CSc 520

Principles of Programming Languages

53: Semantics — Denotational Semantics

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
Denotational Semantics

- Denotational semantics gives the meaning of a program in terms of mathematical objects: integers, booleans, tuples, and functions.

- The basic idea is to associate a mathematical object with each phrase of the language:
  - The phrase \textit{denotes} the mathematical object.
  - The object is the \textit{denotation} of the phrase.

- Definitions in Denotational Semantics are \textbf{compositional}:
  - The denotation of a language construct is defined in the denotations of its sub-phrases.
Meaning Brackets

- We use the **emphatic** (or **Strachey** or **meaning**) brackets to enclose pieces of abstract syntax, as in

  \[ [p] \].

- If \( p \) is a phrase in the language, we define a mapping \( \text{meaning} \) such that

  \[ \text{meaning} [p] \]

  is a mathematical entity that models the semantics of \( p \).
Meaning Brackets — Examples

- Addition in an imperative language:

\[
\text{evaluate } [E_1 + E_2] \text{ sto } = \text{ compute}(m, \text{plus}, n)
\]

where \( m = \text{evaluate } [E_1] \text{ sto } \)

\( n = \text{evaluate } [E_2] \text{ sto } \)

- The expressions \( 2 \times 4, (5 + 3), 008, 8 \) all denote the same abstract object, 8:

\[
\text{meaning } [2 \times 4] = \text{meaning } [(5 + 3)] = \\
\text{meaning } [008] = \text{meaning } [8] = 8
\]
Denotational Specification

A denotational specification consists of five parts:
1. Syntactic categories
2. Abstract production rules
3. Semantic domains
4. Semantic functions
5. Semantic equations.
Example — A Language of Numerals
Denotational Specification

Syntactic Domains:

\[ N : \text{Numeral} \]
\[ D : \text{Digit} \]

Abstract Production Rules:

\[
\text{Numeral ::= Digit } \mid \text{Numeral Digit}
\]
\[
\text{Digit ::= 0 } \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

Semantic Domains:

\[ \text{Number} = \{0, 1, 2, 3, 4, \ldots\} \]
Denotational Specification...

Semantic Functions:

\[
\begin{align*}
\text{value} : & \quad \text{Numeral} \rightarrow \text{Number} \\
\text{digit} : & \quad \text{Digit} \rightarrow \text{Number}
\end{align*}
\]

Semantic Equations:

\[
\begin{align*}
\text{value} [N \ D] & = 10 \times \text{value} [N] + \text{digit} [D] \\
\text{value} [D] & = \text{digit} [D] \\
\text{digit} [0] & = 0 \\
\vdots & \\
\text{digit} [9] & = 9
\end{align*}
\]
Example

Let’s see how the meaning of the phrase $65$ would be derived:

\[
\]
\[
= 10 \times digit[6] + 5
\]
\[
= 10 \times 6 + 5
\]
\[
= 60 + 5 = 65
\]
Example...

- And the meaning of the phrase $088$:

$$value[088] = 10 \times value[00] + digit[8]$$
$$= 10 \times (10 \times value[0] + digit[0]) + 8$$
$$= 10 \times (10 \times 0 + 0) + 8$$
$$= 10 \times (0 + 0) + 8$$
$$= 8$$

- Note that

The Semantics of Wren
Imperative Languages

- Wren is an imperative language.
- Programs consist of commands (statements).
- Commands alter a store, a global data structure simulating computer memory.
- The program updates the store until the required result is reached.
- The most important command is the assignment statement which modifies the store.
- Basic program control consists of sequencing, selection, and iteration (;, if, while).
Abstract Syntactic Domains

These are the abstract syntactic domains of Wren:

- $P$: Program
- $C$: Command
- $D$: Declaration
- $T$: Type
- $E$: Expression
- $O$: Operator
- $N$: Numeral
- $I$: Identifier
Abstract Syntax of Wrens

Program ::= program identifier is Declaration* begin Command end

Declaration ::= var Identifier : Type

Type ::= integer | boolean

Command ::= command | Command ; Command | variable ::= Expression | skip | read read Identifier | write Expression | while Expression do Command | if Expression then Command | if Expression then Command else Command

Expression ::= Numeral | Identifier | true | false | | Expression Operator Expression | not ( Expression ) | - ( Expression )

Operator ::= <= | < | = | > | >= | <> | + | * | / | and | or
Semantic Domains of Wren

- **SV** (storable values) represents the values that may be placed in the store.

