CSc 520

Principles of Programming Languages

53: Semantics — Denotational Semantics

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

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 denotes the mathematical object.
  - The object is the denotation of the phrase.
- Definitions in Denotational Semantics are 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 $meaning$ such that

$$meaning\ [\![p]\!]$$

  is a mathematical entity that models the semantics of p.

Meaning Brackets — Examples

- Addition in an imperative language:

$$\begin{aligned} evaluate\ [\![E_1 + E_2]\!]\ \mathrm{sto} &= \mathtt{compute}(m, \mathtt{plus}, n) \\ \text{where } m &= evaluate\ [\![E_1]\!]\ \mathrm{sto} \\ n &= evaluate\ [\![E_2]\!]\ \mathrm{sto} \end{aligned}$$

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

$$\begin{aligned} meaning\ [\![2*4]\!] &= meaning\ [\![(5+3)]\!] = \\ meaning\ [\![008]\!] &= meaning\ [\![8]\!] = 8 \end{aligned}$$

Denotational Specification

- A denotational specification consists of five parts:
  1. Syntactic categores
  2. Abstract production rules
  3. Semantic domains
  4. Semantic functions
  5. Semantic equations.

- **Example — A Language of Numerals**

Denotational Specification

Syntactic Domains:

- N : Numeral
- D : Digit

Abstract Production Rules:

Numeral ::= Digit | Numeral Digit

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

Semantic Domains:

- Number = $\{0, 1, 2, 3, 4, \ldots\}$

Denotational Specification. . .

Semantic Functions:

$$\begin{aligned} \mathit{value} &: \text{Numeral} \rightarrow \text{Number} \\ \mathit{digit} &: \text{Digit} \rightarrow \text{Number} \end{aligned}$$

Semantic Equations:

$$\begin{aligned} \mathit{value}\,[\![N\ D]\!] &= 10 * \mathit{value}\,[\![N]\!] + \mathit{digit}\,[\![D]\!] \\ \mathit{value}\,[\![D]\!] &= \mathit{digit}\,[\![D]\!] \\ \mathit{digit}\,[\![\underline{\mathtt{0}}]\!] &= 0 \\ &\vdots \\ \mathit{digit}\,[\![\underline{\mathtt{9}}]\!] &= 9 \end{aligned}$$

Example

- Let's see how the meaning of the phrase 65 would be derived:

$$\begin{aligned} value\,[\![\texttt{65}]\!] &= 10 * value\,[\![\texttt{6}]\!] + digit\,[\![\texttt{5}]\!] \\ &= 10 * digit\,[\![\texttt{6}]\!] + 5 \\ &= 10 * 6 + 5 \\ &= 60 + 5 = 65 \end{aligned}$$

Example...

- And the meaning of the phrase 088:

$$\begin{aligned} value\,[\![\texttt{088}]\!] &= 10 * value\,[\![\texttt{00}]\!] + digit\,[\![\texttt{8}]\!] \\ &= 10 * (10 * value\,[\![\texttt{0}]\!] + digit\,[\![\texttt{0}]\!]) + 8 \\ &= 10 * (10 * digit\,[\![\texttt{0}]\!] + 0) + 8 \\ &= 10 * (10 * 0 + 0) + 8 \\ &= 8 \end{aligned}$$

- Note that

$$value\,[\![\texttt{088}]\!] = digit\,[\![\texttt{8}]\!] = value\,[\![\texttt{8}]\!] = 8$$

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` identifi er `is` Declaration* `begin` Command `end`

Declaration ::= `var` Identifi er `:` 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 | Identifi er| `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.

$$\begin{aligned} \text{Integer} &= \{\ldots -2, -1, 0, 1, 2, \ldots\} \\ \text{Boolean} &= \{\mathbf{true}, \mathbf{false}\} \\ \mathbf{EV} &= \text{Integer} + \text{Boolean} \\ \mathbf{SV} &= \text{Integer} + \text{Boolean} \\ \text{Store} &= \text{Identifier} \to (\mathbf{SV} + \mathit{undefined}) \end{aligned}$$

Semantic Functions of Wren

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

$$\mathit{evaluate} \ : \ \text{Expression} \to (\text{Store} \to \mathbf{EV})$$

- Commands (statements) can modify the store:

$$\mathit{execute} \ : \ \text{Command} \to (\text{Store} \to \text{Store})$$

- The meaning of a program is its resulting store:

$$\mathit{meaning} \ : \ \text{Program} \to \text{Store}$$

- The meaning of a number is handled elsewhere:

$$\mathit{value} \ : \ \text{Numeral} \to \mathbf{EV}$$

Semantic Equations

Commands

- The semantics of sequenced commands:

$$execute\ [\![C_1; C2]\!] = execute\ [\![C_2]\!] \circ\ execute\ [\![C1]\!]$$

  This could also be written as

$$execute\ [\![C_1; C2]\!] = execute\ [\![C_2]\!]\ (execute\ [\![C1]\!]\ \text{sto})$$

- $\mathbf{skip}$ does not affect the store:

$$execute\ [\![\mathbf{skip}]\!]\ \text{sto} = \text{sto}$$

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

$$execute\ [\![I := E]\!]\ \text{sto} = updateSto(\text{sto}, I, (evaluate\ [\![E]\!]\ \text{sto}))$$

Commands...

