DYNAMIC SEMANTICS

Dynamic Semantics

- Describe the meaning of expressions, statements, and program units
- No single widely acceptable notation or formalism for describing semantics
- Two common approaches:
  - Operational
  - 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

Operational Semantics Definition Process

1. Design an appropriate intermediate language. Each construct of the intermediate language must have an obvious and unambiguous meaning
2. Construct a virtual machine (an interpreter) for the intermediate language. The virtual machine can be used to execute either single statements, code segments, or whole programs

An Example

<table>
<tr><th>C</th><th>Operational Semantics</th></tr>
<tr><td><pre>for (expr1; expr2; expr3)
{
  . . .
}</pre></td><td><pre>   expr1;
loop: if expr2 == 0 goto out
        . . .
        expr3;
        goto loop
out: . . .</pre></td></tr>
</table>

- The virtual computer is supposed to be able to correctly "execute" the instructions and recognize the effects of the "execution"

Key Points of Operational Semantics

- Advantages
  - May be simple and intuitive for small examples
  - Good if used informally
  - Useful for implementation
- Disadvantages
  - Very complex for large programs
  - Lacks mathematical rigor

Typical Usage of Operational Semantics

- Vienna Definition Language (VDL) used to define PL/I (Wegner 1972)
- Unfortunately, VDL is so complex that it serves no practical purpose

Denotational Semantics

- The most rigorous, widely known method for describing the meaning of programs
- Solely based on recursive function theory
- Originally developed by Scott and Strachey (1970)

Denotational Semantics

- Key Idea
  - Define for each language entity both a mathematical object, and a function that maps instances of that entity onto instances of the mathematical object
- The basic idea
  - There are rigorous ways of manipulating mathematical objects but not programming language constructs

Denotational Semantics

- Difficulty
  - How to create the objects and the mapping functions?
- The method is named *denotational*, because the mathematical objects denote the meaning of their corresponding syntactic entities

Denotational vs. Operational

- Both denotational semantics and operational semantics are defined in terms of state changes in a virtual machine
- In operational semantics, the state changes are defined by **coded algorithms** in the machine
- In denotational semantics, the state change is defined by **rigorous mathematical functions**

Program State

- Let the state s of a program be a set of pairs as follows:

  $\{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$
  - Each i is the name of a variable
  - The associated v is the current value of the variable
  - Any v can have the special value **undef**, indicating that the associated variable is undefined
- Let VARMAP be a function as follows:

  $VARMAP(i_j, s) = v_j$

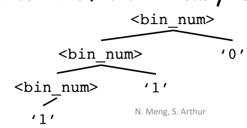
Program State

- Most semantics mapping functions for programs and program constructs map from states to states
- These state changes are used to define the meanings of programs and program constructs
- Some language constructs, such as expressions, are mapped to values, not state changes

An Example

- CFG for binary numbers

  <bin_num> -> '0'
  <bin_num> -> '1'
  <bin_num> -> <bin_num> '0'
  <bin_num> -> <bin_num> '1'
- Parse tree of the binary number 110



Example Semantic Rule Design

- Mathematical objects
  - Decimal number equivalence for each binary number
- Functions
  - Map binary numbers to decimal numbers
  - Rules with terminals as RHS are translated as direct mappings from terminals to mathematical objects
  - Rules with nonterminals as RHS are translated as manipulations on mathematical objects

Example Semantic Rules

Syntax Rules	Semantic Rules
<bin_num>->'0' <bin_num>->'1' <bin_num>-><bin_num> '0' <bin_num>-><bin_num> '1'	M_{bin}('0')=0 M_{bin}('1')=1 M_{bin}(<bin_num> '0')= 2*M_{bin}(<bin_num>) M_{bin}(<bin_num> '1')= 2*M_{bin}(<bin_num>)+1

