b \in Bexp$
- commands Com
  - $c \in Com$

Abstract Syntax for Axp

- Aexp
  - $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$
- Haskell

```
data Aexp = Number(Int) | Var(String) |
                          Plus(Aexp, Aexp) |
                           Minus(Aexp, Aexp)|
                           Times(Aexp, Aexp)
```

Abstract Syntax for Bexp

- Bexp
  - b ::= true | false | $a_0 = a_1$ | $a_0 \leq a_1$ | $\neg b$ | $b_0 \wedge b_1$ | $b_0 \vee b_1$
- Haskell
  - data Bexp = TT | FF | Eq(Aexp, Aexp) | Leq(Aexp, Aexp)| Not(Bexp)| Land(Bexp, Bexp)| Lor(Bexp, Bexp)

Abstract Syntax for Com

- Com
  - c ::= skip | X := a | c_0 ; c_1 | if b then c_0 else c_1
    | while b do c
  - data Comm = Skip | Ass(String, Bexp) |
    If(Bexp, Comm, Comm) |
    Loop(Bexp, Aexp)

Abstract Syntax for IMP

- Aexp
  - $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$
- Bexp
  - $b ::= \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$
- Com
  - $c ::= \text{skip} \mid X := a \mid c_0 ; c_1 \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \mid \text{while } b \text{ do } c$

$2+3\times4-5$

$(3+5) \equiv 3 + 5$

$(2+(3\times4))-5$ $\quad$ $((2+3)\times4))-5$ $\quad$ $3 + 5 \not\equiv 5+ 3$

Example Program

```
Y := 1;
while ¬(X=1) do
        Y := Y * X;
        X := X - 1
```

Another Example

```
while ¬(X=Y) do
        if X > Y then
            X := X – Y
        else
            Y := Y - X
```

But what about semantics

Expression Evaluation

- States
  - Mapping locations to values
  - Σ - The set of states
    - $\sigma : Loc \rightarrow N$
    - $\sigma(X) = \sigma X$ = value of X in σ
    - $\sigma = [X \mapsto 5, Y \mapsto 7]$
      - The value of X is 5
      - The value of Y is 7
      - The value of Z is undefined
  - For $a \in Aexp$, $\sigma \in \Sigma$, $n \in N$,
    - $\langle a, \sigma \rangle \rightarrow n$
      - a is evaluated in σ to n

Evaluating $(a_0 + a_1)$ at σ

- Evaluate a_0 to get a number n_0 at σ
- Evaluate a_1 to get a number n_1 at σ
- Add n_0 and n_1

Expression Evaluation Rules

Axioms

- Numbers
  - $\langle n, \sigma \rangle \rightarrow n$
- Locations
  - $\langle X, \sigma \rangle \rightarrow \sigma(X)$
- Sums $\dfrac{\langle a_0, \sigma \rangle \rightarrow n_0, \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n}$ *where* $n = n_0 + n_1$
- Subtractions $\dfrac{\langle a_0, \sigma \rangle \rightarrow n_0, \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 - a_1, \sigma \rangle \rightarrow n}$ *where* $n = n_0 - n_1$
- Products $\dfrac{\langle a_0, \sigma \rangle \rightarrow n_0, \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 \times a_1, \sigma \rangle \rightarrow n}$ *where* $n = n_0 \times n_1$

Derivations

- A rule instance
  - Instantiating meta variables with corresponding values

$$\frac{< 2, \sigma_0 > \rightarrow 2, < 3, \sigma_0 > \rightarrow 3}{< 2 \times 3, \sigma > \rightarrow 6}$$

$$\frac{< 2, \sigma_0 > \rightarrow 3, < 3, \sigma_0 > \rightarrow 4}{< 2 \times 3, \sigma > \rightarrow 12}$$

Derivation (Tree)

