Semantics

In Text: Chapter 3

Outline

- Semantics:
  - Attribute grammars (static semantics)
  - Operational
  - Axiomatic
  - Denotational

Static Semantics

- CFGs cannot describe all of the syntax of programming languages—context-specific parts are left out
- Static semantics refers to type checking and resolving declarations; has nothing to do with "meaning" in the sense of run-time behavior
- Often described using an attribute grammar (AG) (Knuth, 1968)
- Basic idea: add to CFG by carrying some semantic information along inside parse tree nodes
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Dynamic Semantics

- No single widely acceptable notation or formalism for describing semantics
- Three common approaches:
  - Operational
  - Axiomatic
  - Denotational

Operational Semantics

- Gives a program's meaning in terms of its implementation on a real or virtual machine
- Change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine in needed
- A pure hardware interpreter is too expensive
- A pure software interpreter also has problems:
  - machine-dependent
  - Difficult to understand
- A better alternative: A complete computer simulation

Operational Semantics (cont.)

- The process:
  - Identify a virtual machine (an idealized computer)
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Operational semantics is sometimes called translational semantics, if an existing PL is used in place of the virtual machine

Operational Semantics Example

Pascal	Operational Semantics

for i := x to y do
  begin
    . . .
  end
	i := x
	loop: if i>y goto out
	. . .
	i := i + 1
	goto loop
	out: . . .

- Operational semantics could be much lower level:

```
    mov i,r1
    mov y,r2
    jmpifless(r2,r1,out)
    ...
out: ...
```

Evaluation of Operational Semantics

- Advantages:
  - May be simple, intuitive for small examples
  - Good if used informally
  - Useful for implementation
- Disadvantages
  - Very complex for large programs
  - Lacks mathematical rigor
- Uses:
  - Vienna Definition Language (VDL) used to define PL/I (Wegner 1972)
  - Compiler work

Axiomatic Semantics

- Based on formal logic (first order predicate calculus)
- Original purpose: formal program verification
- Approach: Define axioms or inference rules for each statement type in the language
- Such an inference rule allows one to transform expressions to other expressions
- The expressions are called assertions, and state the relationships and constraints among variables that are true at a specific point in execution
- An assertion before a statement is called a precondition
- An assertion following a statement is a postcondition

Weakest Preconditions

- Pre-post form: {P} statement {Q}
- A weakest precondition is the least restrictive precondition that will guarantee the postcondition
- An example:

$$a := b + 1 \quad \{a > 1\}$$

- One possible precondition: {b > 10}
- Weakest precondition: {b > 0 }

Program Proofs

- Program proof process:
  - The postcondition for the whole program is the desired results
  - Work back through the program to the first statement
  - If the precondition on the first statement is the same as the program spec, the program is correct

An Axiom for Assignment

- An axiom for assignment statements:

$$\{Q_{x\text{->}E}\}\ \ x := E\ \ \{Q\}$$

- Substitute E for every x in Q

$$\{\ P?\ \}\ x := y+1\ \ \{\ x > 0\ \}$$

$$P = x > 0\ _{x\ \text{->}\ y+1}$$

$$P = y+1 > 0$$

$$P = y \geq 0$$

- Basically, “undoing” the assignment and solving for y

Some Inference Rules

- The Rule of Consequence:

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

- For a sequence S1;S2 the inference rule is:

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

A Rule for Loops

- An inference rule for logical pretest loops:

{P} while B do S end {Q}

- The inference rule is:

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \text{while } B \text{ do } S\ \{I \text{ and } (\text{not } B)\}}$$

- Where I is the loop invariant.

Loop Invariant Characteristicss

I must meet the following conditions:

1. $P \Rightarrow I$ (the loop invariant must be true initially)
2. $\{I\}$ B $\{I\}$ (evaluation of the Boolean must not change the validity of I)
3. $\{I \text{ and } B\}$ S $\{I\}$ (I is not changed by executing the body of the loop)
4. $(I \text{ and } (\text{not } B)) \Rightarrow Q$ (if I is true and B is false, Q is implied)
5. The loop terminates (can be difficult to prove)

More on Loop Invariants

- The loop invariant I is:
  - A weakened version of the loop postcondition, and
  - Also the loop's precondition
- I must be:
  - Weak enough to be satisfied prior to the beginning of the loop, but
  - when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

Finding Loop Invariants

- Work backwards through a few iterations and look for a pattern

while y <> x do y:= y+1 {y = x}

{P?} y := y + 1 {y = x}

$P = \{y = x\}_{y \to y+1} = \{y = x - 1\}$ —last iteration

{P?} y := y + 1 {y = x - 1}

$P = \{y = x - 1\}_{y \to y+1} = \{y = x - 2\}$ —next to last

Finding Invariants (cont.)

