Formal Semantics

The *syntax* of a programming language is formally defined by a grammar.

How can its *semantics* be formally defined? The semantics definition should capture the "meaning" of each syntactic construct.

Kinds of Formal Semantics

- Operational semantics
  - Example: natural semantics
  - Shows the effects of each construct on an abstract machine
  - Most useful for implementers who wish to build correct compilers or interpreters
- Axiomatic semantics
  - Describes the meanings by inference rules, using preconditions and postconditions
  - Most useful for programmers who wish to rigorously verify the correctness of their programs
- Denotational semantics
  - Maps each construct to a value, which is expressed in terms of the values of its syntactic subcomponents
  - Most useful for language designers who wish to develop cleaner language descriptions

Axiomatic Semantics

[Covered in section 8.6 of textbook]

Inference rules have the form:

$$P \{ S \} Q$$

- P = precondition (logical assertion)
- S = program fragment (e.g. Impcore)
- Q = postcondition (logical assertion)

Preliminary Notation

$P_{[x \to e]}$ denotes the assertion P with each occurrence of variable x replaced by expression e

Example: If P is the assertion "x < y" then $P_{[x \to (+\ x\ 1)]}$ is the assertion "x+1 < y"

Note: We'll freely convert between prefix and infix expression formats

Assignment rule (RO)

$$P_{[x \to e]} \ \{ \ (\text{set } x \ e) \ \} \ P$$

Example of (RO):

$$x\text{-}1 > -1 \ \{ \ (\text{set } x \ (- \ x \ 1)) \ \} \ x > -1$$

Note: To keep it simple, we're assuming that "set" works like "val" if variable x was not previously defined

Rules of consequence (R1)

Strengthening the precondition (R1a):

If

$$P \{ S \} Q$$

and

$$P' \rightarrow P$$

then

$$P' \{ S \} Q$$

Example of (R1a)

We know

$$x\text{-}1 > -1 \ \{ (\text{set } x\ (-\ x\ 1)) \}\ x > -1$$

and

$$x > 0 \rightarrow x\text{-}1 > -1 \ \text{[easy to prove]}$$

Therefore

$$x > 0 \ \{ (\text{set } x\ (-\ x\ 1)) \}\ x > -1$$

Rules of consequence (R1)

Weakening the postcondition (R1b):

If

$$P \{ S \} Q$$

and

$$Q \rightarrow Q'$$

then

$$P \{ S \} Q'$$

Example of (R1b)

We know

$$x > 0 \ \{ (\text{set } x \ (- \ x \ 1)) \} \ x > -1$$

and

$$x > -1 \rightarrow x \geq 0 \ \text{[when x is integer]}$$

Therefore

$$x > 0 \ \{ (\text{set } x \ (- \ x \ 1)) \} \ x \geq 0$$

Rules of consequence (R1)

(R1a) and (R1b) can be applied simultaneously:

If

$$P \{ S \} Q$$

and

$$P' \rightarrow P$$

and

$$Q \rightarrow Q'$$

then

$$P' \{ S \} Q'$$

Composition or Sequence rule (R2)

If

$$P_0 \{ S_1 \} P_1$$

and

$$P_1 \{ S_2 \} P_2$$

…

and

$$P_{n-1} \{ S_n \} P_n$$

then

$$P_0 \{ (\text{begin } S_1 S_2 \ldots S_n) \} P_n$$

Example of (R2)

We know

$$x > 0 \ \{ (\text{set } x \ (- \ x \ 1)) \} \ x \geq 0$$

and

$$x \geq 0 \ \{ (\text{set } y \ x) \} \ y \geq 0 \ \text{[by (R0)]}$$

Therefore

$$x > 0 \ \{ (\text{begin } (\text{set } x \ (- \ x \ 1)) \ (\text{set } y \ x)) \} \ y \geq 0$$

Alternation or Selection rule (R3)

If

$$P \wedge e \ \{ S_1 \} \ Q$$

and

$$P \wedge \neg e \ \{ S_2 \} \ Q$$

then

$$P \ \{ (\text{if } e \ S_1 \ S_2) \} \ Q$$

Note: e denotes any Boolean expression that contains no side effects

Example of (R3)

