3 Semantics

How goes the obedience training?

Great! Watch this.

Heel.

Next week we start on semantics.
3 Semantics

en.wikipedia.org/wiki/Blissymbols

Semantics

Syntax

"I just invented it...I call it 'O'."
3 Semantics

Why semantics?
What is semantics?
Semantics of simple expression languages
Elements of semantic definitions
Examples: Shape & Move languages
Advanced semantic domains
Translating Haskell into denotational semantics
Haskell as a metalanguage
Why Semantics?

If this statement is true, then God exists

If this statement is true, then God does not exist

If this statement is true, then Facebook protects your privacy

Curry’s Paradox


See also: John Allen Paulos: Irreligion, Hill and Wang 2008
Why Semantics?

Recursion without a base case

\[ S = \text{If } S \text{ is true, then God exists} \]

\[ S = \text{If } S \text{ is true, then God does not exist} \]

\[ S = \text{If } S \text{ is true, then Facebook protects your privacy} \]
Why Semantics?

Access to non-local variables

```c
{  
    int x=2;  
    int f(int y) {return y+x;};  
    {  
        int x=4;  
        printf("%d", f(3));  
    }  
}
```

Output? 5
Why Semantics?

Swap the values of two variables

Effect?

```
{ int x=1;
 int y=8;
 y = x;
 x = y;
}
```

Effect?  
**y:** 1  
**x:** 1

```
{ int x=1;
 int y=8;
 int z;
 z = y;
 y = x;
 x = z;
}
```

Effect?  
**y:** 1  
**x:** 8

```
{ int x=1;
 int y=8;
 y = x + 0*(x = y);
}
```

Effect?  
**y:** 1  
**x:** 8
Why Semantics?

- Understand what program constructs do
- Judge the correctness of a program
  (compare expected with observed behavior)
- Prove properties about languages
- Compare languages
- Design languages
- Specification for implementations

Syntax: Form of programs
Semantics: Meaning of programs
The Meaning of Programs

What is the meaning of a program?

It depends on the language!

<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>Logo</td>
<td>Picture</td>
</tr>
</tbody>
</table>

Denotational Semantics of a language:
Transformation of representation
(abstract syntax → semantic domain)
Simple Examples

BoolSyn.hs
BoolSem.hs

ExprSyn.hs
ExprSem.hs
Exercises

(1) Extend the boolean expression language by an and operation (abstract syntax and semantics)

(2) Extend the arithmetic expressions by multiplication and division (abstract syntax and semantics)

(3) Define a Haskell function to apply DeMorgan’s laws to boolean expression, i.e., a function to transform any expression not (x and y) into (not x) or (not y) (and accordingly for not (x or y))
Defining Semantics in 3 Steps

(1) Define the abstract syntax $S$, i.e. set of syntax trees

(2) Define the semantic domain $D$, i.e. the representation of semantic values

(3) Define the semantic function / valuation $⟦⋅⟧ : S \rightarrow D$ that maps trees to semantic values

Example Language
“Arithmetic expressions”

$S$: Expr
$D$: Int
$⟦⋅⟧$: $\text{sem} :: \text{Expr} \rightarrow \text{Int}$
Language Definitions

Definition Profile

Language

Aspect

Metalanguage

Definition
Example Expression Language

Syntax

```
data Expr = N Int |
             Plus Expr Expr |
             Neg Expr
```

Grammar

```
Expr ::= Num | Expr+Expr | -Expr
```

Haskell

```
sem :: Expr → Int
sem (N i)       = i
sem (Plus e e') = sem e + sem e'
sem (Neg e)     = -(sem e)
```

Math

```
[·] : Expr → Int
[n]     = n
[e+e']  = [e] + [e']
[-e]    = -[e]
```
Related: Expressions vs. Values

- **tail [1 .. 4]**
  - Semantic: \([2, 3, 4]\) or \(2:3:4:[]\)
- **3 + 5**
  - Semantic: 8
  - **3*5 and 5+5+5** have different sense (meaning) but the same referent

=Gottlob Frege: Sense vs. Reference
Example: Shape Language

A language for constructing bitmap images: an image is either a pixel or a vertical or horizontal composition of images.

Operation \( \text{TD } s_1 \ s_2 \) puts \( s_1 \) on top of \( s_2 \)

Operation \( \text{LR } s_1 \ s_2 \) puts \( s_1 \) left next to \( s_2 \)
Abstract Syntax

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

**Example:**

LR (TD X X) X
TD X (LR X X)
TD (LR X X) X

LR aligns at bottom
TD aligns at left

... part of semantics
How to represent a bitmap image?

```haskell
data Shape = X |
            TD Shape Shape |
            LR Shape Shape

type Image = Array (Int,Int) Bool

LR (TD X X) X

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

Drawback: size is fixed, operations require complicated bit shifting
Semantic Function (I)

**Approach:** Translate individual shapes separately into *pixel lists* and then compose pixel lists

```haskell
-- data
data Shape = X
  | TD Shape Shape
  | LR Shape Shape

-- type
type Pixel = (Int, Int)

-- type
type Image = [Pixel]

-- semantics
sem :: Shape -> Image
sem X = [(1,1)]
```

Base case: Individual pixel
Semantic Function (2)

How can we compose (horizontally and vertically) two pixel lists images without overlapping?

```
sem (TD s1 s2) = adjustY ht p1 ++ p2
where p1 = sem s1
      p2 = sem s2
      ht = maxY p2

Take bounding boxes and adjust y-coordinates of top shape by height of bottom shape
```
Semantic Function (3)

```haskell
maxY :: [(Int,Int)] -> Int
maxY p = maximum (map snd p)

adjustY :: Int -> [(Int,Int)] -> [(Int,Int)]
adjustY ht p = [(x,y+ht) | (x,y) <- p]

sem (TD s1 s2) = adjustY ht p1 ++ p2
  where p1 = sem s1
        p2 = sem s2
        ht = maxY p2
```
Exercise

(1) Define the functions:

\[
\begin{align*}
\text{sem (LR } s_1 s_2) & \quad \text{maxX} & \quad \text{adjustX} \\
\text{sem (TD } s_1 s_2) & = \text{adjustY } ht \ p_1 ++ p_2 \\
& \quad \text{where } p_1 = \text{sem } s_1 \\
& \quad p_2 = \text{sem } s_2 \\
& \quad ht = \text{maxY } p_2 \\
\text{maxY} & : [(\text{Int,Int})] \to \text{Int} \\
\text{maxY } p & = \text{maximum } (\text{map } \text{snd } p) \\
\text{adjustY} & : \text{Int } \to [(\text{Int,Int})] \to [(\text{Int,Int})] \\
\text{adjustY } ht \ p & = [(x, y+ht) \mid (x, y) \leftarrow p]
\end{align*}
\]
Exercise

(1) Define the functions:

\[
\text{sem (LR } s_1 s_2) \quad \text{maxX} \quad \text{adjustX}
\]

\[
\text{sem (LR } s_1 s_2) = p_1 ++ \text{adjustX } wd \ p_2
\]

\[
\text{where } p_1 = \text{sem } s_1
\]

\[
p_2 = \text{sem } s_2
\]

\[
wd = \text{maxX } p_1
\]

\[
\text{maxX } :: [(\text{Int,Int})] \to \text{Int}
\]

\[
\text{maxX } p = \text{maximum } (\text{map } \text{fst } p)
\]

\[
\text{adjustX } :: \text{Int } \to [(\text{Int,Int})] \to [(\text{Int,Int})]
\]

\[
\text{adjustX } wd \ p = [(x+wd,y) \mid (x,y) \leftarrow p]
\]
Example

Shape.hs
ShapePP.hs
Example: Move Language

A language describing vector-based movements in the 2D plane. A step is an $n$-unit horizontal or vertical move, a move is a sequence of steps.

Go Up 3;
Go Right 4;
Go Down 1
Abstract Syntax

```haskell
data Dir = Lft | Rgt | Up | Dwn

data Step = Go Dir Int

type Move = [Step]
```

Example:

```
[Go Up 3, Go Rgt 4, Go Dwn 1]
```
Exercises

(1) Give a type definition for the data type *Step*

```
data Step = Go Dir Int
```

(2) Define the data type *Move* without using built-in lists

