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 $\rightarrow$ State
  - M: Statement x State $\rightarrow$ State
  - M: Expression x State $\rightarrow$ 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.

```java
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 $M(\text{Statement } s, \text{ State } state)$ {
    if ($s$ instanceof Skip) return $M((\text{Skip})s, state)$;
    if ($s$ instanceof Assignment) return $M((\text{Assignment})s, state)$;
    if ($s$ instanceof Block) return $M((\text{Block})s, state)$;
    if ($s$ instanceof Loop) return $M((\text{Loop})s, state)$;
    if ($s$ instanceof Conditional) return $M((\text{Conditional})s, state)$;
    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 $M(\text{Skip } s, \text{ State } state)$ {
    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 × 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;
    - \( tm = \{ <i, \text{int}>, <j, \text{int}>, <p, \text{boolean}> \} \)
Declarations

• How is the type map created?
  – When we declare variables

• typing: Declarations \( \rightarrow \) Typemap
  – i.e. declarations produce a typemap

• More formally
  – \( \text{typing}(\text{Declarations } d) = \bigcup_{i=1}^{n} <d_i.v, d_i.t> \)
  – 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, \( \gamma = \) set of pairs of variables and memory locations
    • \( \gamma = \{<i, 100>, <j, 101>\} \) for \( i \) at Addr 100, \( j \) at Addr 101
  – Memory, \( \mu = \) set of pairs of memory locations and the value stored there
    • \( \mu = \{<100, 10>, <101, 50>\} \) for Mem(100)=10, Mem(101)=50
  – State of the program, \( \sigma = \) set of pairs of active variables and their current values
    • \( \sigma = \{<i,10>, <j, 50>\} \) for \( i=10, j=50 \)
State Example

• $x=1; \ y=2; \ z=3;$
  – At this point $\sigma = \{<x,1>,<y,2>,<z,3>\}$
  – Notation: $\sigma(y)=2$
• $y=2*z+3;$
  – At this point $\sigma = \{<x,1>,<y,9>,<z,3>\}$
• $w=4;$
  – At this point $\sigma = \{<x,1>,<y,9>,<z,3>, \ w,4\}$

• Can also have expressions; e.g. $\sigma(x>0) = true$

Overriding Union

State transformation represented using the Overriding Union

$X \bigcup \ Y =$ 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: $\sigma_1 = \{<x,1>,<y,2>,<z,3>\}$
$\sigma_2 = \{<y,9>,<w,4>\}$
$\sigma_1 \bigcup \sigma_2 = \{<x,1>,<y,9>,<z,3>,<w,4>\}$

This will be used for assignment of a variable
Denotational Semantics

$\Sigma : \text{Set of all program states } \sigma$

$M : \text{Meaning function}$
- Meaning function
  - Input: abstract class, current state
  - Output: new state

$M : \text{Class } \times \Sigma \rightarrow \Sigma$

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

Denotational Semantics

$M : \text{Program } \rightarrow \Sigma$

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

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

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

- $M : \text{Statement} \times \text{State} \rightarrow \text{State}$
- $M (\text{Statement } s, \text{State } \sigma) =$
  - $M ((\text{Skip}) s, \sigma)$ if $s$ is a Skip
  - $M ((\text{Assignment}) s, \sigma)$ if $s$ is Assignment
  - $M ((\text{Conditional}) s, \sigma)$ if $s$ is Conditional
  - $M ((\text{Loop}) s, \sigma)$ if $s$ is a Loop
  - $M ((\text{Block}) s, \sigma)$ if $s$ is a Block

Semantics of Skip

- Skip

$$M (\text{Skip } s, \text{State } \sigma) = \sigma$$

- Skip statement can’t change the state
Semantics of Assignment

• Evaluate expression and assign to var

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

• Examples of: \( x = a + b \)

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

Semantics of Conditional

\[ M(\text{Conditional} c, \text{State} \sigma) \]
