CSc 520

Principles of Programming Languages

50: Semantics — Introduction

Christian Collberg

`collberg@cs.arizona.edu`

Department of Computer Science

University of Arizona

Formal Semantics

- In order for
  1. compiler writers to know exactly how to implement a language, and
  2. language users to know exactly what (combinations of) language constructs mean,

  the meaning of a language needs to be defined.
- Most definitions of real languages are in a stylized, but informal, English.
- It is also possible to give formal semantic language definitions.

Formal Semantics...

- In practice, most languages are not defined in a formal, precise, mathematical way.
- There have been some attempts, for example Modula-2, Algol 68, and PL/I.
- “Simple” languages such as Scheme and Haskell are comparatively easy to define formally, compared to C, C++, Java, etc.

Formal Semantics — Modula-2

- The Modula-2 specification was written in *VDM-SL (Vienna Development Method - Specification Language),* a formalism for giving a precise definition of a programming language in a denotational style.
- It was over 500 pages long, and didn’t include specifications of the standard libraries.
- Wirth’s original Modula-2 report was 28 pages.
- For a history of this disastrous standardization effort, see `http://www.scifac.ru.ac.za/cspt/sc22wg13.htm`.
- Note also that Modula-2 is a very simple language compared to Ada, C++, Java, etc.

Formal Semantics — PL/I

- *VDL (Vienna Definition Language)* was used to specify PL/I.
- A specification has two parts:
  1. A translator that specified a translation into an abstract syntax tree,
  2. an interpreter of the abstract syntax tree.
- VDL is a kind of operational semantics.
- PL/I is large and complex.
- The resulting (large) document was called the Vienna Telephone Directory. It was impossible to comprehend.

Methods

In this class we will consider two methods for defining the semantics of programming languages:

- Operational semantics define a computation by giving step-by-step transformations on a abstract machine that simulate the execution of the program.
- Denotational semantics constructs a mathematical object (typically a function) which is the meaning of the program.

Contextual Constraints

- A compiler performs syntactic and semantic analysis. There really isn't a sharp distinction between the two.
- Is `String x; ···; print x/2` a syntactic or semantic error?
- Some would say that it violates the static semantic rules of the language, and hence is a semantic (not a syntactic) error.
- Others would say it violates context-sensitive syntax rules of the language. I.e., they'd consider the program as a whole to determine if it is well-formed or not.
- We will use the term contextual constraints for those rules that restricts the programs which are considered well-formed.

Operational Semantics

- Operational Semantics specifies a language through the steps by which each program is executed.
- This is often done informally. For example, the statement `while` E `do` C is specified as
  1. Evaluate E to a truthvalue B;
  2. If $B = \mathbf{true}$ then execute C, then repeat from 1).
  3. If $B = \mathbf{false}$, terminate.
- The emphasis is on specifying the steps needed to execute the program. This makes the specification useful for language implementers.

Operational Semantics…

- We need two things:
  1. an abstract syntax, and
  2. an interpreter.
- The abstract syntax defines the structure of each construct in the language, for example, that an if-statement consists of three parts: the test e, the then-part c_1 and the else-part c_2:

  if ::= e:bool_expr c_1:statement c_2:statement

  Note that no syntactic information is given.
- The interpreter generates a sequence of machine configurations that define the program's semantics. The interpreter is defined by rewriting rules.

Operational Sem. — Peano Arithmetic

Abstract Syntax ($N \in \mathrm{Nat}$, the Natural Numbers):

N ::= 0 | S(N) | (N + N) | (N × N)

Interpreter:

$$I : N \rightarrow N$$

$$
\begin{aligned}
I\,[\![(n+0)]\!] &\Rightarrow n\\
I\,[\![(m+S(n))]\!] &\Rightarrow S(I\,[\![(m+n)]\!])\\
I\,[\![(n\times 0)]\!] &\Rightarrow 0\\
I\,[\![(m\times S(n))]\!] &\Rightarrow I\,[\![((m\times n)+m)]\!]
\end{aligned}
$$

where $m, n \in \mathrm{Nat}$

Operational Sem. — Peano Arithmetic

- The rewrite rules are used to turn an expression into standard form, containing only S (succ) and 0.
- $S(S(S(S(0)))) = 4$.

