3 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?

Modus Ponens

\[ \frac{S \rightarrow G \quad S}{G} \]

If this statement is true, then God exists

If this statement is true, then God does not exist

If this statement is true, then United Airlines is customer friendly


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

Curry’s Paradox

Haskell B. Curry, 1900-1982

United Airlines
Version 2.1.18, 85.8 MB
Apr 10, 2017
Supports new drag and drop feature.
Why Semantics?

Recursion without a base case

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

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

\[ S = \text{If } S \text{ is true, then United Airlines is customer friendly} \]
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

```
{  
    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: 1

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

Effect?  
y: 1  
x: 8

Why?!
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 $\rightarrow$ 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** $⟦⋅⟧ : S → D$ that maps trees to semantic values

**Example Language**

"Arithmetic expressions"

$S$: $\text{Expr}$

$D$: $\text{Int}$

$⟦⋅⟧$: $\text{sem} :: \text{Expr} → \text{Int}$
Language Definitions
Example Expression Language

Syntax

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

Haskell

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

Semantic Domain

```
Expr
```

Semantics

```
Expr
```

Semantic Function

```
Expr
```

Math

```
Expr
```

Syntactic Symbol

```
Expr
```

Semantic Operation

```
Expr
```
Related: Expressions vs. Values

- **tail** [1 .. 4] → [2, 3, 4] or 2:3:4:[]
  - ![False](2 .. 4)

- 3 + 5 → 8
  - ![False](2 + 6)

Gottlob Frege: Meaning vs. Reference

- 3*5 and 5+5+5 have different meaning but the same referent
Example: Shape Language

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

- Operation **TD** \(s_1 \ s_2\) puts \(s_1\) on top of \(s_2\).
- Operation **LR** \(s_1 \ s_2\) puts \(s_1\) left next to \(s_2\).
Abstract Syntax

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

**Example:**

- LR \((TD \text{ X X}) \text{ X}\)
- TD \text{ X (LR X X)}
- TD \((LR \text{ X X}) \text{ X}\)

LR aligns at bottom
TD aligns at left

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

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

LR (TD X X) X
```

```
type Image = Array (Int,Int) Bool

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

Drawback: size is fixed, operations require complicated bit shifting

```
type Pixel = (Int,Int)
type Image = [Pixel]
```
Semantic Function (I)

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

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

type Pixel = (Int,Int)
type Image = [Pixel]

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)

\[
\text{sem (TD } s_1 \ s_2) = \text{adjustY } ht \ p_1 ++ p_2
\]

where
\[
\begin{align*}
\text{p1} &= \text{sem } s_1 \\
\text{p2} &= \text{sem } s_2 \\
\text{ht} &= \text{maxY } p_2
\end{align*}
\]

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

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

(1) Define the functions:

\[
\begin{align*}
\text{sem (LR } s_1 \ s_2) & = \text{p1} ++ \text{adjustX wd p2} \\
\text{where} & \quad \text{p1} = \text{sem } s_1 \\
& \quad \text{p2} = \text{sem } s_2 \\
& \quad \text{wd} = \text{maxX p1}
\end{align*}
\]

\[
\begin{align*}
\text{maxX} & :: [(\text{Int,Int})] \rightarrow \text{Int} \\
\text{maxX } p & = \text{maximum (map } \text{fst} \text{ p)}
\end{align*}
\]

\[
\begin{align*}
\text{adjustX} & :: \text{Int} \rightarrow [(\text{Int,Int})] \rightarrow [(\text{Int,Int})] \\
\text{adjustX } \text{ht} \ p & = [(x, y+\text{ht}) \mid (x, y) \leftarrow p]
\end{align*}
\]

\[
\begin{align*}
\text{sem (TD } s_1 \ s_2) & = \text{adjustY ht p1} ++ \text{p2} \\
\text{where} & \quad \text{p1} = \text{sem } s_1 \\
& \quad \text{p2} = \text{sem } s_2 \\
& \quad \text{ht} = \text{maxY } p_2
\end{align*}
\]

\[
\begin{align*}
\text{maxY} & :: [(\text{Int,Int})] \rightarrow \text{Int} \\
\text{maxY } p & = \text{maximum (map } \text{snd} \text{ p)}
\end{align*}
\]

\[
\begin{align*}
\text{adjustY} & :: \text{Int} \rightarrow [(\text{Int,Int})] \rightarrow [(\text{Int,Int})] \\
\text{adjustY } \text{ht} \ p & = [(x, y+\text{ht}) \mid (x, y) \leftarrow p]
\end{align*}
\]
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

\[ \text{type Step} = (\text{Dir}, \text{Int}) \]

\[ \text{data Step} = \text{Go} \ \text{Dir} \ \text{Int} \]

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

\[ \text{type Move} = \text{One Step} \ |
\text{Seq Step Move} \]

\[ \text{data Move} = \text{One Step} \ |
\text{Seq Step Move} \]

(3) Write the move \([\text{Go Up 3, Go Rgt 4, Go Dwn 1}]\)
using the representation from (1) and (2)

\[ (\text{Up,3}) \ \text{`Seq`} \ (\text{Rgt,4}) \ \text{`Seq`} \ \text{One} \ (\text{Dwn,1}) \]
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]
```

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

```
type Pos = (Int,Int)
```
Semantic Function

```
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
```

```
vector :: Dir -> (Int,Int)
vector Lft = (-1,0)
vector Rgt = (1,0)
vector Up  = (0,1)
vector Dwn = (0,-1)
```

*pattern matching in definitions*
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 $\text{Maybe } T$

data Maybe a = Just a | Nothing
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.

```haskell
data T = C1 T1 |
        ... |
        Ck Tk
```

Semantic domain

Different types of result values
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

data Val = I Int | B Bool

type T =Either T1 T2

type Val = Either Int Bool
```
Example

Expr2.hs
Exercises

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

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

(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 \( T \rightarrow T \).

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

\[
\text{sem} :: S \rightarrow D
= \quad \text{semantic function takes state as an additional argument}
\]

\[
\text{sem} :: S \rightarrow (T \rightarrow T)
= \quad \text{Semantic function}
\]

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

RegMachine.hs
Exercises

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.

RegMachine2.hs

```haskell
data Op = LD Int
        | INC
        | DUP

data Op = LD Reg Int
        | INC Reg
        | DUP Reg

data Reg = A | B

type RegCont = Int

type D = RegCont -> RegCont
```
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

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

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

Math World

Sets
(Semantic domains)

Functions
(Semantics)

Grammars
(Languages)

Data Types
(Haskell World)

Functions

Data Types

2 Syntax
3 Semantics

Semantics

= Executable Math World