Semantics

Semantics

- Semantics is a precise definition of the meaning of a syntactically and type-wise correct program.
- Ideas of meaning:
  - Operational Semantics
    - The meaning attached by compiling using compiler C and executing using machine M. Ex: Fortran on IBM 709
  - Axiomatic Semantics
    - Formal specification to allow us to rigorously prove what the program does with a systematic logical argument
  - Denotational Semantics
    - Statements as state transforming functions
- We start with an informal, operational model

Program State

- Definition: The state of a program is the binding of all active objects to their current values.
- Maps:
  1. The pairing of active objects with specific memory locations, and
  2. The pairing of active memory locations with their current values.
- E.g. given i = 13 and j = -1
  - Environment = {<i,154>,<j,155>}
  - Memory = {<0, undef>, … <154, 13>, <155, -1> …}

- The current statement (portion of an abstract syntax tree) to be executed in a program is interpreted relative to the current state.
- The individual steps that occur during a program run can be viewed as a series of state transformations.

Assignment Semantics

- Three issues or approaches
  - Multiple assignment
  - Assignment statement vs. expression
  - Copy vs. reference semantics

Multiple Assignment

- Example:
- a = b = c = 0;
- Sets all 3 variables to zero.

Assignment Statement vs. Expression

- In most languages, assignment is a statement; cannot appear in an expression.
- In C-like languages, assignment is an expression.
  - Example:
  - *if (a = 0) ... // an error?*
  - *while (*p++ = *q++) ; // strcpy*
  - *while (p = p->next) ... // ???*

Copy vs. Reference Semantics

- Copy: a = b;
  - *a, b* have same value.
  - Changes to either have no effect on other.
  - Used in imperative languages.
- Reference
  - *a, b* point to the same object.
  - A change in object state affects both
  - Used by many object-oriented languages.

State Transformations

- **Defn**: The *denotational semantics* of a language defines the meanings of abstract language elements as a collection of state-transforming functions.
- **Defn**: A *semantic domain* is a set of values whose properties and operations are independently well-understood and upon which the rules that define the semantics of a language can be based.

Partial Functions

- State-transforming functions in the semantic definition are necessarily **partial functions**
- A partial function is one that is not well-defined for all possible values of its domain (input state)

C-Like Semantics

- *State* – represent the set of all program states
- A *meaning* function M is a mapping:

  M: Program → State

  M: Statement x State → State

  M: Expression x State → Value

Meaning Rule - Program

- The meaning of a *Program* is defined to be the meaning of the *body* when given an initial state consisting of the variables of the *decpart* initialized to the *undef* value corresponding to the variable's type.

```
State M (Program p) {
    // Program = Declarations decpart; Statement body
    return M(p.body, initialState(p.decpart));
}
public class State extends HashMap { ... }
```

```
State initialState (Declarations d) {
    State state = new State( );
    for (Declaration decl : d)
      state.put(decl.v,  Value.mkValue(decl.t));
    return state;
}
```

Statements

- M: Statement x State → State

- Abstract Syntax

  Statement = Skip | Block | Assignment | Loop | Conditional

```
State M(Statement s, State state) {
    if (s instanceof Skip) return M((Skip)s, state);
    if (s instanceof Assignment) return M((Assignment)s, state);
    if (s instanceof Block) return M((Block)s, state);
    if (s instanceof Loop) return M((Loop)s, state);
    if (s instanceof Conditional) return M((Conditional)s, state);
    throw new IllegalArgumentException( );
}
```

Meaning Rule - Skip

- The meaning of a *Skip* is an identity function on the state; that is, the state is unchanged.

```
State M(Skip s, State state) {
    return state;
}
```

Meaning Rule - Assignment

- The meaning of an Assignment statement is the result of replacing the value of the *target* variable by the computed value of the *source* expression in the current state

```
Assignment  = Variable target;
              Expression source
```

```
State M(Assignment a, State state) {
    return state.onion(a.target, M(a.source, state));
}

// onion replaces the value of target in the map by the source
// called onion because the symbol used is sometimes sigma σ
   to represent state
```

Meaning Rule - Conditional

- The meaning of a conditional is:
  - If the test is true, the meaning of the thenbranch;
  - Otherwise, the meaning of the elsebranch

Conditional = Expression test;
    Statement thenbranch, elsebranch

```
State M(Conditional c, State state) {
if (M(c.test, state).boolValue( ))
    return M(c.thenbranch);
else
    return M(e.elsebranch, state);
}
```

Expressions

- M: Expression x State → Value

- Expression = Variable | Value | Binary | Unary
- Binary = BinaryOp op; Expression term1, term2
- Unary = UnaryOp op; Expression term
- Variable = String id
- Value = IntValue | BoolValue | CharValue | FloatValue

Meaning Rule – Expr in State

- The meaning of an expression in a state is a value defined by:
  1. If a value, then the value. Ex: 3
  2. If a variable, then the value of the variable in the state.
  3. If a Binary:
     a) Determine meaning of term1, term2 in the state.
     b) Apply the operator according to rule 8.8 (perform addition/subtraction/multiplication/division)