- By extension, we get $I = \{ y < x \}$
- When we factor in that the loop may not be executed even once (when $y = x$), we get

$$I = \{ y \leq x \}$$

- This also satisfies loop termination, so
- $P = I = \{y \leq x\}$

Is I a Loop Invariant?

- Does $\{y \le x\}$ satisfy the 5 conditions?

(1) $\{y \le x\} \Rightarrow \{y \le x\}$?

(2) if $\{y \le x\}$ and $y <> x$ is then evaluated, is $\{y \le x\}$ still true?

(3) if $\{y \le x\}$ and $y <> x$ are true and then $y := y+1$ is executed, is $\{y \le x\}$ true?

(4) does $\{y \le x\}$ and $\{y = x\} \Rightarrow \{y = x\}$?

- Can you argue convincingly that the program segment terminates?

A Harder Loop Invariant Example

$$\{P\} \text{ while } y < x + 1 \text{ do } y := y + 1 \; \{y>5\}$$

$$\{y > 5\}_{y \to y + 1} \Rightarrow y > 4$$

$$\{y > 4\}_{y \to y + 1} \Rightarrow y > 3$$

etc.

- Tells us nothing about x because x is not in $Q \equiv \{y > 5\}$
- What else can we do?

Using Loop Criterion 4

- Try guessing invariant using criterion 4:
- {I and (not B)} $\Rightarrow$ Q
- I? and $y \geq x + 1 \Rightarrow y > 5$
- I? and $y > x \Rightarrow y > 5$
- any $x \geq 5$ satisfies implication
- so . . . let $I = \{x \geq 5\}$
- Do the 4 Axioms hold?

Evaluation of Axiomatic Semantics

- Advantages
  - Can be very abstract
  - May be useful in proofs of correctness
  - Solid theoretical foundations
- Disadvantages
  - Predicate transformers are hard to define
  - Hard to give complete meaning
  - Does not suggest implementation
- Uses of Axiomatic Semantics
  - Semantics of Pascal
  - Reasoning about correctness

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)
- Key idea: Define a function that maps a program (a syntactic object) to its meaning (a semantic object)

Denotational vs. Operational

- Denotational semantics is similar to high-level operational semantics, except:
  - Machine is gone
  - Language is mathematics (lamda calculus)
- The difference between denotational and operational semantics:
  - In operational semantics, the state changes are defined by coded algorithms for a virtual machine
  - In denotational semantics, they are defined by rigorous mathematical functions

Denotational Specification Process

1. Define a mathematical object for each language entity
2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

Program State

- The meaning of language constructs are defined only by the values of the program's variables
- The state of a program is the values of all its current variables, plus input and output state

$$s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$$

- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable:

$$\mathrm{VARMAP}(i_j, s) = v_j$$

Example: Decimal Numbers

<digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<dec_num> -> <digit> | <dec_num><digit>

- M_{dec}('0') = 0, M_{dec}('1') = 1, ..., M_{dec}('9') = 9
- M_{dec}(<dec_num>) Δ=

    case <dec_num> of

        <digit> $\Rightarrow$ M_{dec}(<digit>)

        <dec_num><digit> $\Rightarrow$

            10 $\times$ M_{dec}(<dec_num>) + M_{dec}(<digit>)

Expressions

- M_e(<expr>, s) Δ=

case <expr> of

<dec_num> ⇒ M_{dec}(<dec_num>, s)

<var> ⇒ VARMAP(<var>, s)

<binary_expr> ⇒

if (<binary_expr>.<operator> = ‘+’) then

Me(<binary_expr>.<left_expr>, s) +

Me(<binary_expr>.<right_expr>, s)

else

Me(<binary_expr>.<left_expr>, s) ×

Me(<binary_expr>.<right_expr>, s)