\[ = M(c.\text{thenbranch}, \sigma) \quad \text{if } M(c.\text{test}, \sigma) \text{ is true} \]
\[ = M(c.\text{elsebranch}, \sigma) \quad \text{otherwise} \]

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

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

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

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

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

\[= \sigma \bar{U} \{< \text{max}, 3 >\}\]

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

Semantics of Block

- Block is just a sequence of statements

\[M (\text{Block } b, \text{State } \sigma)\]

\[= \sigma \quad \text{if } b = \varnothing\]

\[= M ((\text{Block})b_{2..n}, M ((\text{Statement})b_1, \sigma)) \quad \text{if } b = b_1b_2...b_n\]

- Example for Block b:
  
  fact = fact * i;
  i = i - 1;
Block example

• \( b_1 = \) \( \text{fact} = \text{fact} \times i; \)  
• \( b_2 = \) \( i = i - 1; \)  
• \( M(b, \sigma) = M(b_2, M(b_1, \sigma)) \)  
  \( = M(i=i-1, M(\text{fact}=\text{fact} \times i, \sigma)) \)  
  \( = M(i=i-1, M(\text{fact}=\text{fact} \times i, \{<i,3>,<\text{fact},1>\})) \)  
  \( = M(i=i-1, \{<i,3>,<\text{fact},3>\}) \)  
  \( = \{<i,2>,<\text{fact},3>\} \)

Semantics of Loop

• Loop = Expression test; Statement body

\[
M(\text{Loop } l, \text{State } \sigma) \\
= M(l, M(l.\text{body}, \sigma)) \quad \text{if } M(l.\text{test}, \sigma) \text{ is true} \\
= \sigma \quad \text{otherwise}
\]

• Recursive definition
Loop Example

• Initial state $\sigma = \{<N,3>\}$

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

After first two statements, $\sigma =$
$\{<fact,1>,<N,3>,<i,3>\}$

Loop Example

$\sigma = \{<fact,1>,<N,3>,<i,3>\}$

$M(\text{while}(i>1) \{\ldots\}, \sigma )$
$= M(\text{while}(i>1) \{\ldots\}, M(\text{fact}=\text{fact}*i; \ i=i-1;, \ \sigma )$
$= M(\text{while}(i>1) \{\ldots\}, \{<fact,3>,<N,3>,<i,2>\})$
$= M(\text{while}(i>1) \{\ldots\}, \{<fact,6>,<N,3>,<i,1>\})$
$= M(\sigma )$
$=\{<fact,6>,<N,3>,<i,1>\}$
Defining Meaning of Arithmetic Expressions for Integers

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

\[
\text{ApplyBinary} : \text{Operator} \times \text{Value} \times \text{Value} \rightarrow \text{Value}
\]

\[
\text{ApplyBinary}(\text{Operator } op, \text{Value } v_1, \text{Value } v_2) = \begin{cases} 
v_1 + v_2 & \text{if } op = + \\
v_1 - v_2 & \text{if } op = - \\
v_1 \times v_2 & \text{if } op = * \\
\lfloor \frac{v_1}{v_2} \rfloor \times \text{sign}(v_1 \times v_2) & \text{if } op = /
\end{cases}
\]

Denotational Semantics for Arithmetic Expressions

Use our definition of ApplyBinary to expressions:

\[
M : \text{Expression} \times \text{State} \rightarrow \text{Value}
\]

\[
M \left( \text{Expression } e, \text{State } \sigma \right) = \begin{cases} 
e & \text{if } e \text{ is a Value} \\
\sigma(e) & \text{if } e \text{ is a Variable} \\
\text{ApplyBinary}(e,op, \\
M \left( e.\text{term1}, \sigma \right), \\
M \left( e.\text{term2}, \sigma \right)) & \text{if } e \text{ is a Binary}
\end{cases}
\]

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 $\sigma=\{<x,2>,<y,-3>,<z,75>\}$

• Want to show: $M(x+2*y,\sigma) = -4$
  – $x+2*y$ is Binary
  – From $M(\text{Expression } e, \text{ State } \sigma)$ this is
    
    ApplyBinary($e$.op, $M(e$.term1, $\sigma$), $M(e$.term2, $\sigma$))
    = ApplyBinary($+$,$M(x,\sigma)$,$M(2*y,\sigma)$)
    = ApplyBinary($+$,$2$,$M(2*y,\sigma)$)
    $M(2*y,\sigma)$ is also Binary, which expands to:
    
    ApplyBinary($\ast$,$M(2,\sigma)$, $M(y,\sigma)$)
    = ApplyBinary($\ast$,$2$,$-3$) = -6

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

Java Implementation

```java
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!