  ...

```
Value M(Expression e, State state) {
if (e instanceof Value)  return (Value)e;
if (e instanceof Variable)  return (Value)(state.get(e));
if (e instanceof Binary) {
    Binary b = (Binary)e;
    return applyBinary(b.op, M(b.term1, state),
        M(b.term2, state);
}
...
```

Formalizing the Type System

- Approach: write a set of function specifications that define what it means to be type safe
- Basis for functions: Type Map, *tm*
  - $tm = \{ <v_1,t_1>, <v_2,t_2>, \ldots <v_n,t_n>\}$
  - Each v_i represents a variable and t_i its type
  - Example:
    - int i,j; boolean p;
    - *tm* = { <i, int>, <j, int>, <p, boolean> }

Declarations

- How is the type map created?
  - When we declare variables
- typing: Declarations → Typemap
  - i.e. declarations produce a typemap
- More formally
  - typing(Declarations d) = $\bigcup_{i=1}^{n} < d_i.v, d_i.t >$
  - i.e. the union of every declaration variable name and type
  - In Java we implemented this using a HashMap

Semantic Domains and States

- Beyond types, we must determine semantically what the syntax means
- **Semantic Domains** are a formalism we will use
  - Environment, γ = set of pairs of variables and memory locations
    - γ = {<i, 100>, <j, 101>} for i at Addr 100, j at Addr 101
  - Memory, μ = set of pairs of memory locations and the value stored there
    - μ = {<100, 10> , <101, 50>} for Mem(100)=10, Mem(101)=50
  - State of the program, σ = set of pairs of active variables and their current values
    - σ = {<i,10>, <j, 50>} for i=10, j=50

State Example

- x=1; y=2; z=3;
  - At this point σ = {<x,1>,<y,2>,<z,3>}
  - Notation: σ(y)=2
- y=2*z+3;
  - At this point σ = {<x,1>,<y,9>,<z,3>}
- w=4;
  - At this point σ = {<x,1>,<y,9>,<z,3>, <w,4>}

- Can also have expressions; e.g. σ(x>0) = true

Overriding Union

State transformation represented using the Overriding Union

X $\overline{\cup}$ Y =replace all pairs <x,v> whose first member matches a pair <x,w> from Y by <x,w> and then add to X any remaining pairs in Y

Example:

$$\sigma_1 = \{<x,1>,<y,2>,<z,3>\}$$

$$\sigma_2 = \{<y,9>,<w,4>$$

$$\sigma_1 \overline{\cup} \sigma_2 = \{<x,1>,<y,9>,<z,3>,<w,4>\}$$

This will be used for assignment of a variable

Denotational Semantics

Σ : Set of all program states σ

M : Meaning function

- Meaning function
  - Input: abstract class, current state
  - Output: new state

$$M : Class \times \Sigma \rightarrow \Sigma$$

Let's revisit our Meaning Rules and redefine them using our more Formal Denotational Semantics

Denotational Semantics

$M : Program \rightarrow \Sigma$

$M(Program\ \mathrm{p}) = M(p.body, \sigma_{init})$

$\sigma_{init} = \{< v_1, undef >, < v_2, undef >, \ldots\ldots, < v_n, undef >\}$

Meaning of a program: produce final state

This is just the meaning of the body in an initial state

Java implementation:

```
State M (Program p) {
        // Program = Declarations decpart; Statement body
        return M(p.body, initialState(p.decpart));
}
public class State extends HashMap { ... }
```

Meaning for Statements

- M : Statement × State $\rightarrow$ State
- M (Statement s, State σ) =

M ((Skip) s, σ)	if s is a Skip
M ((Assignment) s, σ)	if s is Assignment
M ((Conditional) s, σ)	if s is Conditional
M ((Loop) s, σ)	if s is a Loop
M ((Block) s, σ)	if s is a Block

Semantics of Skip

- Skip

$$M(Skip\ s, State\ \sigma) = \sigma$$

  - Skip statement can't change the state

Semantics of Assignment

- Evaluate expression and assign to var

$$M : Assignment \times \Sigma \rightarrow \Sigma$$
$$M(Assignment \text{ a}, State\ \sigma) = \sigma \overline{U}\{< a.target, M(a.source, \sigma) >\}$$

- Examples of: x=a+b

$$\sigma = \{< a,3 >, < b,1 >, < x,88 >\}$$
$$M(x = a + b;, \sigma) = \sigma \overline{U}\{< x, M(a + b, \sigma) >\}$$
$$\sigma = \{< a,3 >, < b,1 >, < x,4 >\}$$

Semantics of Conditional

$$M(Conditional\ c, State\ \sigma)$$
$$= M(c.thenbranch, \sigma) \quad if\ M(c.test, \sigma)\ is\ true$$
$$= M(c.elsebranch, \sigma) \quad otherwise$$

```
If (a>b) max=a; else max=b
```

$$\sigma = \{< a,3 >< b,1 >\}$$
$$M(\text{if (a > b)max = a;else max = b;}, \sigma)$$
$$= M(\text{max} = a;, \sigma) \quad if\ M(a > b, \sigma)\ is\ true$$
$$= M(\text{max} = b;, \sigma) \quad otherwise;$$

