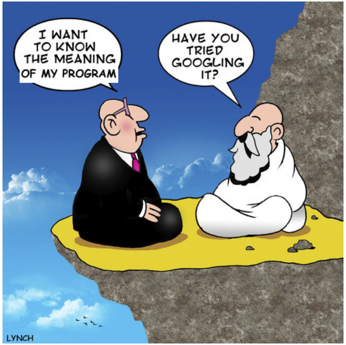
Denotational Semantics

January 22, 2015

What is the meaning of a program?



Recall: syntax vs. semantics

- **syntax**: the structure of its programs
- **semantics**: the meaning of its programs

There are many ways to assign meaning to programs

- **denotational semantics**: relates programs to external objects
- **operational semantics**: describes how programs are evaluated
- **axiomatic semantics**: describes the effects of running a program
- ...

Denotational semantics

A denotational semantics relates each **program** to a **denotation**

an abstract syntax tree

a value in some **semantic domain**

Semantic function (a.k.a. valuation)

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

Semantics domains

Semantics domain: captures the set of possible meanings of a program

what is a meaning? – it depends on the language!

Example semantic domains

Language	Meaning
Boolean expressions	Boolean value
Arithmetic expressions	Integer value
Imperative language	State transformation
Functional language	Mathematical function
SQL	Set of relations
Logo	Picture

Defining a language with denotational semantics

1. Define the **abstract syntax**, S
   the set of abstract syntax trees

2. Define the **semantics domain**, D
   the representation of semantic values

3. Define the **semantic function**, $[\![\cdot]\!] : S \to D$
   the mapping from ASTs to semantic values

Example: simple arithmetic expressions

1. Define abstract syntax

$$
\begin{array}{rcl}
n \in Nat & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \dots \\
e \in Exp & ::= & \mathtt{add}\ e\ e \\
& \mid & \mathtt{mul}\ e\ e \\
& \mid & \mathtt{neg}\ e \\
& \mid & n
\end{array}
$$

2. Define semantic domain

Use the set of all integers, *Int*

3. Define the semantic function

$$\llbracket Exp \rrbracket : Int$$

$$
\begin{aligned}
\llbracket \mathtt{add}\ e_1\ e_2 \rrbracket &= \llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket \\
\llbracket \mathtt{mul}\ e_1\ e_2 \rrbracket &= \llbracket e_1 \rrbracket \times \llbracket e_2 \rrbracket \\
\llbracket \mathtt{neg}\ e \rrbracket &= -\llbracket e \rrbracket \\
\llbracket n \rrbracket &= toInt(n)
\end{aligned}
$$

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 have same denotation iff they are "equivalent"

- equivalence defined in terms of other properties you care about
- ensures that semantics function is correct

Encoding denotational semantics in Coq

1. **abstract syntax**: define a new **data type**, as usual
2. **semantics domain**: identify and/or define a new **type**, as needed
3. **semantics function**: define a **function** from ASTs to semantics domain

Semantics function in Coq

```
Fixpoint sem (e:Exp) : Z :=
  match e with
    | add e1 e2 => sem e1 + sem e2
    | mul e1 e2 => sem e1 * sem e2
    | neg e     => - (sem e)
    | lit n     => Z.of_nat n
  end.
```

In-depth exercise: a language with multiple types

$$
\begin{array}{rcl}
b \in \mathit{Bool} & ::= & \texttt{true} \mid \texttt{false} \\
n \in \mathit{Nat} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
e \in \mathit{Exp} & ::= & \texttt{add}\ e\ e \\
& \mid & \texttt{neg}\ e \\
& \mid & \texttt{equal}\ e\ e \\
& \mid & \texttt{cond}\ e\ e\ e \\
& \mid & n \\
& \mid & b
\end{array}
$$

Design a denotational semantics for *Exp*

1. How should we define our semantics domain?
2. Define a semantics function

You may use either math notation or Coq

- **neg** – negates either an integer or boolean value
- **equal** – compares two values of the same type for equality
- **cond** – equivalent to **if-then-else**

Solution

$$[\![\, Exp \,]\!] \; : \; Int \oplus Bool \oplus \bot$$

$$[\![\texttt{add}\;\; e_1 \;\; e_2]\!] = \begin{cases} [\![e_1]\!] + [\![e_2]\!] & [\![e_1]\!] \in Int, [\![e_2]\!] \in Int \\ \bot & \textit{otherwise} \end{cases}$$

$$[\![\texttt{neg}\;\; e]\!] = \begin{cases} -[\![e]\!] & [\![e]\!] \in Int \\ \neg[\![e]\!] & [\![e]\!] \in Bool \\ \bot & \textit{otherwise} \end{cases}$$

$$[\![\texttt{equal}\;\; e_1 \;\; e_2]\!] = \begin{cases} [\![e_1]\!] =_{Int} [\![e_2]\!] & [\![e_1]\!] \in Int, [\![e_2]\!] \in Int \\ [\![e_1]\!] =_{Bool} [\![e_2]\!] & [\![e_1]\!] \in Bool, [\![e_2]\!] \in Bool \\ \bot & \textit{otherwise} \end{cases}$$

$$[\![\texttt{cond}\;\; e_1 \;\; e_2 \;\; e_3]\!] = \begin{cases} [\![e_2]\!] & [\![e_1]\!] = \texttt{true} \\ [\![e_3]\!] & [\![e_1]\!] = \texttt{false} \\ \bot & \textit{otherwise} \end{cases}$$

$$[\![n]\!] = toInt(n)$$

$$[\![b]\!] = b$$

In-depth exercise: a functional semantic domain

$$
\begin{array}{rcl}
n \in \mathit{Nat} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
e \in \mathit{Exp} & ::= & \mathtt{add}\ e\ e \\
& \mid & \mathtt{neg}\ e \\
& \mid & \mathtt{set}\ e\ e \\
& \mid & \mathtt{get} \\
& \mid & n
\end{array}
$$

Design a denotational semantics for *Exp*

1. How should we define our semantics domain?
2. Define a semantics function

You may use either math notation or Coq

This language simulates a calculator with a single integer memory cell

- **set** e_1 e_2 – sets the memory cell to the result of e_1, then evaluates e_2
- **get** – returns the value of the memory cell

In-depth exercise: a simple stack language

$$
\begin{array}{rcl}
n \in \mathit{Nat} & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \ldots \\
p \in \mathit{Prog} & ::= & c^* \\
c \in \mathit{Cmd} & ::= & \texttt{load}\ n \\
 & \mid & \texttt{dup} \\
 & \mid & \texttt{neg} \\
 & \mid & \texttt{add} \\
 & \mid & \texttt{mul}
\end{array}
$$

Define a denotational semantics for *Cmd*

1. What is a good semantics domain?
2. Define a semantics function

Use the semantics of *Cmd* to define a denotational semantics for *Prog*

In this language, commands manipulate a shared stack, similar to Forth or PostScript

- **`load`** n – pushes the value n onto the stack
- **`dup`** – duplicate the top value on the stack
- **`neg`** – negate the top value on the stack
- **`add`** – pop the top two values off the stack, push their sum
- **`mul`** – pop the top two values off the stack, push their product