Describing Syntax and Semantics – Chapter 3 (continued)

Describing the Meanings of Programs: Dynamic Semantics

- Dynamic semantics is the task describing the meaning of expressions, statements, and program rules.
- There is no universally accepted notation devised for dynamic semantics.

Operational Semantics

Is the idea of describing the meaning of a program by executing its statements on a machine, either real or simulated. The changes that occur in the machine's state (memory, registers, etc…) when it executes a given statement define the meaning of the statement.

The Basic Process

- Requires the construction of either a real or virtual machine in order to use with high-level languages.
- Hardware pure interpreter – machine language – too expensive
- Software pure interpreter – machine hardware characteristics would make the actions difficult to understand – semantic definition is machine dependent
- Software simulation
    - Low-level virtual computer
    - Memory, registers, execution process, etc… all simulated
    - Semantic definition easy to describe and understand
    - Requires a translator – source code to low-level language
    - Requires a virtual computer – simulator

Operational semantics is often used in programming textbooks and programming language reference manuals.

C Statement

```
for ( expr1; expr2; expr3 ) {
        …
}
```

Operational Semantics

```
        expr1;
loop:   if expr2 == 0 goto out
        …
        expr3;
        goto loop
out:    …
```

Evaluation

- Operational semantics provides an effective means of describing semantics as long as the descriptions are kept informal.
- Extremely complex if used formally – Vienna Definition Language, VDL, used to describe PL/I is so complex that it serves no practical purpose.

Axiomatic Semantics

- Based on formal mathematical logic – predicate calculus
- Original purpose – formal program verification – correctness proofs
- Axiom – logical statement that is assumed to be true.

Assertions

- Assertions are logical expressions called predicates.
- Assertion immediately preceding a program statement describes the constraints on program variables at that point – precondition
    - {x > 10} sum = 2 * x + 1
- Assertion immediately following a program statement describes new constraints on those variables – postcondition
    - sum = 2 * x + 1 {sum > 1}

Weakness Preconditions

- This is the least restrictive precondition that will guarantee the validity of the associated postcondition.
- sum = 2 * x + 1
    - possible preconditions are {x > 10}, {x > 50}, and {x > 1000}
    - weakest precondition is {x > 0} based on the postcondition {sum > 1}
- The postcondition for the entire program is the result of executing the last executable statement.
- If the precondition of the first statement, after working backwards through the program, matches the program specification then the program is correct.
- In most cases the weakest precondition can be computed only by an inference rule.
- Inference rule – method of inferring the truth of one assertion on the basis of the values of other assertions.

Assignment Statements

- Usual notation for specifying axiomatic semantics for a statement: {P} S {Q}
  - P is the precondition; S is the statement; Q is the postcondition
- Axiom: (x = E): $\{Q_{x->E}\}$ x = E{Q}
  - $P = Q_{x->E}$ is computed as Q with all instances of x replaced by E.

Examples:

a = b / 2 – 1 {a < 10} postcondition
b / 2 – 1 < 10 weakest precondition is computed by substitution
b < (10 + 1) * 2
b < 22 computed weakest precondition

x = x + y – 3 {x > 10}
x + y – 3 > 10
y > 10 – x + 3
y > 13 – x computed weakest precondition

{x > 3} x = x – 3 {x > 0}
x – 3 > 0
x > 3
Matches precondition – proven

{x > 5} x = x – 3 {x > 0}
x – 3 > 0
x > 3
Does not match precondition
{x > 5} implies {x > 3}

The Rule of Consequence – inference rule

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

If logical statement {P} S {Q} is true, the assertion P′ implies the assertion P, and the assertion Q implies the assertion Q′, then it can be inferred that {P′} S {Q′}. The rule of consequence says that a postcondition can always be weakened and a precondition can always be strengthened.

$$\frac{\{x > 3\}\ x = x - 3\ \{x > 0\},\ (x > 5) \Rightarrow (x > 3),\ (x > 0) \Rightarrow (x > 0)}{\{x > 5\}\ x = x - 3\ \{x > 0\}}$$

The first term was proven with the assignment axiom and the second and third terms are obvious. Therefore, by the rule of consequence, the consequent is true.

Sequences – Inference rule

{P1} S1 {P2}
{P2} S2 {P3}

S1 and S2 are two consecutive statements

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

y = 3 * x + 1	y + 3 < 10	y = 3 * x + 1	3 * x + 1 < 7
x = y + 3	y < 7 {P2}	{y < 7} {P2}	x < (7 – 1) / 3
{x < 10} {P3}			x < 2 {P1}

Selection – Inference rule

Axiomatic description: {P} if B then S1 else S2 {Q}

$$\frac{\{B \text{ and } P\}\ S1\ \{Q1\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\} \text{ if } B \text{ then } S1 \text{ else } S2\ \{Q\}}$$

if (x > 0) then
y = y – 1 {y > 0}
else
y = y + 1 {y > 0}

y – 1 > 0	
y > 1	precondition – then
y + 1 > 0	
y > -1	precondition – else – implies {y > 1}
	{y > 1} precondition for all

Logical Pretest Loops – Inference rule

Axiomatic description: {P} while B do S end {Q}

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

I is the loop invariant. The complexity lies in finding an appropriate loop invariant.

Loop invariant characteristics – I must meet the following conditions:

- P => I – the loop invariant must be true initially
- {I} B {I} – evaluation of Boolean must not change validity of I
- {I and B} S {I} – I is not changed by executing the body of the loop
- (I and (not B)) => Q – if I is true and B is false, is implied
- The loop terminates

To find a loop invariant, a method similar to that used for determining the inductive hypothesis in mathematical induction can be used. The relationship for a few cases is computed, with the hope that a pattern emerges that will apply to the general case.