- **EV** (expressible or first-class values) represents the values that expressions can produce.

\[
\text{Integer} = \{ \ldots -2, -1, 0, 1, 2, \ldots \} \\
\text{Boolean} = \{ \text{true}, \text{false} \} \\
\text{EV} = \text{Integer} + \text{Boolean} \\
\text{SV} = \text{Integer} + \text{Boolean} \\
\text{Store} = \text{Identifier} \rightarrow (\text{SV} + \text{undefined})
\]
Semantic Functions of Wren

- The value of an expression depends on the values of variables in the store:

  \[ \text{evaluate} : \text{Expression} \rightarrow (\text{Store} \rightarrow \text{EV}) \]

- Commands (statements) can modify the store:

  \[ \text{execute} : \text{Command} \rightarrow (\text{Store} \rightarrow \text{Store}) \]

- The meaning of a program is its resulting store:

  \[ \text{meaning} : \text{Program} \rightarrow \text{Store} \]

- The meaning of a number is handled elsewhere:

  \[ \text{value} : \text{Numeral} \rightarrow \text{EV} \]
Semantic Equations
The semantics of sequenced commands:

\[
execute [C_1; C_2] = execute [C_2] \circ execute [C_1]
\]

This could also be written as

\[
execute [C_1; C_2] = execute [C_2] (execute [C_1] sto)
\]

**skip** does not affect the store:

\[
execute [skip] sto = sto
\]

The assignment statement evaluates the right-hand-side and produces an updated store:

\[
\]
Commands...

Conditionals:

\[
\text{execute } [[\text{if } E \text{ then } C]] \text{ sto } = \begin{cases} 
\text{if } p \text{ then} \\
\text{execute } [C] \text{ sto} \\
\text{else } \text{ sto}
\end{cases}
\]

where \( p = \text{evaluate } [E] \text{ sto} \)

\[
\text{execute } [[\text{if } E \text{ then } C_1 \text{ else } C_2]] \text{ sto } = \begin{cases} 
\text{if } p \text{ then} \\
\text{execute } [C_1] \text{ sto} \\
\text{else } \text{ execute } [C_2] \text{ sto}
\end{cases}
\]

where \( p = \text{evaluate } [E] \text{ sto} \)
Commands...

Loops:

\[
\text{execute } [\text{while } E \text{ do } C] \text{ sto } = \text{ loop } \\
\text{where } \text{loop sto } = \text{ if } p \text{ then } \\
\text{loop}(\text{execute } [C] \text{ sto}) \text{ else sto } \\
\text{where } p = \text{ evaluate } [E] \text{ sto }
\]

Here we have factored out the looping behavior into a special recursive function \text{loop}. 
Expressions

\[\begin{align*}
\text{evaluate} \ [I] \ sto &= \ \text{if } v = \text{Undefined} \ \text{then error else } v \\
\text{where } v &= \ \text{applySto}(sto, I) \\
\text{evaluate} \ [N] \ sto &= \ \text{value} \ [N] \\
\text{evaluate} \ [\text{true}] \ sto &= \ \text{true} \\
\text{evaluate} \ [\text{false}] \ sto &= \ \text{false} \\
\text{evaluate} \ [E_1 + E_2] \ sto &= \ \text{compute}(m, \text{plus}, n) \\
\text{where } m &= \ \text{evaluate} \ [E_1] \ sto \\
\text{} &= \ \text{evaluate} \ [E_2] \ sto
\end{align*}\]
Expressions...

\[
\text{evaluate } [E_1/E_2] \text{ sto } = \begin{cases} 
\text{if } n = 0 \text{ then error} \\
\text{else compute}(m, \text{div}, n)
\end{cases}
\]

where \( m = \text{evaluate } [E_1] \text{ sto} \)

\( n = \text{evaluate } [E_2] \text{ sto} \)

\[
\text{evaluate } [E_1 < E_2] \text{ sto } = \begin{cases} 
\text{if } n < m \text{ then true else false}
\end{cases}
\]

where \( m = \text{evaluate } [E_1] \text{ sto} \)

\( n = \text{evaluate } [E_2] \text{ sto} \)

\[
\text{evaluate } [E_1 \text{and} E_2] \text{ sto } = \begin{cases} 
\text{if } p \text{ then } q \text{ else false}
\end{cases}
\]

where \( p = \text{evaluate } [E_1] \text{ sto} \)

\( q = \text{evaluate } [E_2] \text{ sto} \)
A Haskell Prototype
Abstract Syntax

type Num = Rational

data SV = IVal Num | BVal Bool | Undefined

type Identifier = String

data Operator = Add | Sub | Mul | Minus | Div | Not | Or | And | Lt | Gt | Eq | Ne | Le | Ge

data Expression = Id String |
    LitInt Num |
    TrueVal |
    FalseVal |
    Unary Operator Expression |
    Binary Expression Operator Expression
Abstract Syntax...