Suppose we want to prove

$$\text{true}\ \{\ (\text{if}\ (>\ x\ y)\ (\text{set}\ z\ x)\ (\text{set}\ z\ y))\ \}\ \ z \geq x \wedge z \geq y$$

To use (R3) we'd first need to show

$$\text{true} \wedge x > y\ \{\ (\text{set}\ z\ x)\ \}\ \ z \geq x \wedge z \geq y$$

and

$$\text{true} \wedge \neg\ x > y\ \{\ (\text{set}\ z\ y)\ \}\ \ z \geq x \wedge z \geq y$$

Example of (R3) continued

By assignment rule (R0) we know

$$x \geq x \wedge x \geq y \ \{ (\text{set } z\ x) \}\ z \geq x \wedge z \geq y$$

and

$$y \geq x \wedge y \geq y \ \{ (\text{set } z\ y) \}\ z \geq x \wedge z \geq y$$

These implications are easy to prove:

$$\text{true} \wedge x > y \ \rightarrow\ x \geq x \wedge x \geq y$$

and

$$\text{true} \wedge \neg\, x > y \ \rightarrow\ y \geq x \wedge y \geq y$$

Next use rule of consequence (R1), then apply (R3)

Here is the entire proof

1. $x \geq x \wedge x \geq y \ \{ (\text{set } z\ x) \}\ z \geq x \wedge z \geq y$ (R0)
2. $y \geq x \wedge y \geq y \ \{ (\text{set } z\ y) \}\ z \geq x \wedge z \geq y$ (R0)

3. $\text{true} \wedge x > y \rightarrow x \geq x \wedge x \geq y$ (axiom)
4. $\text{true} \wedge \neg\, x > y \rightarrow y \geq x \wedge y \geq y$ (axiom)

5. $\text{true} \wedge x > y \ \{ (\text{set } z\ x) \}\ z \geq x \wedge z \geq y$ (R1; 1,3)
6. $\text{true} \wedge \neg\, x > y \ \{ (\text{set } z\ y) \}\ z \geq x \wedge z \geq y$ (R1; 2,4)

7. $\text{true} \ \{ (\text{if } (> x\ y)\ (\text{set } z\ x)\ (\text{set } z\ y)) \}\ z \geq x \wedge z \geq y$ (R3; 5,6)

Iteration rule (R4)

If

$$P \wedge e \ \{ S \} \ P$$

then

$$P \ \{ (\text{while } e \ S) \} \ P \wedge \neg e$$

Here P is called a "loop invariant"

Finding an appropriate loop invariant often requires cleverness or insight

Example of (R4)

Suppose we want to prove

$$x \geq 0 \ \{ \text{(while (> x 0) (set x (- x 1)))} \} \ x = 0$$

To use (R4) we first need to choose an appropriate loop invariant:

$$x \geq 0$$

How do we know that $x \geq 0$ is an appropriate loop invariant?

Mostly by experience, some trial-and-error

Example of (R4) continued

Now (R4) in this case becomes:

If

$$x \geq 0 \wedge x > 0 \ \{ (\text{set } x \ (- \ x \ 1)) \} \ x \geq 0$$

then

$$x \geq 0 \ \{ (\text{while } (> \ x \ 0) \ (\text{set } x \ (- \ x \ 1))) \} \ x \geq 0 \wedge \neg \ x > 0$$

Here is the entire proof

1. $x-1 > -1$ { (set x (- x 1)) } $x > -1$ (R0)
2. $x \geq 0 \wedge x > 0 \rightarrow x-1 > -1$ (axiom)
3. $x > -1 \rightarrow x \geq 0$ (axiom, x is integer)
4. $x \geq 0 \wedge x > 0$ { (set x (- x 1)) } $x \geq 0$ (R1; 1,2,3)
5. $x \geq 0$ { (while (> x 0) (set x (- x 1))) } $x \geq 0 \wedge \neg x > 0$ (R4; 4)
6. $x \geq 0 \wedge \neg x > 0 \rightarrow x = 0$ (axiom)
7. $x \geq 0$ { (while (> x 0) (set x (- x 1))) } $x = 0$ (R1; 5,6)

Axioms should be obviously easy-to-prove statements using standard laws of logic and algebra

