Δενοτατιοναλ Σεμαντικοσ
(Denotational Semantics)

Formal Semantics

What does \( y := f(x) + x \) mean?

- \( y \) is assigned the value of \( f(x) + x \)
- \( y \) becomes a pointer to the result of \( f(x) + x \)
- \( f(x) \) may or may not have side effects
- statement is undefined if types aren't equivalent
- statement is undefined if types aren't compatible
- etc

- Need formal semantics to make meanings of programs unambiguous.
Utility of Formal Semantics

- Handy for:
  - language design
  - proofs of correctness
  - language implementation
  - reasoning about programs
  - providing a clear specification of behavior

Formal Semantics (continued)

Tennent: "... a precise specification of the meanings of programs for use by programmers, language designers and implementers, and in the theoretical investigations of language properties."

- Three major approaches:
  1) Denotational: define functions that map syntactic structures into mathematical objects (e.g. numbers, truth values & functions)
     (Algebraic) - considered a component of denotational
  2) Operational: formal virtual machine description (VDL, H-Graphs)
  3) Axiomatic: development of axioms defining meanings of classic statement types. (Dijkstra, Hoare)
Uses

- Denotational: Ashcroft and Wadge argue best use is language design. (as opposed to retrofit, as attempted with Ada). Used some for formal verification.

- Operational: Best for implementation description.

- Axiomatic: Most often used for formal verification.

Axiomatic Semantics

**Axioms:**

- **null:** \{P\} skip \{P\}
- **assignment:** \{P_E^x\} x:= E \{P\} where P_E^x is the assertion formed by replacing every occurrence of x in P by E.
- **alternation:** \{P ^ B \} S_1 \{ Q \}, \{P ^ \neg B \} S_2 \{ Q \}
  - \{P\} if B then S_1 else S_2 \{ Q \}
- **iteration:** \{P ^ B \} S \{ P \}
  - \{P\} while B do S \{P ^ \neg B\}
- **composition:** \{P_1\} S_1 \{P_2\}, \{P_2\} S_2 \{P_3\}, \ldots, \{P_n\} S_n \{P_{n+1}\}
  - \{P_1\} begin S_1, S_2, \ldots, S_n end \{P_{n+1}\}

**rules of inference**

a: antecedent  
c: consequent
### Axiomatic Semantics

**More axioms:**

- **consequence:** \( \{ P \} S \{ Q \}, \quad P \vdash_{ax} P \& Q \vdash_{ax} Q \)
  \( \{ P \} S \{ Q \} \)

- **await:** \( \{ P \land B \} S \{ Q \} \)
  \( \{ P \} \) await B then \( S \{ Q \} \)

- **cobegin:** \( \{ P \} S_1 \{ Q_1 \}, \ldots, \{ P \} S_n \{ Q_n \} \) are interference free
  \( \{ P \land \ldots \land P \} \) cobegin \( S_1 \// \ldots \// S_n \)
  \( \) coend \( \{ Q_1 \land \ldots \land Q_n \} \)

**Uses:** Dijkstra’s weakest preconditions

Temporal logic

---

### Using Axiomatic Semantics

Prove noninterference in the following:

\( \{ x = 0 \text{ and } y = 0 \} \)

**S:** cobegin

- **s1:** await true then \( y := y + 1 \)
  \( // \)

- **s2:** await true then \( x := x + 2 \)
  \( // \)

- **s3:** await \( y > 0 \) then \( x := x + y \)

coend

\( \{ x = 3 \text{ and } y = 1 \} \)
Denotational Semantics

- Assigning denotations to language constructs
- Utilizes domains and functions over domains
  - domains are sets with properties that allow us to deal with questions regarding
    - recursive definitions of functions (over domains)
    - recursive definitions of domains

  e.g. consider (recursive function over domain)
  
  \[ f: \text{Num} \rightarrow \text{Num} \quad \text{-- f: maps numbers into numbers} \]

Candidates f’s

- Two candidate "defining" functions for f:
  
  (i) \[ fx = (fx) + 1 \]
  (ii) \[ fx = fx \]

- Assuming Num = \{0,1,2,...\}, there is no f for (i) and every f satisfies (ii).

- In contrast:
  
  (iii) \[ fx = (x=0) \rightarrow 1, \quad x \neq f(x-1) \]

  uniquely defines f as factorial
Scott’s Theory

- Scott's (1969) theory of domains ensures every definition is good by:
  - requiring all domains to have an "implicit structure." This requirement guarantees that all equations (e.g. i, ii and iii) have at least one solution.
  - providing direction, using implicit structure, for choosing an "intended" solution from the solutions guaranteed by (a).
    - based on lattices and fixed point theory.

- e.g. Num consists of 0, 1, 2, ... and undefined
  - Num⊥ is called a lifted domain

Defining Moment

- Thus,
  (i) and (ii) define f to be undefined and (iii) defines f as
    fx = x! if x=0, 1, 2, ...
    and f undefined = undefined
- Using ⊥ as a value is an alternative to using partial functions.
- With ⊥, all elements in domain have a value.
  - e.g. f undefined = undefined

- Scott's theory applies as well to recursive definitions of domains.
  - e.g. lists defined in terms of lists
On Defining a Language's Denotational Semantics

Three components:
• Abstract syntax (syntactic domain)
  – list of syntactic categories
  – list of syntactic clauses (a mapping onto immediate constituents)
• Semantic Domain (Semantic Algebras)
  – domain equations: provide framework for defining denotations
  – sets that are used as value spaces in PL semantics
• Semantic functions
  – functions that define denotation of constructs
  – semantic clauses