data Program = Prog [Declaration] Command

data Declaration = Var [Identifier] Type

data Type = IntType | BoolType

data Command = Skip |
    Assign String Expression |
    Read String |
    Write Expression |
    IfThen Expression Command |
    IfThenElse Expression Command Command |
    While Expression Command |
    Seq Command Command
Expressions

bcompute :: SV -> Operator -> SV -> SV

bcompute (IVal a) Add (IVal b) = (IVal (a + b))
bcompute (IVal a) Mul (IVal b) = (IVal (a * b))
bcompute (IVal a) Div (IVal b) =
    if b==0 then error "Division by 0"
    else (IVal (toRational(a / b)))
bcompute (IVal a) Sub (IVal b) = (IVal (a - b))
bcompute (BVal a) And (BVal b) = (BVal (a && b))
bcompute (BVal a) Or (BVal b) = (BVal (a || b))
bcompute (IVal a) Lt (IVal b) = (BVal (a < b))
bcompute (IVal a) Gt (IVal b) = (BVal (a > b))
bcompute (IVal a) Le (IVal b) = (BVal (a <= b))
bcompute (IVal a) Ge (IVal b) = (BVal (a >= b))
bcompute (IVal a) Eq (IVal b) = (BVal (a == b))
bcompute (IVal a) Ne (IVal b) = (BVal (not (a == b)))
Expressions...

ucompute :: Operator -> SV -> SV
ucompute Minus (IVal b) = (IVal (- b))
ucompute Not (BVal b) = (BVal (not b))
Expressions...

evaluate :: Expression -> Store -> SV
evaluate (Id id) sto =
    if val == Undefined then val else val
    where val = applySto sto id
evaluate (LitInt n) sto = (IVal n)
evaluate (TrueVal) sto = (BVal True)
evaluate (FalseVal) sto = (BVal False)
evaluate (Unary op r) sto = ucompute op n
    where n = evaluate r sto
evaluate (Binary l op r) sto = bcompute m op n
    where m = evaluate l sto
    n = evaluate r sto
Expressions — Examples

```haskell
> s1
[("b",True), ("a", 5 % 1)]

> evaluate (Binary (LitInt 5) Add (LitInt 6)) s1
11 % 1

> evaluate (Binary (LitInt 5) Add (Id "a")) s1
10 % 1

> evaluate (Binary (Binary (LitInt 6) Mul (LitInt 2)) Add 17 % 1
```
Commands

execute :: Command -> Store -> Store
execute (Skip) sto = sto
execute (Assign id e) sto = updateSto sto id (evaluate e sto)
execute (Seq c1 c2) sto = execute c2 (execute c1 sto)
execute (IfThen b c) sto =
    if (evaluate b sto) == (BVal True) then
        execute c sto
    else sto
execute (IfThenElse b c1 c2) sto =
    if (evaluate b sto) == (BVal True) then
        execute c1 sto
    else execute c2 sto
execute (While b c) sto = loop sto
    where loop sto = if (evaluate b sto) == (BVal True) then
        (loop (execute c sto))
    else sto
> s1
[('b', True), ('a', 5 % 1)]

> execute (Assign "a" (LitInt 9)) s1
[('a', 9 % 1), ('b', True)]

> execute (IfThen (Unary Not (Id "b")) (Assign "a" (LitInt 9))) s1
[('b', True), ('a', 5 % 1)]

> execute (While (Binary (Id "a") Lt (LitInt 10)) (Assign "a" (Binary (Id "a") Add (LitInt 1)))) s1
[('a', 10 % 1), ('b', True)]
type Store = [(Identifier, SV)]

emptySto :: Store
emptySto = []

updateSto :: Store -> Identifier -> SV -> Store
updateSto env id val =
  (id, val) : (filter (\ (x, _) -> not(id==x)) env)

applySto :: Store -> Identifier -> SV
applySto env id =
  snd (foldl (\ r c -> if id==(fst r) then r else c) ("", Undefined) env)
s1 = updateSto (updateSto emptySto "a" (IVal 5)) "b" (BVal True)

> s1
[("b",True),("a",5 % 1)]

> applySto s1 "a"
5 % 1

> applySto s1 "b"
True

> applySto emptySto "a"
Undefined
Readings and References

Read pp. 271–277, 285–310, in Chapter 9 of *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz,

Acknowledgments

Much of the material in this lecture on Denotational Semantics is taken from the book *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz,