Semantics
July 12, 2017
Outline

What is semantics?

Denotational semantics

Semantics of naming
Why do we need semantics?

Understand what program constructs do
Judge the correctness of a program (compare the expected with observed behaviors)
Prove properties about languages
Compare languages
Design languages
Specification for implementation
Recall aspects of a language

- **syntax**: the structure of its program
- **semantics**: the meaning of its programs

What is the meaning of a program?
How to define the meaning of a program

Formal specifications

- **denotational semantics**: relates terms directly to values
- **operational semantics**: describes how to evaluate a term
- **axiomatic semantics**: describes the effects of evaluating a term

Informal/non-specifications

- **reference implementation**: execute/compile program in some implementation
- **community/designer intuition**: how people “think” a program should behave
Advantages of a formal semantics

A formal semantics...

• is simpler than an implementation, more precise than intuition
  • can answer: is this implementation correct?
• supports definition of analyses and transformations
  • prove properties about the language
  • prove properties about programs
• promotes better language design
  • better understand impact of design decisions
  • apply semantic insights to improve elegance (simplicity + power)
Outline

What is semantics?

Denotational semantics

Semantics of naming
Denotational semantics

A denotational semantics relates each term to a denotation, an abstract syntax tree to a value in a semantic domain.

**Semantic function**

\[ [\cdot] : \text{abstract syntax} \rightarrow \text{semantics domain} \]

**Semantic function in Haskell**

`sem :: Term \rightarrow Value`
Semantics domains

**Semantics domain**: captures the set of possible meanings of a program/term

What is a meaning? ... it depends on the language!

### Example semantic domains

<table>
<thead>
<tr>
<th>Language</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean expression</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Arithmetic expression</td>
<td>Integer</td>
</tr>
<tr>
<td>Imperative</td>
<td>State Transformation</td>
</tr>
<tr>
<td>SQL query</td>
<td>Set of relations</td>
</tr>
<tr>
<td>MiniLogo program</td>
<td>Drawing</td>
</tr>
</tbody>
</table>
Defining a language with denotational semantics

1. Define the **abstract syntax**, \( T \)
   
   *the set of abstract syntax trees*

2. Identify or define the **semantics domain**, \( V \)
   
   *the representation of semantic values*

3. Define the **semantic function**, \( [\cdot] : T \rightarrow V \)
   
   *the mapping from ASTs to semantic values*

---

**Haskell encoding**

```haskell
data Term = …
type Value = …
sem :: Term -> Value
```
Example: simple arithmetic expression language

1. Define abstract syntax
   data Expr = Add Expr Expr
   | Mul Expr Expr
   | Neg Expr
   | Lit Int

2. Identify semantic domain
   Let’s just use Int.

3. Define semantic function
   sem :: Expr -> Int
   sem (Add l r) = sem l + sem r
   sem (Mul l r) = sem l * sem r
   sem (Neg e)   = negate (sem e)
   sem (Lit n)   = n

ExprSem.hs
Exercise: simple arithmetic expression language

Extend the expression language with \texttt{sub} and \texttt{div} operations (abstract syntax and semantics)

\textbf{Abstract syntax}
\begin{verbatim}
data Expr = Add Expr Expr |
           Mul Expr Expr |
           Neg Expr |
           Lit Int
\end{verbatim}

\textbf{Semantic function}
\begin{verbatim}
sem :: Expr -> Int
sem (Add l r) = sem l + sem r
sem (Mul l r) = sem l * sem r
sem (Neg e)   = negate (sem e)
sem (Lit n)   = n
\end{verbatim}

\texttt{ExprSem.hs}
Exercise: simple arithmetic expression language

Abstract syntax

```haskell
data Expr = Add Expr Expr 
  | Mul Expr Expr 
  | Neg Expr 
  | Lit Int 
  | Sub Expr Expr 
  | Div Expr Expr 
```

Semantic function

```haskell
sem :: Expr -> Int
sem (Add l r) = sem l + sem r
sem (Mul l r) = sem l * sem r
sem (Neg e)   = negate (sem e)
sem (Lit n)   = n
sem (Sub l r) = sem l - sem r
sem (Div l r) = sem l `div` sem r
```
Example: simple Boolean expression language