- Axioms in the leafs
- Rule instances at internal nodes

$$\frac{}{<\text{Init},\sigma_0>\rightarrow 0} \qquad \frac{}{<5,\sigma_0>\rightarrow 5} \qquad \frac{}{<7,\sigma_0>\rightarrow 7} \qquad \frac{}{<9,\sigma_0>\rightarrow 9}$$

$$\frac{}{<(\text{Init}+5),\sigma_0>\rightarrow 5} \qquad \frac{}{<7+9,\sigma_0>\rightarrow 16}$$

$$\frac{}{<(\text{Init}+5)+(7+9),\sigma_0>\rightarrow 21}$$

Computing a derivation

- We write <a, σ> → n when there exists a derivation tree whose root is <a, σ> → Can be computed in a top-down manner
- At every node try all derivations "in parallel"

$$\frac{}{<\text{Init},\sigma_0>\rightarrow 0} \qquad \frac{}{<5,\sigma_0>\rightarrow 5} \qquad \frac{}{<7,\sigma_0>\rightarrow 7} \qquad \frac{}{<9,\sigma_0>\rightarrow 9}$$

$$\frac{}{<(\text{Init}+5),\sigma_0>\rightarrow 5} \qquad \frac{}{<7+9,\sigma_0>\rightarrow 16}$$

$$\frac{}{<(\text{Init}+5)+(7+9),\sigma_0>\rightarrow 21}$$

Haskell Code

```
retrieve env id = case env of
                      [] -> 0
                      (id1, v) : t -> if id1 = id then v
                                                  else retrieve t id
eval  exp env =
   case exp of    Number(n) -> n
                  Variable(v) -> retrieve env v
                   Plus(e1, e2) -> let n1 = eval e1 env in
                                              let n2 = eval e2 env in
                                                 n1 + n2
    …
```

Recap

- Operational Semantics
  - The rules can be implemented easily
  - Define interpreter
- Structural Operational Semantics
  - Syntax directed
- Natural semantics

Equivalence of IMP expressions

$a_0 \sim a_1$ iff

$$\forall n \in N \forall \sigma \in \Sigma. < a_0, \sigma > \rightarrow n \Leftrightarrow < a_1, \sigma > \rightarrow n$$

Boolean Expression Evaluation Rules

- $\langle \mathbf{true}, \sigma \rangle \rightarrow \mathbf{true}$
- $\langle \mathbf{false}, \sigma \rangle \rightarrow \mathbf{false}$

- $$\frac{\langle a_0, \sigma \rangle \rightarrow n, \langle a_1, \sigma \rangle \rightarrow m}{\langle a_0 = a_1, \sigma \rangle \rightarrow \mathbf{true}} \; if \quad n = m$$
- $$\frac{\langle a_0, \sigma \rangle \rightarrow n, \langle a_1, \sigma \rangle \rightarrow m}{\langle a_0 = a_1, \sigma \rangle \rightarrow \mathbf{false}} \; if \quad n \neq m$$
- $$\frac{\langle a_0, \sigma \rangle \rightarrow n, \langle a_1, \sigma \rangle \rightarrow m}{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \text{true}} \; if \quad n \leq m$$
- $$\frac{\langle a_0, \sigma \rangle \rightarrow n, \langle a_1, \sigma \rangle \rightarrow m}{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \mathbf{true}} \; if \quad not \quad n \leq m$$

Boolean Expression Evaluation Rules(cont)

- $$\frac{< b, \sigma > \rightarrow \textbf{true}}{< \neg b, \sigma > \rightarrow \textbf{false}}$$

- $$\frac{< b, \sigma > \rightarrow \textbf{false}}{< \neg b, \sigma > \rightarrow \textbf{true}}$$

- $$\frac{< b_0, \sigma > \rightarrow t_0, < b_1, \sigma > \rightarrow t_1}{< b_0 \wedge b_1, \sigma > \rightarrow t}$$ where t = **true** when $t_0 = t_1 =$ **true** and t = **false** otherwise

- $$\frac{< b_0, \sigma > \rightarrow t_0, < b_1, \sigma > \rightarrow t_1}{< b_0 \vee b_1, \sigma > \rightarrow t}$$ where t = false when $t_0 = t_1 =$ false and t = true otherwise

Equivalence of Boolean expressions

$b_0 \sim b_1$ iff

$$\forall t \in T \forall \sigma \in \Sigma . < b_0, \sigma > \rightarrow t \Leftrightarrow < b_1, \sigma > \rightarrow t$$