```
type Move = [Step]
```

(3) Write the move `[Go Up 3, Go Rgt 4, Go Dwn 1]` using the representation from (1) and (2)
Semantic Domain

What is the meaning of a move?

The distance traveled, the final position, or both.

data Dir = Lft | Rgt | Up | Dwn

data Step = Go Dir Int

type Move = [Step]

type Pos = (Int, Int)

[Go Up 3, Go Rgt 4, Go Dwn 1] semantics (4,2)
Semantic Function

\[
\text{sem} :: \text{Move} \rightarrow \text{Pos} \\
\text{sem} \ [ ] = (0,0) \\
\text{sem} \ (\text{Go} \ d \ i:ss) = (dx \times i + x, dy \times i + y) \\
\text{where} \ (dx, dy) = \text{vector} \ d \\
(x, y) = \text{sem} \ ss
\]

\[
\text{vector} :: \text{Dir} \rightarrow (\text{Int}, \text{Int}) \\
\text{vector} \ \text{Lft} = (-1, 0) \\
\text{vector} \ \text{Rgt} = (1, 0) \\
\text{vector} \ \text{Up} = (0, 1) \\
\text{vector} \ \text{Dwn} = (0, -1)
\]
Example

Move.hs
Exercises

sem :: Move -> Pos
sem [] = (0,0)
sem (Go d i:ss) = (dx*i+x, dy*i+y)
where (dx,dy) = vector d
       (x,y) = sem ss

(1) Define the semantic function for the move language for the semantic domain

\textbf{type} Dist = Int

(2) Define the semantic function for the move language for the semantic domain

\textbf{type} Trip = (Dist, Pos)
Advanced Semantic Domains

The story so far: Semantic domains were mostly simple types (such as Int or [(Int,Int)])

How can we deal with language features, such as errors, union types, or state?

(1) Errors: Use the Maybe data type
(2) Union types: Use corresponding data types
(3) State: Use function types
If \( T \) is the type representing “regular” values, define the semantic domain as \( \text{Maybe } T \).

```haskell
data Maybe a = Just a | Nothing
```

*regular value*  
*error value*  
*type of regular values*
Example

ExprErr.hs
If \( T_1 \ldots T_k \) are types representing different semantic values for different nonterminals, define the semantic domain as a data type with \( k \) constructors.

```
data T = C1 T1 |
           ... |
           Ck Tk
```
Special Case: Binary Union Domains

If $T_1$ and $T_2$ are types representing different semantic values for different nonterminals, define the semantic domain as a data type with 2 constructors.

Or: Use the `Either` data type.

```haskell
data Either a b = Left a | Right b

data T = C1 T1 | C2 T2
type T = Either T1 T2

data Val = I Int | B Bool
type Val = Either Int Bool
```
Example

Expr2.hs
(1) Extend the semantic domain for the two-type expression language to include errors

```haskell
data Val = I Int
         | B Bool
         | Error
```

(2) Extend the semantic function for the two-type expression language to handle errors
Function Domains

If a language operates on a state that can be represented by a type $T$, define the semantic domain as a function type $T \rightarrow T$.

\[
\text{type } D = T \rightarrow T
\]

\[
\text{sem :: S } \rightarrow \text{ D}
\]

\[
\text{sem :: S } \rightarrow \text{ (T } \rightarrow \text{ T)}
\]

\[
\text{sem :: S } \rightarrow \text{ T } \rightarrow \text{ T}
\]

Semantic function takes state as an additional argument.
Example

RegMachine.hs
(1) Extend the machine language to work on two registers A and B

(2) Define a new semantic domain for the extended language

(3) Define the semantics functions for the extended language
Piazza Question

Semantic Domain
Translating Haskell into Mathematical Denotational Semantics

(1) Replace type definitions by sets (should actually be CPOs)

(2) Replace patterns by grammar productions (and replace nonterminals by variables)

(3) Replace function names by semantic brackets that enclose only syntactic objects

\[
\begin{align*}
\text{sem} &: \text{Expr} \rightarrow \text{Int} \\
\text{sem} (N \, i) &= i \\
\text{sem} (\text{Plus} \, e \, e') &= \text{sem} \, e + \text{sem} \, e' \\
\text{sem} (\text{Neg} \, e) &= -(\text{sem} \, e)
\end{align*}
\]

\[
\begin{align*}
\text{Expr} &::= \text{Num} \mid \text{Expr} + \text{Expr} \mid -\text{Expr} \\
[\cdot] &: \text{Expr} \rightarrow \text{Int} \\
[n] &= n \\
[e+e'] &= [e] + [e'] \\
[-e] &= -[e]
\end{align*}
\]
Haskell as a Mathematical Metalanguage

Math World

Haskell World

= Executable Math World

1. Data Types → Functions → Data Types
2. Syntax
3. Semantics

Grammars (Languages) → Functions (Semantics) → Sets (Semantic domains)

Sets

Functions

Data Types

Semantics

Data Types