Loop termination

- Rule (R4) avoids the question of proving whether the loop terminates. If termination is not obvious, then termination must be proven by other means.
- Example:

  true { (while (!= x 0) (set x (- x 1))) } x = 0
- The above claim can be proved with (R4) plus other rules. But it is only useful if the loop terminates, that is, when $x \geq 0$ initially.
- Total correctness vs. partial correctness

Sample exercise to prove

Exponentiation algorithm:

$x \geq 0$

```
{ (set z 1) (set y x)
  (while (> y 0) (begin (set z (* z 2))
                        (set y (- y 1)))) }
```

$z = 2^x$

Hint: Loop invariant is $y \geq 0 \wedge z = 2^{x-y}$

Another sample exercise

Logarithm algorithm:

$x \geq 1 \wedge b \geq 2$

```
{ (set j x) (set k 0)
  (while (>= j b) (begin (set j (/ j b))
                         (set k (+ k 1)))) }
```

$k = \lfloor \log_b x \rfloor$

Hint: Loop invariant is

$j \geq 1 \wedge b \geq 2 \wedge k = \lfloor \log_b (x/j) \rfloor$

Yet another sample exercise

Russian Peasants' multiplication algorithm:

$x \geq 0$

```
{ (set n 0) (set a x) (set b y)
  (while (> a 0)
        (begin (if (= (mod a 2) 1) (set n (+ n b)) 0)
                   (set a (/ a 2)) (set b (* b 2)))) }
```

$n = x \cdot y$

Hint: Loop invariant is $a \geq 0 \wedge n = x \cdot y - a \cdot b$

Denotational Semantics

[Not covered in our textbook]

Map each syntactic construct to a mathematical value

- The value should be computable from the values associated with the pieces that combine to form this construct

Example of Den. Semantics

Num ::= Num Digit | Digit
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

nvalue[Num Digit] = 10*nvalue[Num] + dvalue[Digit]
nvalue[Digit] = dvalue[Digit]
dvalue[0] = 0
dvalue[1] = 1
...
dvalue[9] = 9

Semantic Domains

Integer = {…, -2, -1, 0, 1, 2, …}

Boolean = {true, false}

Value = {int}×Integer ∪ {bool}×Boolean

Store = Identifier → Value ∪ {undefined}

Notational convention:

Store s = {x → 345, y → false}

s(x) = (int, 345) and s(y) = (bool, false)

s(z) = undefined, s(w) = undefined, etc.

Operations on Stores

emptyStore : Store

emptyStore(x) = undefined, $\forall x \in$ Identifier

applyStore : Store$\times$Identifier $\rightarrow$ Value $\cup$ {undefined}

applyStore(s, x) = s(x)

updateStore : Store$\times$Identifier$\times$Value $\rightarrow$ Store

updateStore(s, x, v) y =

if x=y then v else s(y)

[note that this definition uses currying]

Example of updateStore

Store s = $\{x \rightarrow 345, y \rightarrow \text{false}\}$

Store s′ = updateStore(s, x, 99) =

$\{x \rightarrow 99, y \rightarrow \text{false}\}$

Store s″ = updateStore(s′, z, true) =

$\{x \rightarrow 99, y \rightarrow \text{false}, z \rightarrow \text{true}\}$

An Impcore-like language

Prog ::= Cmd*

Cmd ::= (begin Cmd*) | (set Ident Exp)
| (if Exp Cmd Cmd) | (while Exp Cmd)

Exp ::= Num | true | false | Ident
| (Binop Exp Exp) | (Unop Exp)

Binop ::= + | - | * | / | < | > | = | and | or

Unop ::= ~ | not

Semantic Functions

meaning : Prog $\rightarrow$ Store

execute : Cmd $\rightarrow$ Store $\rightarrow$ Store

evaluate : Exp $\rightarrow$ Store $\rightarrow$ Value $\cup$ {undefined}

nvalue : Num $\rightarrow$ Value

Den. Semantics of Program

meaning $[c_1\ c_2\ \ldots\ c_n]$ =

execute $[(\text{begin}\ c_1\ c_2\ \ldots\ c_n)]$

emptyStore

Den. Semantics of Begin Cmd

execute [(begin c_1 c_2 … c_n)] s =

execute [c_n]