1. Define abstract syntax
   data BExpr = LitB Bool
   | And BExpr BExpr
   | Or BExpr BExpr
   | Not BExpr

2. Identify semantic domain
   Let’s just use `Bool`.

3. Define semantic function
   sem :: BExpr -> Bool
   sem (LitB b)   = b
   sem (And b b') = sem b && sem b'
   sem (Or b b')  = sem b || sem b'
   sem (Not b)    = not (sem b)
Exercise: simple Boolean expression language

Define a Haskell function to apply DeMorgan’s laws to Boolean expressions (i.e. a function to transform \( \text{not} (x \text{ and } y) \) into \( (\text{not} x) \text{ or } (\text{not} y) \) and accordingly \( \text{not} (x \text{ or } y) \) )

**Abstract syntax**

```haskell
data BExpr = LitB Bool 
  | And BExpr BExpr 
  | Or BExpr BExpr 
  | Neg BExpr
```

**Semantic function**

```haskell
sem :: BExpr -> Bool
sem (LitB b)   = b
sem (And b b') = sem b && sem b'
sem (Or b b')  = sem b || sem b'
sem (Neg b)    = not (sem b)
```

```haskell
deMorgan :: BExpr -> BExpr
deMorgan (Not (And (LitB x) (LitB y))) = Or (Not (LitB x)) (Not (LitB y))
deMorgan (Not (Or (LitB x) (LitB y)))  = And (Not (LitB x)) (Not (LitB y))
```
Example: move language

A language describing movements on a 2D plane

- a **step** is an $n$-unit horizontal or vertical shift
- a **move** is a sequence of **steps**

Abstract syntax

```haskell
data Dir = N | S | E | W
data Step = Go Dir Int
type Move = [Step]
```

MoveLang> [Go N 3, Go E 4, Go S 1]
Semantics of move language

1. Define abstract syntax
   data Dir = N | S | E | W
   data Step = Go Dir Int
   type Move = [Step]

2. Identify semantic domain
   type Pos = (Int, Int)

3. Define semantic function
   sem :: Move -> Pos
   sem ss = foldr step (0,0) ss

Define semantics of Step (helper)
   step :: Step -> Pos -> Pos
   step (Go N k) (x,y) = (x,y + k)
   step (Go S k) (x,y) = (x,y - k)
   step (Go E k) (x,y) = (x + k,y)
   step (Go W k) (x,y) = (x - k,y)
Alternative semantics

Often multiple **interpretations** (semantics) of the same language

**Distance traveled**

type Dist = Int

dist :: Move -> Dist
dist = sum . move distS
where distS (Go _ k) = k

**Example: Recipe language**

- Library — estimate time and difficulty
- Market — extract ingredients to buy
- Kitchen — execute to make the dish

**Combined trip information**

trip :: Move -> (Dist,Pos)
trip m = (dist m,sem m)
Picking the right semantic domain

Simple semantics domains can be combined in two ways:

- **products**: contains a value from both domains
  - e.g. combined trip information for move language
  - use Haskell (a,b) or define a new data type
- **sum**: contains a value from one domain or the other
  - e.g. IntBool language can evaluate to Int or Bool
  - use Haskell Either a b or define a new data type

Can errors occur?

- use Haskell Maybe a or define a new data type

Does the language manipulate the state or use naming?

- use a function type
Exercise: expression language with two types

Extend the IntBool language by a **cond** operation (abstract syntax and semantics)

**Abstract syntax**

```haskell
data Expr = Lit Int
  | Add Expr Expr
  | Equ Expr Expr
  | Not Expr

data Val = I Int
  | B Bool
  | TypeError
```

**Semantic function**

```haskell
sem :: Expr -> Val
sem (Lit n) = I n
sem (Add l r) = case (sem l, sem r) of
  (I i, I j) -> I (i + j)
_          -> TypeError
sem (Equ l r) = case (sem l, sem r) of
  (I i, I j) -> B (i == j)
  (B a, B b) -> B (a == b)
_          -> TypeError
sem (Not e) = case sem e of
  B b -> B (not b)
_    -> TypeError
```
Exercise: expression language with two types

Abstract syntax

```haskell
data Expr = Lit Int
         | Add Expr Expr
         | Equ Expr Expr
         | Not Expr
         | Cond Expr Expr Expr

data Val = I Int
         | B Bool
         | TypeError
```

Semantic function

```haskell
sem :: Expr -> Val
sem (Lit n)       = I n
sem (Add l r)     = case (sem l, sem r) of
                    (I i, I j) -> I (i + j)
                    _          -> TypeError
sem (Equ l r)     = case (sem l, sem r) of
                    (I i, I j) -> B (i == j)
                    (B a, B b) -> B (a == b)
                    _          -> TypeError
sem (Not e)       = case sem e of
                    B b -> B (not b)
                    _   -> TypeError
sem (Cond f t e)  = case sem f of
                    B True  -> sem t
                    B False -> sem e
```
Outline

What is semantics?