Operational Sem. — Simple…

- *Simple* is a language with if-statements, while-statements, assignment-statements, and integer arithmetic.
- The semantic function I interprets commands.
- The semantic function ν interprets expressions.
- The store σ maps variables to their values.
- Assignments update the store.
- The result of the interpretation (the semantics of the program) is the resulting store.

Operational Sem. — Simple...

Interpreter:

$$\begin{aligned} I &: C \times \Sigma \rightarrow \Sigma \\ \nu &: E \times \Sigma \rightarrow T \cup Z \end{aligned}$$

Semantic Equations:

$$\begin{aligned} I(skip, \sigma) &= \sigma \\ I(V := E, \sigma) &= \sigma[V \mapsto \nu(E, \sigma)] \\ I(C_1 \;;\, C_2, \sigma) &= E(C_2, E(C_1, \sigma)) \\ I(\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, \sigma) &= I(C_1, \sigma) \text{ if } \nu(E, \sigma) = \textbf{true} \\ & I(C_2, \sigma) \text{ if } \nu(E, \sigma) = \textbf{false} \end{aligned}$$

Operational Sem. — Simple...

Interpreter:

$$\begin{aligned} & \textbf{while } E \textbf{ do } C \textbf{ end} = \\ & \qquad \textbf{if } E \textbf{ then } (C;\ \textbf{while } E \textbf{do } C \textbf{ end}) \textbf{ else skip} \\ \nu(V, \sigma) &= \sigma[V] \\ \nu(N, \sigma) &= N \\ \nu(E_1 + E_2, \sigma) &= \nu(E_1, \sigma) + \nu(E_2, \sigma) \\ \nu(E_1 = E_2, \sigma) &= \textbf{true} \text{ if } \nu(E, \sigma) = \nu(E, \sigma) \\ &= \textbf{false} \text{ if } \nu(E, \sigma) \neq \nu(E, \sigma) \end{aligned}$$

Denotational Semantics

- We think of each program as implementing a mathematical function.
- An imperative program is a function from inputs to outputs. This function is the meaning of the program.
- Example

```
exec [[while E do C]] =
   let exec-while env sto =
      let Boolean tr = evaluate [[E]] env sto in
         if tr then
            exec-while env (exec [[C]] env sto)
         else sto
   in
      exec-while
```


Denotational Semantics...

- We need three things:
  1. an abstract syntax,
  2. a semantic algebra defining a computational model, and
  3. valuation functions.
- The valuation functions map the syntactic constructs of the language to the semantic algebra.
- Denotational semantics relies on defining an object in terms of its constituent parts.

Denotational Sem. — Peano Arithmetic

Abstract Syntax ($N \in \mathrm{Nat}$, the Natural Numbers):

N ::= 0 | S(N) | (N + N) | (N × N)

Semantic Algebra:

$$+ \;:\; \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$$

Valuation Function:

$$\begin{aligned} D &: \mathrm{Nat} \to \mathrm{Nat} \\ D[\![(n+0)]\!] &= D[\![n]\!] \\ D[\![(m+S(n))]\!] &= D[\![(m+n)]\!]+1 \\ D[\![(n\times 0)]\!] &= 0 \\ D[\![(m\times S(n))]\!] &= D[\![((m\times n)+m)]\!] \end{aligned}$$

where $m, n \in \mathrm{Nat}$

Denotational Sem. — Simple

Abstract Syntax:

- C ∈ Command
- E ∈ Expression
- O ∈ Operator
- N ∈ Numeral
- V ∈ Variable

C ::= V := E | if E then C_1 else C_2 end | while E do C end | C_1 ; C_2 | skip

E ::= V | N | E_1 O E_2 | (E)

O ::= + | - | * | / | = | < | > | <>

Denotational Sem. — Simple...