- Conditionals:

$$\begin{aligned} execute\ [\![\mathbf{if}\ E\ \mathbf{then}\ C]\!]\ \text{sto} &= \mathbf{if}\ p\ \mathbf{then} \\ &\quad execute\ [\![C]\!]\ \text{sto} \\ &\quad \mathbf{else}\ \text{sto} \\ \text{where } p &= evaluate\ [\![E]\!]\ \text{sto} \\ execute\ [\![\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]\ \text{sto} &= \mathbf{if}\ p\ \mathbf{then} \\ &\quad execute\ [\![C_1]\!]\ \text{sto} \\ &\quad \mathbf{else}\ execute\ [\![C_2]\!]\ \text{sto} \\ \text{where } p &= evaluate\ [\![E]\!]\ \text{sto} \end{aligned}$$

Commands...

- Loops:

$$\begin{aligned} execute\ [\![\mathbf{while}\ E\ \mathbf{do}\ C]\!]\ \text{sto} &= \texttt{loop} \\ \text{where } \texttt{loop}\ \text{sto} &= \mathbf{if}\ p\ \mathbf{then} \\ &\quad \texttt{loop}(execute\ [\![C]\!]\ \text{sto}) \\ &\quad \mathbf{else}\ \text{sto} \\ \text{where } p &= evaluate\ [\![E]\!]\ \text{sto} \end{aligned}$$

- Here we have factored out the looping behavior into a special recursive function `loop`.

Expressions

$$
\begin{aligned}
\mathit{evaluate}\,[\![I]\!]\,\mathrm{sto} &= \textbf{if } v = \texttt{Undefined} \textbf{ then } \texttt{error} \textbf{ else } v\\
\text{where } v &= \mathit{applySto}(\mathrm{sto}, I)\\
\mathit{evaluate}\,[\![N]\!]\,\mathrm{sto} &= \texttt{value}\,[\![N]\!]\\
\mathit{evaluate}\,[\![\textbf{true}]\!]\,\mathrm{sto} &= \textbf{true}\\
\mathit{evaluate}\,[\![\textbf{false}]\!]\,\mathrm{sto} &= \textbf{false}\\
\mathit{evaluate}\,[\![E_1 + E_2]\!]\,\mathrm{sto} &= \texttt{compute}(m, \texttt{plus}, n)\\
\text{where } m &= \mathit{evaluate}\,[\![E_1]\!]\ \mathrm{sto}\\
n &= \mathit{evaluate}\,[\![E_2]\!]\ \mathrm{sto}
\end{aligned}
$$

Expressions...

$$
\begin{aligned}
\mathit{evaluate}\,[\![E_1 / E_2]\!]\,\mathrm{sto} &= \textbf{if } n = 0 \textbf{ then } \texttt{error}\\
& \quad \textbf{else } \texttt{compute}(m, \texttt{div}, n)\\
\text{where } m &= \mathit{evaluate}\,[\![E_1]\!]\ \mathrm{sto}\\
n &= \mathit{evaluate}\,[\![E_2]\!]\ \mathrm{sto}\\
\mathit{evaluate}\,[\![E_1 < E_2]\!]\,\mathrm{sto} &= \textbf{if } n < m \textbf{ then true else false}\\
\text{where } m &= \mathit{evaluate}\,[\![E_1]\!]\ \mathrm{sto}\\
n &= \mathit{evaluate}\,[\![E_2]\!]\ \mathrm{sto}\\
\mathit{evaluate}\,[\![E_1 \textbf{and} E_2]\!]\,\mathrm{sto} &= \textbf{if } p \textbf{ then } q \textbf{ else false}\\
\text{where } p &= \mathit{evaluate}\,[\![E_1]\!]\ \mathrm{sto}\\
q &= \mathit{evaluate}\,[\![E_2]\!]\ \mathrm{sto}
\end{aligned}
$$

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

```
> 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 s
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) the
                     (loop (execute c sto)) else sto
```


Commands — Examples

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

> execute (While (Binary (Id "a") Lt (LitInt 10)) (Assign
[("a",10 % 1),("b",True)]
```


Store

```
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) ("",
```

Store — Examples

```
s1 = updateSto (updateSto emptySto "a" (IVal 5)) "b" (BVal

> 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, `http://www.cs.uiowa.edu/~slonnegr/plf/Book`.

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, `http://www.cs.uiowa.edu/~slonnegr/plf/Book`.