Extensions

- Shortcut evaluation of Boolean expressions
- “Parallell” evaluation of Boolean expressions
- Other data types

Haskell Code

```
beval  exp env =
    case exp of     tt -> true
                    ff -> false
                    Eq(e1, e2) ->  eval(e1, e) = eval(e2, e)
                    Leq(e1, e2) -> eval(e1, e) = eval(e2, e)
                    Land(e1, e2)-> let b1 = beval e1 env in
                                          let b2 = eval e2 env in
                                            b1 and b2
        …
```

The execution of commands

- $\langle c, \sigma \rangle \rightarrow \sigma'$
  - c terminates on σ in a final state σ'
- Initial state σ_0
  - $\sigma_0(X)=0$ for all X
- Handling assignments $\langle X:=5, \sigma \rangle \rightarrow \sigma'$
-

$$\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X \\ \sigma(\mathrm{Y}) & \text{if } Y \neq X \end{cases}$$

- $\langle X:=5, \sigma \rangle \rightarrow \sigma[5/X]$

Rules for commands

Atomic

- $\langle \textbf{skip}, \sigma \rangle \rightarrow \sigma$
- $$\frac{< a, \sigma > \rightarrow n}{< X := a, \sigma > \rightarrow \sigma[n/X]}$$
- Sequencing: $$\frac{< c0, \sigma > \rightarrow \sigma'' \quad < c1, \sigma'' > \rightarrow \sigma'}{< c_0 \,;\, c_1, \sigma > \rightarrow \sigma'}$$
- Conditionals: $$\frac{< b, \sigma > \rightarrow \textbf{true} \quad < c_0, \sigma > \rightarrow \sigma'}{< \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma > \rightarrow \sigma'}$$

$$\frac{< b, \sigma > \rightarrow \textbf{false} \quad < c_1, \sigma > \rightarrow \sigma'}{< \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma > \rightarrow \sigma'}$$

Rules for commands (while)

$$\frac{< b, \sigma > \rightarrow \textbf{false}}{< \textbf{while}\ \ b\ \textbf{do}\ c\ , \sigma > \rightarrow \sigma}$$

$$\frac{< b, \sigma > \rightarrow \textbf{true} \quad < \textbf{c}, \sigma > \rightarrow \sigma'' \quad < \textbf{while}\ \ b\ \textbf{do}\ c\ , \sigma'' > \rightarrow \sigma'}{< \textbf{while}\ \ b\ \textbf{do}\ c\ , \sigma > \rightarrow \sigma'}$$

Example Program

```
Y := 1;
while ¬(X=1) do
        Y := Y * X;
        X := X - 1
```

Equivalence of commands

$c_0 \sim c_1$ iff

$$\forall \sigma, \sigma' \in \Sigma \forall . < c_0, \sigma > \rightarrow \sigma' \Leftrightarrow < c_1, \sigma > \rightarrow \sigma'$$

Proposition 2.8

while b do c ~ if b then (c; while b do c) else skip

Haskell Code

Extensions to IMP

- Abort
- Non-determinism
- Procedure
- Parallelism

Small Step Operational Semantics

- The natural semantics define evaluation in large steps
  - Abstracts “computation time”
- It is possible to define a small step operational semantics
  - $\langle a, \sigma \rangle \rightarrow_1 \langle a', \sigma' \rangle$
    - “one” step of executing a in a state σ yields a’ in a state σ'

Small Step Semantics for Additions

$$\frac{< a_0, \sigma > \rightarrow_1 < a'_0, \sigma >}{< a_0 + a_1, \sigma > \rightarrow_1 < a'_0 + a_1, \sigma >}$$

$$\frac{< a_1, \sigma > \rightarrow_1 < a'_1, \sigma >}{< n + a_1, \sigma > \rightarrow_1 < n + a'_1, \sigma >}$$

$$\frac{}{< n + m, \sigma > \rightarrow_1 < p, \sigma >} \; where \; p = n + m$$

Summary

- Operational semantics enables to naturally express program behavior
- Can handle
  - Non determinism
  - Concurrency
  - Procedures
  - Object oriented
  - Pointers and dynamically allocated structures
- But remains very closed to the implementation
  - Two programs which compute the same functions are not necessarily equivalent