Expressions

- CFG for expressions

  <expr> -> <dec_num> | <var> | <binary_expr>
  <binary_expr> -> <l_expr> <op> <r_expr>
  <l_expr> -> <dec_num> | <var>
  <r_expr> -> <dec_num> | <var>
  <op> -> + | *

Expressions

M_e(<expr>, s) Δ=
case <expr> of
<dec_num> ⇒ M_{dec}(<dec_num>)
<var> ⇒ VARMAP(<var>, s)
<binary_expr> ⇒
if (<binary_expr>.<op> = '+') then
M_e(<binary_expr>.<l_expr>, s) +
M_e(<binary_expr>.<r_expr>, s)
else
M_e(<binary_expr>.<l_expr>, s) ×
M_e(<binary_expr>.<r_expr>, s)

Statement Basics

- The meaning of a single statement executed in a state s is a new state s', which reflects the effects of the statement

  M_{stmt}(stmt , s) = s'

Assignment Statements

M_a(x := E, s) Δ=

s' = {<i_1', v_1'>, <i_2', v_2'>, ..., <i_n',v_n'>},

where for j = 1, 2, ..., n,

$v_j' = VARMAP(i_j, s)$ if $i_j \neq x$

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

Sequence of Statements

M_{stmt}(stmt1; stmt2 , s) Δ=

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')

Sequence of Statements

```
x := 5;              } P0
y := x + 1;    } P1
write(x * y);  } P2
```

Initial state s_0 = <mem_0, i_0, o_0>

M_{stmt}(P_0, s_0) = M_{stmt}(P_1, $\underbrace{M_a(x := 5 , s_0)}_{s_1}$)

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 ≠ x

$i_1 = i_0$, $o_1 = o_0$

Sequence of Statements

```
x := 5;              } P0
y := x + 1;    } P1
write(x * y);  } P2
```

M_{stmt}(P_1, s_1) = M_{stmt}(P_2, $\underbrace{M_a(y := x + 1, s_1)}_{s_2}$)

s_2 = <mem_2, i_2, o_2> where

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

VARMAP(z, s_2) = VARMAP(z, s_1) for all z ≠ y

$i_2 = i_1$, $o_2 = o_1$

Sequence of Statements

```
x := 5;              } P0
y := x + 1;    } P1
write(x * y);  } P2
```

M_{stmt}(P_2, s_2) = M_{stmt}(write(x * y), s_2) = s_3

s_3 = <mem_3, i_3, o_3> where

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

$i_3 = i_2$, $o_3 = o_2 \cdot M_e(x * y, s_2) = o_2 \cdot 30$

Sequence of Statements

Therefore,

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

$VARMAP(y, s_3) = 6$

$VARMAP(x, s_3) = 5$

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

$i_3 = i_0$

$o_3 = o_0 \cdot 30$

Logical Pretest Loops

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

Logical Pretest Loop

- M_l(while B do L, s) Δ=

  if M_b(B, s) = false then

  s

  else

  M_l(while B do L, $\mathbf{M_{stmt}}$**(L, s)**)

Postest Loop ?

- M_{ptl}(do L until not B, s) Δ = ?

Key Points of Denotational Semantics

- Advantages
  - **Compact & precise**, with solid mathematical foundation
  - Provide a **rigorous** way to think about programs
  - Can be used to prove the correctness of programs
  - Can be an aid to language design

Key Points of Denotational Semantics

- Disadvantages
  - **Require 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 semantics
  - Informally describe the meaning of language constructs in terms of their effects on an ideal machine
- Denotational semantics
  - Formally define mathematical objects and functions to represent the meanings

Reference

[1] Cormac Flanagan, A Simple Langauge of Arithmetic Expressions, https://classes.soe.ucsc.edu/cmps203/Winter11/02-arith-bigstep.ppt.pdf

[2] Cormac Flanagan, Operational Semantics: Big-Step vs. Small-Step, https://classes.soe.ucsc.edu/cmps203/Winter11/04-smallstep.ppt.pdf