Semantic Algebra:

$$\begin{aligned} \tau &\in T = true, false; && \text{the boolean values} \\ \zeta &\in Z = \{... -1, 0, 1, ...\}; && \text{the integers} \\ + &: Z \to Z \to Z \\ = &: Z \to Z \to T \\ \sigma &\in S = \mathrm{Variable} \to \mathrm{Numeral}; && \text{the state} \end{aligned}$$

Valuation Functions:

$$\begin{aligned} C &\in C \to (S \to S) \\ E &\in E \to E \to (N \cup T) \end{aligned}$$

Denotational Sem. — Simple...

$$\begin{aligned} C[\![\mathbf{skip}]\!]\,\sigma &= \sigma \\ C[\![V{:=}E]\!]\,\sigma &= \sigma[V \mapsto E[\![E]\!]\,\sigma \\ C[\![C1;C2]\!] &= C[\![C2]\!]\,C[\![C1]\!] \\ C[\![\mathbf{if}\ E\ \mathbf{then}\ C_1\mathbf{else}\ C_2\ \mathbf{end}]\!]\,\sigma &= C[\![C_1]\!]\,\sigma \text{ if } E[\![E]\!]\,\sigma = \mathbf{true} \\ &= C[\![C_2]\!]\,\sigma \text{ if } E[\![E]\!]\,\sigma = \mathbf{false} \\ C[\![\mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{end}]\!]\,\sigma &= \\ & \lim_{n\to\infty} C[\![(\mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else\ skip\ enc} \\ E[\![V]\!]\,\sigma &= \sigma(V) \\ E[\![N]\!] &= \zeta \\ E[\![E_1+E_2]\!] &= E[\![E_1]\!]\,\sigma + E[\![E_2]\!]\,\sigma \\ E[\![E_1=E_2]\!]\,\sigma &= E[\![E_1]\!]\,\sigma = E[\![E_2]\!]\,\sigma \end{aligned}$$

Concrete Syntax of Wren

Wren

- Wren is a small imperative language that we will be using as a running example.
- The complete concrete syntax of Wren is given in the next few slides.

Concrete Syntax

program ::= `program` identifi er `is` block

block ::= declaration_seq `begin` command_seq `end`

declaration_seq ::= | declaration declaration_seq

declaration ::= `var` variable_list `:` type `;`

type ::= `integer` | `boolean`

variable_list ::= variable | variable `,` variable_list

command_seq ::= command | command `;` command_seq

command ::= variable := expr | `skip`
  | `read` variable | `write` integer_expr
  | `while` boolean expr `do` command_seq `end while`
  | `if` boolean_expr `then` command_seq `end if`
  | `if` boolean_expr `then` command_seq `else` command_seq
  `end if`

Concrete Syntax. . .

expr ::= integer_expr | boolean_expr

integer_expr ::= term | integer_expr weak_op term

term ::= element | term strong_op element

element ::= numeral | variable | `(` integer_expr `)` | element

boolean_expr ::= boolean_term | boolean_expr `or` boolean_term

boolean_term ::= boolean_element
  | boolean_term `and` boolean_element

boolean_element ::= `true` | `false` | variable | comparison
  | `not` `(` boolean_expr `)` | `(` boolean_expr `)`

comparison ::= integer_expr relation integer_expr

Concrete Syntax...

variable ::= identifier

identifier ::= letter | identifier letter | identifier digit

relation ::= <= | < | = | > | >= | <>

weak_op ::= + |

strong_op ::= * | /

letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z

numeral ::= digit | digit numeral

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Wren — Example

```
program  binary is
    var n,p : integer;
begin
    read n; p := 2;
    while  p<=n do
        p := 2*p
    end while;
    p := p/2;
    while  p>0 do
       if n>= p then
          write  1; n := np
       else
          write 0
       end if;
       p := p/2
    end while
end
```


Readings and References

- Read Chapter 1, in *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz, http://www.cs.uiowa.edu/~slonnegr/plf/Book.

Acknowledgments

- Some examples are taken from *Introduction to Programming Languages*, by Anthony A. Aaby, http://burks.brighton.ac.uk/burks/pcinfo/progdocs/plbook/semantic.htm
- The Wren lanuage is taken from the book *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz, http://www.cs.uiowa.edu/~slonnegr/plf/Book.