(execute [c_{n-1}]

… (execute [c_2]

(execute [c_1] s)) …)

Den. Semantics of Set Cmd

execute [(set x e)] s =

    updateStore(s, x, evaluate [e] s)

Note: As we did before, we're assuming that "set" works like "val" if variable x was not previously defined

Den. Semantics of If Cmd

execute [(if e c_1 c_2)] s =

if v then execute [c_1] s

else execute [c_2] s

where (bool, v) = evaluate [e] s

Den. Semantics of While Cmd

```
execute [(while e c)] s =
      if v  then  execute [(while e c)]
                                (execute [c] s)
            else  s
      where (bool, v) = evaluate [e] s
```

Den. Semantics of While Cmd

Equivalent partially-curried version:

```
execute [(while e c)] = loop
      where loop s =
      if v then loop (execute [c] s) else s
            where (bool, v) = evaluate [e] s
```

Den. Semantics of Literal Exp

For all $n \in$ Num,

evaluate [n] s = (int, nvalue[n])

evaluate [true] s = (bool, true)

evaluate [false] s = (bool, false)

Den. Semantics of Id Exp

For all x $\in$ Ident,

evaluate [x] s =

    if v = undefined then error else v

    where v = applyStore(s, x)

Den. Semantics of Arith Exp

evaluate [(+ e_1 e_2)] s = (int, v_1 + v_2)

where (int, v_1) = evaluate [e_1] s

and (int, v_2) = evaluate [e_2] s

…

evaluate [(/ e_1 e_2)] s =

if v_2 = 0 then error else (int, v_1 / v_2)

where (int, v_1) = evaluate [e_1] s

and (int, v_2) = evaluate [e_2] s

Den. Semantics of Rel Exp

evaluate [(< e_1 e_2)] s = (bool, v_1 < v_2)

where (int, v_1) = evaluate [e_1] s

and (int, v_2) = evaluate [e_2] s

...

evaluate [(= e_1 e_2)] s = (bool, v_1 = v_2)

where (T, v_1) = evaluate [e_1] s

and (T, v_2) = evaluate [e_2] s

and T may be either int or bool

Den. Semantics of Logical Exp

evaluate [(and e_1 e_2)] s =

if v_1 then (bool, v_2) else (bool, false)

where (bool, v_1) = evaluate [e_1] s

and (bool, v_2) = evaluate [e_2] s

evaluate [(or e_1 e_2)] s =

if v_1 then (bool, true) else (bool, v_2)

where (bool, v_1) = evaluate [e_1] s

and (bool, v_2) = evaluate [e_2] s

Den. Semantics of Unary Exp

evaluate [(~ e)] s = evaluate [(- 0 e)] s

evaluate [(not e)] s =

if v then (bool, false) else (bool, true)

where (bool, v) = evaluate [e] s

Functions

Suppose we add functions to this language.
Assume each function takes one parameter.
Which parts will be affected?

Possible values will include function values:

$$\text{Value} = \{\text{int}\} \times \text{Integer} \cup \{\text{bool}\} \times \text{Boolean} \cup (\text{Value} \rightarrow \text{Value})$$

Den. Semantics of Functions

Anonymous function definitions:

Exp ::= ... | (lambda (Ident) Exp)

evaluate [(lambda (x) e)] s =
    fn v ⇒ evaluate [e] updateStore(s, x, v)

- Here v = value of the actual parameter
- For simplicity, we're using dynamic scope (static scope requires defining closures)

Den. Semantics of Functions

Function calls (applications):

Exp ::= ... | (Exp Exp)

evaluate $[(e_1\ e_2)]\ s$ =

(evaluate $[e_1]\ s$) (evaluate $[e_2]\ s$)

- Expressions do not have side effects

Kinds of Operational Semantics

- Natural semantics
  [Covered throughout our textbook]

- Translational semantics
  [Not covered in our textbook]
  - Translate the high-level language into an easily-understood lower-level language
  - We'll use an *abstract stack machine*