Denotational semantics

Semantics of naming
What is naming?

Most languages provide a way to **name** and **reuse** stuff

**Naming concepts**
- **declaration** introduce a new name
- **binding** associate a name with a thing
- **reference** use the name to stand in for the bound thing

**C/Java variables**
```
int x, int y;
x = slow(42)
y = x + x + x;
```

**In Haskell:**
```
Local variables
let x = slow 42
   in x + x + x

Type names
type Radius = Float
data Shape = Circle Radius

Function parameters
area r = pi * r * r
```
Semantics of naming

Environment: a mapping associating names with things

\[
\text{type Env} = [(\text{Name}, \text{Thing})]
\]

Naming concepts
- declaration: **add** a new name to the environment
- binding: **set** the thing associated with a name
- reference: **get** the thing associated with a name

Example semantics domains for expressions with...

- **immutable** vars (Haskell): \( \text{Env} \rightarrow \text{Val} \)
- **mutable** vars (C/Java/Python): \( \text{Env} \rightarrow (\text{Env}, \text{Val}) \)
Example: shape language

A language for constructing bitmap images, where an image is either a pixel, or a vertical or horizontal composition of images.
Example: shape language (abstract syntax)

Abstract syntax

data Shape = X
    | TD Shape Shape
    | LR Shape Shape

LR (TD X X) X
TD X (LR X X)
TD (LR X) X
Example: shape language (semantic domain)

Abstract syntax

data Shape = X
|   TD Shape Shape
|   LR Shape Shape

Semantic domain
Shape -> Image

LR (TD X X) X

semantics

[(1,1), (1,2), (2,1)]

Denotational semantics
Example: shape language (semantic domain)

Abstract syntax
\[\text{data Shape} = \text{X} \mid \text{TD Shape Shape} \mid \text{LR Shape Shape}\]

Define semantic function
\[\text{sem :: Shape} \rightarrow \text{Image}\]
\[\text{sem X} = [(1,1)]\]
\[\text{sem (TD s1 s2)} = \text{adjustY ht p1 ++ p2}\]
\[\text{where p1 = sem s1}\]
\[\text{p2 = sem s2}\]
\[\text{ht = maxY p2}\]

Identify semantic domain
\[\text{Shape} \rightarrow \text{Image}\]

MaxY helper function
\[\text{maxY :: [(Int,Int)]} \rightarrow \text{Int}\]
\[\text{maxY p} = \text{maximum (map snd p)}\]

AdjustY helper function
\[\text{adjustY :: Int} \rightarrow \text{[(Int,Int)]} \rightarrow \text{[(Int,Int)]}\]
\[\text{adjustY ht p} = [(x,y + ht) | (x,y) \leftarrow p]\]
Exercise: shape language

Define functions for \( \text{sem} (\text{LR } s1 \ s2) \), \( \text{maxX} \), and \( \text{adjustX} \)

**Semantic function**
\[
\text{sem} :: \text{Shape} \rightarrow \text{Image} \\
\text{sem} (\text{TD } s1 \ s2) = \text{adjustY} \ ht \ p1 \ ++ \ p2 \\
\text{where} \ p1 = \text{sem} \ s1 \\
\text{p2} = \text{sem} \ s2 \\
\text{ht} = \text{maxY} \ p2
\]

**MaxY helper function**
\[
\text{maxY} :: [(\text{Int,Int})] \rightarrow \text{Int} \\
\text{maxY} \ p = \text{maximum} \ (\text{map} \ \text{snd} \ p)
\]

**AdjustY helper function**
\[
\text{adjustY} :: \text{Int} \rightarrow [(\text{Int,Int})] \rightarrow [(\text{Int,Int})] \\
\text{adjustY} \ ht \ p = [(x,y + ht) \mid (x,y) \leftarrow p]
\]