2 Denotational Semantics in Haskell
Representing a DSL in Haskell

Semantic Language

\[ S \oplus \left[ \cdot \right] : S \to D \]

(General-Purpose) PL

Turing-complete computation

Other computation

Non-computation

function

type or data type

data type
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>

Semantics of a language: Transformation of representation (abstract syntax → semantic domain)
### Steps for Defining Language Semantics

1. Define the **abstract syntax** ($S$) of the language.
2. Define the **semantic domain** ($D$), i.e., the representation of semantic values.
3. Define the **semantic function / valuation** that maps sentences into semantic values ($\llbracket \cdot \rrbracket : S \rightarrow D$).

**Example**

‘arithmetic expressions’:

- $S$: $\text{Expr}$
- $D$: $\text{Int}$
- $\llbracket \cdot \rrbracket$: $\text{sem} :: \text{Expr} \rightarrow \text{Int}$
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)

LR aligns at bottom
TD aligns at left

... part of semantics
Semantic Domain

\[\text{data Shape} = \begin{cases} X \\ | \text{TD Shape Shape} \\ | \text{LR Shape Shape} \end{cases}\]

How to represent a bitmap image?

\[\text{type Image} = \text{Array} \ (\text{Int}, \text{Int}) \ \text{Bool}\]

Drawback: size is fixed, operations require complicated bit shifting

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

\[\text{type Image} = [\text{Pixel}]\]

\[\text{LR} \ (\text{TD} \ X \ X) \ X \ \rightarrow \ [(1,1),(1,2),(2,1)]\]

Denotational Semantics in Haskell
Semantic Function (1)

Approach: Translate individual shapes separately into bitmaps and then compose bitmaps

```haskell
data Shape = X | ...
type Pixel = (Int,Int)
type Image = [Pixel]
```

Base case: Individual pixel:

```haskell
sem :: Shape -> Image
sem X = [(1,1)]
```
Semantic Function (2)

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

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

\[
\text{where} \quad p1 = \text{sem} \ s1
\]
\[
\quad p2 = \text{sem} \ s2
\]
\[
\quad ht = \text{maxY} \ p2
\]

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

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

where
\[
\begin{align*}
\text{p1} &= \text{sem } s1 \\
\text{p2} &= \text{sem } s2 \\
\text{ht} &= \text{maxY } p2
\end{align*}
\]

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

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

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

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

(1) Define the functions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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`

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

(3) Write the move `[Go Up 3, Go Rgt 4, Go Dwn 1]` using the representation from (1) and (2)
Denotational Semantics in Haskell

Semantic Domain

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

data Step = Go Dir Int

type Move = [Step]
```

What is the meaning of a move?

The distance traveled, the final position, or both.

```haskell
type Pos = (Int,Int)
```

```
[Go Up 3, Go Rgt 4, Go Dwn 1] -- semantics  →  (4,2)
```
Denotational Semantics in Haskell

Semantic Function

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

\[
\text{vector} :: \text{Dir} \rightarrow (\text{Int},\text{Int}) \\
\text{vector} \; \text{Lft} &= (-1,0) \\
\text{vector} \; \text{Rgt} &= (1,0) \\
\text{vector} \; \text{Up} &= (0,1) \\
\text{vector} \; \text{Dwn} &= (0,-1)
\]
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)
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
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**
Union Domains

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 | \ldots | Ck Tk
```
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 $T \rightarrow T$

```
type D = T -> T
```

```haskell
sem :: S -> D
= sem :: S -> (T -> T)
= sem :: S -> T -> T
```

Semantic function takes state as an additional argument
Translating Haskell into Denotational Semantics

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

(2) Replace *constructor expressions* by *grammar productions*

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

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

\[
\begin{align*}
\llbracket \cdot \rrbracket & : \text{exp} \rightarrow \mathbb{Z} \\
\llbracket i \rrbracket & = i \\
\llbracket e + e' \rrbracket & = \llbracket e \rrbracket + \llbracket e' \rrbracket \\
\llbracket -e \rrbracket & = -\llbracket e \rrbracket
\end{align*}
\]

\[
\text{exp} ::= \text{num} | \text{exp} + \text{exp} | -\text{exp}
\]
Haskell as a Mathematical Metalanguage

Math World

Sets (Semantic domains)

Functions (Semantics)

Grammars (Syntax)

Data Types

Syntax → Semantics

Haskell World

= Math World + executable