CSC312 Principles of Programming Languages : Semantics (2)

Semantic Interpretation

8.1 State Transformations and Partial Functions
8.2 Semantics of Clite
8.3 Semantics with Dynamic Typing
8.4 A Formal Treatment of Semantics

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
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

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

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

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

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

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

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

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

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

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

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

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

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

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

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

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

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

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

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

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

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

```
// compute the factorial of n
1    void main ( ) {
2     int n, i, f;
3      n = 3;
4      i = 1;
5      f = 1;
6      while (i < n) {
7                 i = i + 1;
8                 f = f * i;
9      }
10   }
```

{<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) ;
...
}
```

Meaning Rule 8.3

The output state is computed from the input state by replacing the value of the *target* variable by the computed value of the *source* expression.

Assignment = Variable target;

Expression source

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

```
public class Assignment extends Statement {
private Variable target;
private Expression source;
...
public State meaning(State state) {
    return state.add(target, source.value(state));
}
// ??? add, meaning, value
```

Meaning Rule 8.4

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, state);
    else
        return M(e.elsebranch, state);
}
```

```
public class Conditional extends Statement {
private Expression test;
private Statement thenbranch, elsebranch;
...
public State meaning(State state) {
    if (test.value(state).boolValue( ))
        return thenbranch.meaning(state);
    else
        return elsebranch.meaning(state);
}
```

Meaning Rule 8.5

The meaning of a loop is:

- *If the test is false, return the current state.*
- *Otherwise, apply this rule to its body, return the new state.*

Loop = Expression test;

Statement body;

```
State M(Loop l, State state) {
    if (M(l.test, state).boolValue( ))
        return M(l, M(l.body, state));
    else
        return state;
}
```

```
public class Loop extends Statement {
private Expression test;
private Statement body;
...
public State meaning(State state) {
    if (test.value(state).boolValue( ))
        return meaning ( body.meaning(state));
    else
        return state;
}
```

Meaning Rule 8.6

The meaning of a block of statements is:

- *aggregated meaning of the list of statements it contains.*

Block = Statement *

```
State M(Block b, State state) {
    for (Statement s : b.members)
        state = M (s, state);
    return state;
}
```

```
public class Block extends Statement {
private stmts;
...
public State meaning(State state) {
    for (s=stmts.next())
        state = M (s, state);
    return state;
}
```

Expression Semantics

Defn: A *side effect* occurs during the evaluation of an expression if, in addition to returning a value, the expression alters the state of the program.

Ignore for now.

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 8.7

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

...

```
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);
    }
    ...
```

```
public class IntValue extends Value {
private int intval;
...
public Value value(State state) { ??? }
public int intValue( ) { ??? }
public boolean boolValue( ) { ??? }
...
```

```
public class IntValue extends Value {
private int intval;
...
public Value value(State state) { return this; }
public int intValue( ) { ??? }
public boolean boolValue( ) { ??? }
...
```

```
public class IntValue extends Value {
private int intval;
...
public Value value(State state) { return this; }
public int intValue( ) { return intval; }
public boolean boolValue( ) { ??? }
...
```

```
public class IntValue extends Value {
private int intval;
...
public Value value(State state) { return this; }
public int intValue( ) { return intval; }
public boolean boolValue( ) { return intvalue != 0; }
...
```

Meaning Rule 8.8

1. If either term is undefined, the program is meaningless.
2. Do mathematical calculations (with necessary type conversions) for “+”, “-”, “*”, “/”
3. For relations operators (e.g., “<“, “>”), compare the expression with “true” (or “false”)
4. Boolean operator:
   - *“a && b”: if a then b else false*
   - *“a || b”: if a then true else b*

```
Value applyBinary (Operator op, Value v1, Value v2) {
    if (op.val.equals(Operator.PLUS)) // only consider int
        return new IntValue(v1.intValue()+v2.intValue());
    if (op.val.equals(Operator.GreaterThan)
        return v1.Value()>v2.Value;
    if (op.val.equals(Operator.AND))
        if (v1.boolValue())
            return v2.boolValue();
        else
            return FALSE;
    ...
}
```

Practice on our Gee Language Parser

```
class Number( Expression ):
    def __init__(self, val):
        self.val = int(val)

    def __str__(self):
        return str(self.val)

    def value(self, state):
        ???
```

Practice on our Gee Language Parser (Examples for your Project 3)

```
class Number( Expression ):
    def __init__(self, val):
        self.val = int(val)

    def __str__(self):
        return str(self.val)

    def value(self, state):
        return self.val
```

```
class BinaryExpr( Expression ):
    def __init__(self, op, left, right):
        self.op = op
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.op) + " " + str(self.left) + " " + str(self.right)

    def value(self, state):
        ???
```

```
def value(self, state):
    left = self.left.value(state)
    right = self.right.value(state)
    if self.op == "+":
        return left + right
    if self.op == "-":
        return left - right
    if self.op == "*":
        return left * right
    if self.op == "/":
        return left / right
```

```
class Assign( Statement ):
        def __init__(self, var, expr):
            self.var = str(var)
            self.expr = expr

        def __str__( self ):
                    return "= " + self.var + " " + str(self.expr)

        ???
```

```
class Assign( Statement ):
        def __init__(self, var, expr):
            self.var = str(var)
            self.expr = expr

        def __str__( self ):
                    return "= " + self.var + " " + str(self.expr)

        def meaning(self, state):
                state[self.var] = self.expr.value(state)
                return state
```