Finding loop invariants is not always easy. It is helpful to understand the nature of these invariants.

- A loop invariant is a weakened version of the loop postcondition and also a precondition for the loop.
- I must be weak enough to be satisfied prior to the beginning of loop execution, but when combined with the loop exit condition it must be strong enough to force the truth of the post condition.

Evaluation

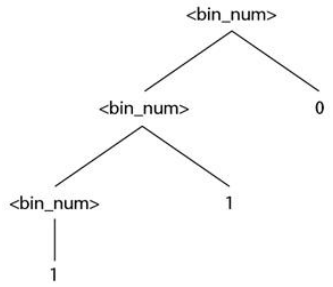
- There must be an axiom or an inference rule for each statement type in the language. This is a difficult task for some statements.
- Design a language with the axiomatic method in mind. The language would be very small and simple.
- Powerful tool for research into program correctness proofs.

Denotation Semantics

- Most rigorous, widely known method for describing the meaning of programs.
- Based on recursive function theory.
- Fundamental concept is to:
    - Define a mathematical object for each language entity.
    - Define a function that maps instances of that entity onto instances of the corresponding mathematical object.
    - Because the objects are rigorously defined, they represent the exact meaning of their corresponding entities.

Example: binary number 110

<bin_num> -> 0
| 1
| <bin_num> 0
| <bin_num> 1

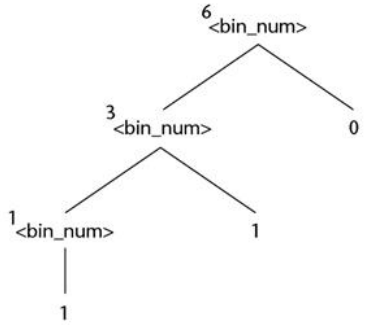


Meaning objects must be associated with the first two grammar rules. The other two rules are computational rules because they combine a terminal (can be associated with an object) with a nonterminal (represents some construct).

Let the domain of semantic values of the objects be N, nonnegative decimal numbers. The semantic function, named M_{bin}, maps the syntactic objects to the objects in N. The function M_{bin} is defined as follows:

M_{bin}(‘0’) = 0, M_{bin}(‘1’) = 1 ‘1’ syntactical digit – 1 mathematical digit
M_{bin}(<bin_num> ‘0’) = 2 * M_{bin}(<bin_num>)
M_{bin}(<bin_num> ‘1’) = 2 * M_{bin}(<bin_num>) + 1

The meanings, denoted objects (decimal numbers), can be attached to the nodes of the above parse tree yielding the parse tree below. This is syntax-directed semantics. Syntactic entities are mapped to mathematical objects with concrete meaning.



– meaning of syntactical decimal literals

<dec_num> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| <dec_num> (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

– denotational mapping for these syntax rules

M_{dec}(‘0’) = 0, … , M_{dec}(‘9’) = 9
M_{dec}(<dec_num> ‘0’) = 10 * M_{dec}(<dec_num>), … ,
M_{dec}(<dec_num> ‘9’) = 10 * M_{dec}(<dec_num>) + 9

The State of a Program

- Operational semantics – state changes defined by coded algorithms.
- Denotational semantics – state changes defined by rigorous mathematical functions.
- State of a program: $\{<i_1, v_1>, <i_2, v_2>, \ldots , <i_n, v_n>\}$
  - i is the name of some variable
  - v is the current value of the corresponding variable
- VARMAP(i_j, s) = v_j mapping function
  - i is the name of some variable
  - s is the state of the corresponding variable
  - v is the value of the corresponding variable

Expressions

- Contain: integer literals, variables, at most one binary operator (*, +), result is an integer
- Map expressions onto Z ∪ {**error**}
  - Z set of all integers
  - **error** is the expression error value – undefined expression value
- Δ= is used to define mathematical functions

M_e(<expr>, s) Δ=

```
    case <expr> of
        <dec_num> => Mdec(<dec_num>, s)
        <var> =>    if (VARMAP(<var>, s) == undef) then
                        error
                    else
                        VARMAP(<var>, s)
        <binary_expr> =>
            if (Me(<binary_expr>.<left_expr>, s) == undef OR
                Me(<binary_expr>.<right_expr>, s) == undef) then
                error
            else 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)
```

Assignment Statements

- Expression evaluation plus setting the left-side variable to the expression value.
  - A mapping function that maps a state to a state.

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

if ($M_e(E, s)$ == **error**) then

error

else

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

for $j = 1, 2, \ldots, n$, $v_j' = VARMAP(i_j, s)$ if i_j != x;

$M_e(E, s)$ if i_j == x

Logical Pretest Loops

- Assumptions:
  - M_{sl} – mapping function that maps statement lists to states
  - M_b – mapping functions that maps Boolean expressions to Boolean values (or error)

$M_l(\text{while B do L}, s)\ \Delta =$

if ($M_b(B, s)$ == **undef**) then

error

else if ($M_b(B, s)$ == false) then

s

else if ($M_{sl}(L, s)$ == error) then

error

else

$M_l(\text{while B do L}, M_{sl}(L, s))$

- The meaning of the loop is the value of the program variables after the loop statements have executed the prescribed number of times, assumes no errors.
- The loop has been converted from iteration to recursion. Recursion is easier to describe with mathematical rigor.

Evaluation

- Can be used to prove program correctness
- Provides a framework for thinking of programs in a rigorous way
- Can be used as an aid to language design
- Of little use to language users because of its complexity