3 Semantics

How goes the obedience training?

Great! Watch this.

Heel.

Next week we start on semantics.

©2010 Creators.com
www.JohnHartStudios.com
3 Semantics

en.wikipedia.org/wiki/Blissymbols

"I just invented it...I call it 'Q.'"
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?

Access to non-local variables

```
{  
    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

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

Effect? y: 1
x: 1

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

Effect? y: 1
x: 1

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

Effect? y: 1
x: 8

What?!
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!

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** $⟦⋅⟧ : S → D$ that maps trees to semantic values

**Example Language**

"Arithmetic expressions"

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

Definition Profile

- Language
- Aspect
- Metalanguage

Aspect

Metalanguage
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
Haskell
```

**Semantic Function**

```
[·] : Expr → Int
[n]         = n
[e+e']      = [e] + [e']
[-e]       = -[e]
```

**Semantics**

```
Semantics
Math
```

**Syntactic Symbol**

```
Expr
Syntax
Grammar
```

**Semantic Operation**

```
Expr
Semantics
Math
```

Semantics
Related: Expressions vs. Values

- 3 + 5 → semantics 8

Gottlob Frege: Sense vs. Reference

3*5 and 5+5+5 have different sense (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 $\text{TD } s_1 \ s_2$ 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

```plaintext
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
```
Semantic Domain

How to represent a bitmap image?

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

type Image = Array (Int,Int) Bool

Pixel = (Int,Int)

type Image = [Pixel]

LR (TD X X) X

semantics

[(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 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?

\[
\text{sem} \ (\text{TD} \ s1 \ s2) = \text{adjustY} \ ht \ p1 \ ++ \ p2
\]

where
- \( p1 = \text{sem} \ s1 \)
- \( p2 = \text{sem} \ s2 \)
- \( ht = \text{maxY} \ p2 \)

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

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

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]
Exercise

Define the functions:

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

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

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

\[
\text{p2 = sem } s_2
\]

\[
\text{ht = maxY } p_2
\]

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

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

\[
\text{adjustY } : : \text{Int} \rightarrow [(\text{Int,Int})] \rightarrow [(\text{Int,Int})]
\]

\[
\text{adjustY } h t \ p = [(x,h t\ + y) \mid (x,y) \leftarrow p]
\]
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

```hs
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 \textit{Step}.

\begin{verbatim}
\textbf{data} \textit{Step} = \textit{Go} \textit{Dir} \textit{Int}
\end{verbatim}

(2) Define the data type \textit{Move} without using built-in lists.

\begin{verbatim}
\textbf{type} \textit{Move} = [	extit{Step}]
\end{verbatim}

(3) Write the move \texttt{[Go Up 3, Go Rgt 4, Go Dwn 1]} using the representation from (1) and (2).
What is the meaning of a move?

The distance traveled, the final position, or both.

\[
\text{data} \quad \text{Dir} = \text{Lft} \mid \text{Rgt} \mid \text{Up} \mid \text{Dwn}
\]

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

\[
\text{type} \quad \text{Move} = [\text{Step}]
\]

\[
\text{type} \quad \text{Pos} = (\text{Int}, \text{Int})
\]

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

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

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

Move.hs
Exercises

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

```
type Dist = Int
```

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

```
type Trip = (Dist,Pos)
```

```
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
```
Advanced Semantic Domains

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

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

(1) \textit{Errors: } Use the \texttt{Maybe data type}

(2) \textit{Union types: } Use corresponding \texttt{data types}

(3) \textit{State: } Use \texttt{function types}
Error Domains

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.

```haskell
data T = C1 T1 |
        \ldots |
        Ck Tk
```

**Union Domains**

Semantic domain

Different types of result values
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.

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

data $T = C_1 T_1$

| $C_2 T_2$

type $T = Either T_1 T_2$

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

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

Type \( D = T \rightarrow T \)

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

RegMachine2.hs
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

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

Grammars (Languages) \(\rightarrow\) Functions (Semantics) \(\rightarrow\) Sets (Semantic domains)

Math World

Data Types \(\rightarrow\) Functions \(\rightarrow\) Data Types

Haskell World = Executable Math World