The semantic question again

- We are trying to give meaning to a program's source code
- The ideal would be a static object that we can study
- Neither operational nor axiomatic methods result in an object that is easy to study
  - Operational method results in a derivation (a kind of proof, since the rules are logical) which may not terminate if a loop does not
  - Axiomatic methods results in a proof involving potentially complex axioms of arithmetic

The meaning triangle

- Ogden & Richards (1923)
- Meaning is in three parts:
  - The thing itself (a real-world object)
  - A symbolic version of it (words, diagrams, etc.)
  - The concept of the thing (stored in the head of a human reasoner)
The meaning triangle for programming languages

- Symbol = source code (syntax)
- Concept = model of computation (semantics)
- Object = realization of semantics

What is a program?

- It *is* the electron flows in the silicon
- It could be the machine code running on the real machine
- It could be an abstract program running on a virtual machine
- These are too closely tied to real or abstract machines
- Better would be a ‘program’ in a machine-independent language

Denotational programs are functions

- Programs are modeled as mathematical functions operating on domains of primitive values
- Sets can serve to represent most value domains
- Lambda calculus, suitably extended, can be used for functions
The big picture

1. Abstract syntax to represent source code
2. Semantic domains to represent value sets
3. Semantic algebras to represent operations on these sets
4. Valuation functions to map syntax to semantics

- A program will be represented as a static functional object that can be:
  - applied to initial values to give a derivation
  - examined for its properties (termination, equivalence etc.)

Semantic domains

- Primitive: \( \mathbb{N} \) (naturals), \( \mathbb{Z} \) (integers), \( \mathbb{R} \) (reals), \( \mathbb{C} \) (characters), \( \mathbb{B} \) (Booleans)
- Compound: \( A \times B \) (product), \( A + B \) (sum), \( A \rightarrow B \) (function), where \( A \) and \( B \) are any sets, primitive or compound
- Operations:
  - On pairs: \( \text{fst}([x,y] \in A \times B) = x \), \( \text{snd}([x,y] \in A \times B) = y \)
  - On sums: \( \text{inA}(x \in A + B) = A + B \), \( \text{inB}(y \in B) = A + B \)
  - On functions: \( (\lambda x \in A.M)N = M[N/x] \in B \)

Function definitions

- Equational form: functionality and one or more equations defining the relationships between domain and range:
  - e.g.
    - add: \( \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \)
      - \( \text{add}(m,n) = m + n \)
    - lessOne: \( \mathbb{Z} \rightarrow \mathbb{B} \)
      - \( \text{lessOne}(n) = n < 1 \rightarrow \text{true} \odot \text{false} \)

This conditional form is an extension to functional language.
Lambda calculus forms

- add = \( \lambda[m,n].m+n \)
- lessOne = \( \lambda.n.n < 1 \rightarrow \square \) false
- Currying of functions:
  - \( (A \times B) \rightarrow C \) is isomorphic to \( A \rightarrow (B \rightarrow C) \) (i.e. for same input, both functions produce the same output)
- Single argument form: add = \( \lambda.m.\lambda.n.m+n \)

Extension for checking disjoint union tags

- If \( m \in A+B \),
  - cases (m) of
    - \( isA(x) \rightarrow \ldots x \ldots \square \)
    - \( isB(y) \rightarrow \ldots y \ldots \)
  - end
- \( x, y \) are \( m \) with the tag removed; \( isA \) and \( isB \) test the tag on \( m \) and throw it away

Lifted domains

- We need to express the concepts of "undefined", "error", "no result", "non-termination"
- They will all be represented as \( \perp \) (bottom)
- Added to any semantic domain, we "lift" the domain: e.g. \( \mathbb{N} \cup \{ \perp \} \) is written \( \mathbb{N}_\perp \)
**Strict and non-strict functions**

- A function that has bottom in domain and range is a lifted function:
- e.g. addTwo: \( \mathbb{N} \rightarrow \mathbb{N} \)
  
  \[
  \text{addTwo} = \lambda n. n + 2, \text{ or } \text{addTwo} = \bot n + 2, \text{ for short}
  \]
- Such a function is **strict**, because it simply transmits bottom if it receives it – it cannot “correct” an error
- Non-strict functions return a result if given botom as input: \((\lambda n. \text{one}) \bot = \text{one}\)