Semantics
Outline

- What is semantics?
- Denotational semantics
- Semantics of naming
What is the meaning of a program?

Recall: aspects of a language

- **syntax**: the structure of its programs
- **semantics**: the meaning of its programs
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 the 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 the language design (e.g. compositionality)
Outline

What is semantics?

Denotational semantics

Semantics of naming
A denotational semantics relates each term to a denotation:

- an abstract syntax tree
- a value in some semantic domain

Valuation function:

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

Valuation function in Haskell:

\[ \text{sem :: Term -> Value} \]
**Semantic 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 expressions</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Arithmetic expressions</td>
<td>Integer</td>
</tr>
<tr>
<td>Imperative language</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

Example encoding in Haskell:

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

   \[
   \text{data Term} = \ldots
   \]

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

   \[
   \text{type Value} = \ldots
   \]

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

   \[
   \text{sem :: Term \rightarrow Value}
   \]
Example: simple arithmetic expressions

1. Define abstract syntax
   
   ```haskell
   data Exp = Add Exp Exp
             | Mul Exp Exp
             | Neg Exp
             | Lit Int
   ```

2. Identify semantic domain
   
   Use the set of all integers, \textbf{Int}

3. Define the valuation function
   
   ```haskell
   sem :: Exp -> 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
   ```
Desirable properties of a denotational semantics

**Compositionality**: a program’s denotation is built from the denotations of its parts
- supports modular reasoning, extensibility
- supports proof by structural induction

**Completeness**: every value in the semantics domain is denoted by some program
- ensures that semantics domain and language align
- if not, language has expressiveness gaps, or semantics domain is too general

**Soundness**: two programs are “equivalent” iff they have the same denotation
- equivalence: have the same observable effects
- ensures that the denotational semantics is correct
More on compositionality

**Compositionality**: a program’s denotation is built from the denotations of its parts

an AST \[ \rightarrow \] sub-ASTs

Example: What is the meaning of \( \text{op } e_1 e_2 e_3 \)?

1. Determine the meaning of \( e_1, e_2, e_3 \)
2. Combine these submeanings in some way specific to \( \text{op} \)

Implications:

- The valuation function is probably **recursive**
- We need different valuation functions for **each syntactic category** (type of AST)
Example: move language

A language describing movements on a 2D plane

- a **step** is an \(n\)-unit horizontal or vertical movement
- a **move** is described by a sequence of steps

**Abstract syntax**

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

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

1. Abstract syntax

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

2. Identify semantic domain

```haskell
type Pos = (Int,Int)
Domain: Pos -> Pos
```

3. Valuation function (Step)

```haskell
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)
```

3. Valuation function (Move)

```haskell
move :: Move -> Pos -> Pos
move [] = \p -> p
move (s:m) = move m . step s
```
Often multiple interpretations (semantics) of the same language

Example: Database schema
One declarative spec, used to:
- initialize the database
- generate APIs
- validate queries
- normalize layout
- ...

Distance traveled

```haskell```

type Dist = Int

dstep :: Step -> Int
dstep (Go _ k) = k

dmove :: Move -> Int
dmove [] = 0
dmove (s:m) = dstep s + dmove m
```

Combined trip information

```haskell```

trip :: Move -> Pos -> (Dist, Pos)
trip m = \p -> (dmove m, move m p)
Picking the right semantic domain

Simple semantics domains can be combined in two ways:

- **product**: 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 state or use naming?
- use a **function type**
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 for the bound thing

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

**In Haskell:**
- **Local variables**
  ```haskell```
  ```
  let x = slow 42
  in x + x + x
  ```
- **Type names**
  ```haskell```
  ```
  type Radius = Float
  data Shape = Circle Radius
  ```
- **Function parameters**
  ```haskell```
  ```
  area r = pi * r * r
  ```
**Environment**: a map 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 semantic domains for expressions with ...**

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

We’ll come back to mutable variables in unit on **scope**