Terms

• $\lambda x.e$: Church's lambda notation (seen before)
• $\lambda x.e : A_\perp \rightarrow B_\perp := (\lambda x.e)\perp = \perp$
  
  $(\lambda x.e)a = [a/x]e$ for $a \neq \perp$
  
  "proper element"

  – $\lambda x.e$ is e.g. of a strict operation
  – non-strict operations allow $\perp$ to be mapped to proper elements

• (let $x = e_1$ in $e_2$) is a syntactic substitute for $(\lambda x.e_2)e_1$

• diverge: statement that goes into an infinite loop
More Terms

- $x \rightarrow e_1 \mid e_2$: syntactic form for conditional
  
e.g. $C[\text{If } B \text{ THEN } C_1 \text{ ELSE } C_2] = \lambda s. B[B]s \rightarrow C[C_1]s \mid C[C_2]s$

- Expressions in mini-language assumed to have *no* side effects.
  - e.g. no reads in expressions.

- $[i \rightarrow n]s$ is a function updating expression
  
  $([i \rightarrow n]s)(i) = n$
  $([i \rightarrow n]s)(j) = s(j) \quad \forall j \neq i$

  - useful for reflecting effects of updating the $i^{th}$ component of a store: $i^{th}$ component changes; rest stays the same
  - update's signature: $\text{Id} \times \text{Nat} \times \text{Store} \rightarrow \text{Store}$

Even More Terms

- Interpretation of:
  
  $P[C.] = \lambda n. \text{let } s = (\text{update } [A] n \text{ newstore}) \text{ in}$
  
  $\text{let } s' = C[C]s \text{ in (access } [Z]s')$

  - input number is associated with identifier $[A]$ in a new store
  - then program body is evaluated
  - then answer is extracted from store at $[Z]$

  (program mapping: $\text{Nat} \rightarrow \text{Nat}_{\perp} \quad -- \perp$ is possible because $\text{diverge}$ is possible)

- Clauses for $C$ are all strict in use of store

- $E$ does not modify store; expression evaluation order is not specified
  - e.g. $E[E_1]s$ plus $E[E_2]s$

- Same for Booleans
A Small Imperative Language

- Abstract Syntax
  
  \[
  P \in \text{Program} \\
  C \in \text{Command} \\
  E \in \text{Expression} \\
  B \in \text{Boolean-expr} \\
  N \in \text{Numeral}
  \]

  \[
P ::= C. \\
  C ::= \text{if } B \text{ then } C | \text{if } B \text{ then } C_1 \text{ else } C_2 | I := E | \text{diverge}
  \\
  E ::= E_1 + E_2 | I | N
  \\
  B ::= E_1 = E_2 | \neg B
  \]

A Small Imperative Language (cont)

- Semantic domain
  
  . . .

- Semantic Functions
  
  \[
P : \text{Program} \to \text{Nat} \to \text{Nat}_\
  
  P[C.] = \lambda n. \text{let } s = (\text{update}[A] \text{ n newstore}) \text{ in} \\
  \text{let } s' = \text{C[C]s in (access[Z] s')}
  \\
  C : \text{Command} \to \text{Store}_\downarrow \to \text{Store}_\downarrow
  
  C[C_1; C_2] = \lambda s. C[C_2] (C[C_1]s)
  
  C[\text{if } B \text{ then } C] = \lambda s. B[B]s \to \text{C[C]s} \mid s
  
  C[\text{if } B \text{ then } C_1 \text{ else } C_2] = \lambda s. B[B]s \to \text{C[C_1]s} \mid \text{C[C_2]s}
  
  C[I := E] = \lambda s. \text{update } [I] (E[E]s) s
  
  C[\text{diverge}] = \lambda s. \bot
  \]
A Small Imperative Language (cont)

- Semantic Functions (cont)
  \[ E : \text{Expression} \rightarrow \text{Store} \rightarrow \text{Nat} \]
  \[ E[E_1+E_2] = \lambda s. E[E_1]_s \text{plus} E[E_2]_s \]
  \[ E[I] = \lambda s. \text{access}[I]_s \]
  \[ E[N] = \lambda s. N[N] \]

- Boolean-expr \rightarrow \text{Store} \rightarrow \text{Tr}
  \[ B[E_1=E_2] = \lambda s. E[E_1]_s \text{equals} E[E_2]_s \]
  \[ B[\neg B] = \lambda s. \text{not} B[B]_s \]

N: Numeral \rightarrow \text{Nat} (omitted)

Semantic Domain

- Truth Values
  Domain \( t \in Tr = B \)
  Operations
    \( \text{true, false: Tr} \)
    \( \text{not: Tr} \rightarrow Tr \)
- Identifiers
  Domain \( i \in Id = \text{Identifier} \)
- Natural numbers
  Domain \( n \in \text{Nat} = N \)
  Operations
    \( \text{zero, one, …: Nat} \)
    \( \text{plus: Nat x Nat} \rightarrow \text{Nat} \)
    \( \text{equals: Nat x Nat} \rightarrow Tr \)
Semantic Domain (cont)

- Store
  Domain \( s \in \text{Store} = \text{Id} \rightarrow \text{Nat} \)
  Operations
    \( \text{newstore: Store} \)
    \( \text{newstore} = \lambda i. \text{zero} \)
    \( \text{access: Id} \rightarrow \text{Store} \rightarrow \text{Nat} \)
    \( \text{access} = \lambda i. \lambda s. s(i) \)
    \( \text{update: Id} \rightarrow \text{Nat} \rightarrow \text{Store} \rightarrow \text{Store} \)
    \( \text{update} = \lambda i. \lambda n. \lambda s. [i \rightarrow n]s \)