An Abstract Stack Machine

PUSH v, POP

ADD, SUB, MULT, DIV

LT, GT, EQ, NE, LE, GE

AND, OR, NOT

DUP

LOAD x

STORE x

JUMP L, JUMPT L, JUMPF L

CALL f, RET

INIT, HALT

Recall: Impcore-like language

Prog ::= Cmd*

Cmd ::= (begin Cmd*) | (set Ident Exp)
| (if Exp Cmd Cmd) | (while Exp Cmd)

Exp ::= Num | true | false | Ident
| (Binop Exp Exp) | (Unop Exp)

Binop ::= + | - | * | / | < | > | = | and | or

Unop ::= ~ | not

Trans. Semantics of Program

run [c_1 c_2 … c_n] =

INIT

execute [(begin c_1 c_2 … c_n)]

HALT

Trans. Semantics of Begin Cmd

execute [(begin c_1 c_2 … c_n)] =

execute [c_1]

execute [c_2]

…

execute [c_n]

Trans. Semantics of Set Cmd

```
execute [(set x e)] =
        evaluate [e]
        STORE  x
```

How to map symbolic labels (such as x) to addresses?

- Details omitted here
- Topic for a compilers course

Trans. Semantics of If Cmd

execute [(if e c_1 c_2)] =

    evaluate [e]
    JUMPF L_1
    execute [c_1]
    JUMP L_2
L_1: execute [c_2]
L_2:

Trans. Semantics of While Cmd

execute [(while e c)] =

L_1: evaluate [e]

JUMPF L_2

execute [c]

JUMP L_1

L_2:

Trans. Semantics of While Cmd

More efficient translation:

execute [(while e c)] =

```
        JUMP  L2
L1:     execute [c]
L2:     evaluate [e]
        JUMPT  L1
```

Trans. Semantics of Literal Exp

For all v $\in$ Num,

evaluate [v] =

PUSH v

evaluate [true] =

PUSH 1

evaluate [false] =

PUSH 0

Trans. Semantics of Id Exp

For all $x \in$ Ident,

evaluate [x] =

LOAD x

Trans. Semantics of Arith Exp

evaluate [(+ e_1 e_2)] =

evaluate [e_1]

evaluate [e_2]

ADD

Similarly

- (- e_1 e_2) uses SUB
- (* e_1 e_2) uses MULT
- (/ e_1 e_2) uses DIV

Trans. Semantics of Rel Exp

evaluate [(< e_1 e_2)] =

evaluate [e_1]

evaluate [e_2]

LT

Similarly

- (> e_1 e_2) uses GT
- (= e_1 e_2) uses EQ

Trans. Semantics of Logical Exp

evaluate [(and e_1 e_2)] =

evaluate [e_1]

evaluate [e_2]

AND

evaluate [(or e_1 e_2)] =

evaluate [e_1]

evaluate [e_2]

OR

Trans. Semantics of Logical Exp

Short-circuited "and":

evaluate [(and e_1 e_2)] =

    evaluate [e_1]
    DUP
    JUMPF L
    POP
    evaluate [e_2]

L:

Trans. Semantics of Logical Exp

Short-circuited "or":

evaluate [(or e_1 e_2)] =

```
        evaluate [e1]
        DUP
        JUMPT  L
        POP
        evaluate [e2]
L:
```

Trans. Semantics of Unary Exp

evaluate [(~ e)] =

    PUSH 0
    evaluate [e]
    SUB

evaluate [(not e)] =

    evaluate [e]
    NOT

Functions

Function definitions:

Cmd ::= ... | (define Ident (Ident) Exp)

Function calls:

Exp ::= ... | (Ident Exp)

Trans. Semantics of Functions

```
execute [(define f (x) e)] =
        JUMP  L
f:      evaluate [e]
        RET
L:

evaluate [(f e)] =
        evaluate [e]
        CALL  f
```

Summary

- Operational semantics
  - Examples: natural semantics and translational semantics
  - Shows the effects of each construct on an abstract machine
  - Most useful for implementers who wish to build correct compilers or interpreters
- Axiomatic semantics
  - Describes the meanings by inference rules, using preconditions and postconditions
  - Most useful for programmers who wish to rigorously verify the correctness of their programs
- Denotational semantics
  - Maps each construct to a value, which is expressed in terms of the values of its syntactic subcomponents
  - Most useful for language designers who wish to develop cleaner language descriptions