CSCI312 Principles of Programming Languages

Semantics

Xu Liu

Contents

7.1 Motivation

To provide an authoritative definition of the meaning of all language constructs for:

1. Programmers
2. Compiler writers
3. Standards developers

A programming language is complete only when its syntax, type system, and semantics are well-defined.

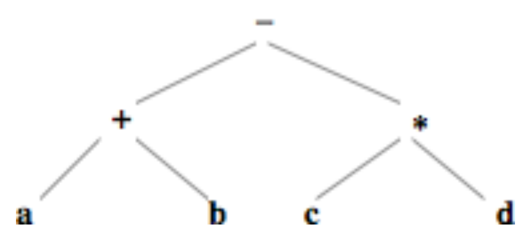
Semantics is a precise definition of the meaning of a syntactically and type-wise correct program.

Ideas of meaning:

- The meaning attached by compiling using compiler C and executing using machine M. Ex: Fortran on IBM 709.
- Axiomatize statements -- Chapter 12
- Statements as state transforming functions

This chapter uses an informal, operational model.

7.2 Expression Semantics



- (a + b) - (c * d)
- Polish Prefix: - + a b * c d
- Polish Postfix: a b + c d * -
- Cambridge Polish: (- (+ a b) (* c d))

Infix uses associativity and precedence to disambiguate.

Associativity of Operators

Language	+ - * /	Unary -	**	== != < ...
C-like	L	R		L
Ada	L	non	non	non
Fortran	L	R	R	L

Meaning of: `a < b < c`

Precedence of Operators

Operators	C-like	Ada	Fortran
Unary -	7	3	3
**		5	5
* /	6	4	4
+ -	5	3	3
== !=	4	2	2
< <= ...	3	2	2
not	7	2	2

Short Circuit Evaluation

a and b evaluated as:

    if a then b else false

a or b evaluated as:

    if a then true else b

Example

```
Node p = head;
while (p != null && p.info != key)
    p = p.next;
if (p == null) // not in list
    ...
else // found it
    ...
```

Versus

```
boolean found = false;
while (p != null && ! found) {
        if (p.info == key)
                found = true;
        else
                p = p.next;
}
```

Side Effect

A change to any non-local variable or I/O.

What is the value of:

```
i = 2; b = 2; c = 5;
a = b * i++ + c * i;
```

7.3 Program State

The state of a program is the collection of all active objects and their current values.

Maps:

1. The pairing of active objects with specific memory locations,
2. and the pairing of active memory locations with

their current values.

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.

For the purposes of this chapter, use only a map from a variable to its value; like a debugger watch window, tied to a particular statement.

```
// compute the factorial of n
void main ( ) {
  int n, i, f;
   n = 3;
   i = 1;
   f = 1;
   while (i < n) {
          i = i + 1;
          f = f * i;
   }
}
```

// compute the factorial of n	n	i	f
`void main ( ) {`			
`int n, i, f;`			
`n = 3;`	undef	undef	undef
`i = 1;`	3	undef	undef
`f = 1;`			
`while (i < n) {`			
`i = i + 1;`			
`f = f * i;`			
`}`			
`}`			

```
// compute the factorial of n
void main ( ) {
  int n, i, f;
   n = 3;
   i = 1;
   f = 1;
   while (i < n) {
          i = i + 1;
          f = f * i;
   }
}
```

n	i	f
3	undef	undef
3	1	undef

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
          i = i + 1;
          f = f * i;
  }
}
```

n	i	f
3	1	undef
3	1	1

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
          i = i + 1;
          f = f * i;
  }
}
```

n	i	f
3	1	1
3	1	1

// compute the factorial of n	n	i	f
`void main ( ) {`			
`int n, i, f;`			
`n = 3;`			
`i = 1;`			
`f = 1;`			
`while (i < n) {`			
`i = i + 1;`	3	1	1
`f = f * i;`	3	2	1
`}`			
`}`			

// compute the factorial of n	n	i	f
`void main ( ) {`			
`int n, i, f;`			
`n = 3;`			
`i = 1;`			
`f = 1;`			
`while (i < n) {`	3	2	2
`i = i + 1;`			
`f = f * i;`	3	2	1
`}`			
`}`			

// compute the factorial of n	n	i	f
void main ( ) {			
int n, i, f;			
n = 3;			
i = 1;			
f = 1;			
while (i < n) {	3	2	2
i = i + 1;	3	2	2
f = f * i;	3	3	2
}	3	3	6
}			

// compute the factorial of n	n	i	f
void main ( ) {			
int n, i, f;			
n = 3;			
i = 1;			
f = 1;			
while (i < n) {	3	3	6
i = i + 1;			
f = f * i;			
}			
}	3	3	6

7.4 Assignment Semantics

Issues

- Multiple assignment
- Assignment statement vs. expression
- Copy vs. reference semantics

Multiple Assignment

Example:

```
a = b = c = 0;
```

Sets all 3 variables to zero.