Statement Basics

- The meaning of a single statement executed in a state s is a new state s' (that reflects the effects of the statement)

  $M_{stmt}(\ Stmt\ ,\ s) = s'$

- For a sequence of statements:

  $M_{stmt}(\ Stmt1;\ Stmt2\ ,\ s)\ \Delta= M_{stmt}(\ Stmt2\ ,\ M_{stmt}(\ Stmt1\ ,\ s))$

  or

  $M_{stmt}(\ Stmt1;\ Stmt2\ ,\ s) = S''$ where

  $s' = M_{stmt}(\ Stmt1\ ,\ s)$

  $s'' = M_{stmt}(\ Stmt2\ ,\ s')$

Assignment Statements

- $M_a(x := E, s) \Delta=$

  $s' = \{<i_1', v_1'>, <i_2', v_2'>, \ldots, <i_n', v_n'>\}$,

  where for $j = 1, 2, \ldots, n$,

  $v_j' = VARMAP(i_j, s) \quad \text{if} \quad i_j \neq x$

  $v_j' = M_e(E, s) \quad \text{if} \quad i_j = x$

Sequence of Statements

```
x := 5;
y := x + 1;                P1      P
write(x * y);  } P2
```

Initial state $s_0 = <mem_0, i_0, o_0>$

$M_{stmt}(P , s) = M_{stmt}(P1 , \underbrace{M_{stmt}(x := 5 , s)}_{s_1})$

$s_1 = <mem_1, i_1, o_1>$ where

$VARMAP(x, s_1) = 5$

$VARMAP(z, s_1) = VARMAP(z, s_0)$ for all $z \neq x$

$i_1 = i_0, o_1 = o_0$

Sequence of Statements (cont.)

$$M_{stmt}(\ P1\ ,\ s_1) = M_{stmt}(\ P2\ ,\ \underbrace{M_{stmt}(\ y := x + 1\ ,\ s_1)}_{s_2})$$

$s_2 = \langle mem_2, i_2, o_2 \rangle$ where

$VARMAP(y, s_2) = M_e(\ x + 1\ ,\ s_1) = 6$

$VARMAP(z, s_2) = VARMAP(z, s_1)$ for all $z \neq y$

$i_2 = i_1$

$o_2 = o_1$

Sequence of Statements (cont.)

$M_{stmt}($ P2 $, s_2) = M_{stmt}($ write (x * y) $, s_2) = s_3$

$s_3 = \langle mem_3, i_3, o_3 \rangle$ where

$VARMAP(z, s_3) = VARMAP(z, s_2)$ for all z

$i_3 = i_2$

$o_3 = o_2 \bullet M_e($ x * y $, s_2) = o_2 \bullet 30$

Sequence of Statements (concl.)

So,

$M_{stmt}(P, s_0) = s_3 = \langle mem_3, i_3, o_3 \rangle$ where

$VARMAP(y, s_3) = 6$

$VARMAP(x, s_3) = 5$

$VARMAP(z, s_3) = VARMAP(z, s_0)$ for all $z \neq x, y$

$i_3 = i_0$

$o_3 = o_0 \bullet 30$

Logical Pretest Loops

- The meaning of the loop is the value of the program variables after the loop body has been executed the prescribed number of times, assuming there have been no errors
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions
- Recursion, when compared to iteration, is easier to describe with mathematical rigor

Logical Pretest Loops (cont.)

- M_l(while B do L, s) Δ=

  if Mb(B, s) = false then

  s

  else

  M_l(while B do L, M_s(L, s))

Evaluation of Denotational Semantics

- Advantages:
  - Compact & precise, with solid mathematical foundation
  - Provides a rigorous way to think about programs
  - Can be used to prove the correctness of programs
  - Can be an aid to language design
  - Has been used in compiler generation systems
- Disadvantages
  - Requires mathematical sophistication
  - Hard for programmer to use
- Uses
  - Semantics for Algol-60, Pascal, etc.
  - Compiler generation and optimization

Summary

- Each form of semantic description has its place:
  - Operational
    - Informal descriptions
    - Compiler work
  - Axiomatic
    - Reasoning about particular properties
    - Proofs of correctness
  - Denotational
    - Formal definitions
    - Provably correct implementations