Conditional, continued

$$\sigma = \{< a,3 >< b,1 >\}$$

$$M(\text{if (a > b)max = a; else max = b;}, \sigma)$$

$$= M(\text{max} = a;, \sigma) \quad \textit{since } M(a > b, \sigma) \textit{ is true}$$

$$= \sigma \overline{U} \{< \text{max}, 3 >\}$$

$$= \sigma \{< a,3 >, < b,1 >, < \text{max}, 3 >\}$$

Semantics of Block

- Block is just a sequence of statements

$$M(Block\ b, State\ \sigma)$$

$$= \sigma \qquad \textit{if } b = \varphi$$

$$= M((Block)b_{2...n}, M((Statement)b_1, \sigma)) \ \textit{ if } b = b_1 b_2 ... b_n$$

- Example for Block b:

  fact = fact * i;

  i = i – 1;

Block example

- b_1 = fact = fact * i;
- b_2 = i = i – 1;

b

- M(b,σ) = M(b_2,M(b_1,σ))

= M(i=i-1,M(fact=fact*i,σ))

= M(i=i-1,M(fact=fact*i,{<i,3>,<fact,1>}))

=M(i=i-1,{<i,3>,<fact,3>})

={<i,2>,<fact,3>}

Semantics of Loop

- Loop = Expression test; Statement body

$$
\begin{aligned}
&M(Loop\ l, State\ \sigma) \\
&\quad = M(l, M(l.body, \sigma)) \qquad if\ \ M(l.test, \sigma)\ \text{is true} \\
&\quad = \sigma \qquad\qquad otherwise
\end{aligned}
$$

- Recursive definition

Loop Example

- Initial state σ={<N,3>}

```
fact=1;
i=N;
while (i>1) {
    fact = fact * i;
    i = i -1;
}
```

After first two statements, σ = {<fact,1>,<N,3>,<i,3>}

Loop Example

σ = {<fact,1>,<N,3>,<i,3>}

M(while(i>1) {…}, σ)

= M(while(i>1) {…}, M(fact=fact*i; i=i-1;, σ)

= M(while(i>1) {…}, {<fact,3>,<N,3>,<i,2>})

= M(while(i>1) {…}, {<fact,6>,<N,3>,<i,1>})

= M(σ)

={<fact,6>,<N,3>,<i,1>}

Defining Meaning of Arithmetic Expressions for Integers

First let's define ApplyBinary, meaning of binary operations:

$$ApplyBinary : Operator \times Value \times Value \to Value$$

$$\begin{aligned}
&ApplyBinary(Operator\ op, Value\ v_1, Value\ v_2) \\
&= v_1 + v_2 && if\ op = + \\
&= v_1 - v_2 && if\ op = - \\
&= v_1 \times v_2 && if\ op = * \\
&= floor\left(\left|\frac{v_1}{v_2}\right|\right) \times sign(v_1 \times v_2) && if\ op = /
\end{aligned}$$

Denotational Semantics for Arithmetic Expressions

Use our definition of ApplyBinary to expressions:

$$M : Expression \times State \to Value$$

$$\begin{aligned}
&M(Expression\ \ e, State\ \sigma) \\
&= e && if\ \ e\ is\ a\ Value \\
&= \sigma(e) && if\ \ e\ is\ a\ Variable \\
&= ApplyBinary(e.op, \\
&\qquad M(e.term1, \sigma), \\
&\qquad M(e.term2, \sigma)) && if\ \ e\ is\ a\ Binary
\end{aligned}$$

Recall: op, term1, term2, defined by the Abstract Syntax
term1,term2 can be any expression, not just binary

Arithmetic Example

- Compute the meaning of x+2*y
- Current state σ={<x,2>,<y,-3>,<z,75>}

- Want to show: M(x+2*y,σ) = -4
  - x+2*y is Binary
  - From M(Expression e, State σ) this is

    ApplyBinary(e.op, M(e.term1, σ), M(e.term2,σ))
    = ApplyBinary(+,M(x,σ),M(2*y,σ))
    = ApplyBinary(+,2,M(2*y,σ))

    M(2*y,σ) is also Binary, which expands to:

    ApplyBinary(*,M(2,σ), M(y,σ))
    = ApplyBinary(*,2,-3)   = -6

    Back up: ApplyBinary(+,2,-6) = -4

Java Implementation

```
Value M(Expression e, State state) {
if (e instanceof Value)  return (Value)e;
if (e instanceof Variable)  return (Value)(state.get(e));
if (e instanceof Binary) {
   Binary b = (Binary)e;
   return applyBinary(b.op, M(b.term1, state),
      M(b.term2, state);
}
...
```

Code close to the denotational semantic definition!