Problems???

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 (ch = getc(fp)) ...  // ???`
  - `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.*

```
public void add (Object word, Object number) {
    Vector set = (Vector) dict.get(word);
    if (set == null) {  // not in Concordance
        set = new Vector( );
        dict.put(word, set);
    }
    if (allowDupl || !set.contains(number))
        set.addElement(number);
}
```

7.5 Control Flow Semantics

To be complete, an imperative language needs:

- Statement sequencing
- Conditional statement
- Looping statement

Sequence

`s1 s2`

Semantics: in the absence of a branch:

- First execute `s1`
- Then execute `s2`
- Output state of `s1` is the input state of `s2`

Conditional

IfStatement → if (*Expresion*) *Statement*

[else *Statement*]

Example:

```
if (a > b)
    z = a;
else
    z = b;
```

If the test expression is true,

then the output state of the conditional is the output state of the then branch,

else the output state of the conditional is the output state of the else branch.

Loops

WhileStatement → while (*Expression*) *Statement*

The expression is evaluated.

If it is true, first the statement is executed,

and then the loop is executed again.

Otherwise the loop terminates.

Semantic Interpretation

Program State

The state of a program is the collection of all active objects and their current values.

Two maps in principle:

1. active objects $\longleftrightarrow$ memory locations,
2. active memory locations $\longleftrightarrow$ current values.

Only one map for our class:

variables $\longleftrightarrow$ current values

Relations with program execution

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.

Example (two maps):

Environment

– *i, j at memory locations 154, 155*

{ <i, 154>, <j, 155> }

State

– *i has value 13, j has value -1*

{ ..., <154, 13>, <155, -1>, ...}

Simplified to one map:

{ <i, 13>, <j, -1> }

8.1 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.

Example showing state transition

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
        i = i + 1;
        f = f * i;
  }
}
```

```
// compute the factorial of n
void main ( ) {
  int n, i, f;
   n = 3;
   i = 1;
   f = 1;
   while (i < n) {
          i = i + 1;
          f = f * i;
   }
}
```

{<n, undef> <i, undef> <f, undef>}

{<n, 3> <i, undef> <f, undef>}

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
        i = i + 1;
        f = f * i;
  }
}
```

{<n, 3> <i, undef> <f, undef>}

{<n, 3> <i, 1> <f, undef>}

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
        i = i + 1;
        f = f * i;
  }
}
```

{<n, 3> <i, 1> <f, undef>}

{<n, 3> <i, 1> <f, 1>}

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
        i = i + 1;
        f = f * i;
  }
}
```

{<n, 3> <i, 1> <f, 1>}

{<n, 3> <i, 1> <f, 1>}

```
// compute the factorial of n
void main ( ) {
  int n, i, f;
   n = 3;
   i = 1;
   f = 1;
   while (i < n) {
            i = i + 1;
            f = f * i;
   }
}
```

{<n, 3> <i, 1> <f, 1>}

{<n, 3> <i, 2> <f, 1>}

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
              i = i + 1;
              f = f * i;
  }
}
```

{<n, 3>   <i, 2>   <f, 1>}

{<n, 3>   <i, 2>   <f, 2>}

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
            i = i + 1;
            f = f * i;
  }
}
```

{<n, 3> <i, 2> <f, 2>}

{<n, 3> <i, 2> <f, 2>}

```
// compute the factorial of n
void main ( ) {
 int n, i, f;
  n = 3;
  i = 1;
  f = 1;
  while (i < n) {
            i = i + 1;
            f = f * i;
  }
}
```

{<n, 3>   <i, 3>   <f, 6>}

8.2 C++Lite Semantics

State – represent the set of all program states

A *meaning* function M is a mapping:

$M: Program \rightarrow State$

$M: Statement \times State \rightarrow State$

$M: Expression \times State \rightarrow Value$

Meaning Rule 8.1

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.

decpart: the declaration part of the program.

```
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 8.2

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;
}
```

```
public class Skip extends Statement {
...
public State meaning(State state) {
    return state;
}
```

```
public abstract class Statement {
...
public abstract State meaning(State state) ;
...
}
```