Denotational Semantics

COS 441
Princeton University
Fall 2004

Denotational Semantics
Describe meaning of syntax by specifying the mathematical meaning of syntax tree as
- Function
- Function on functions
- Value, such as natural numbers or strings
- Sets of values ... etc
defined by each construct

Original Motivation for Topic
- Precision
  - Use mathematics instead of English
- Avoid details of specific machines
  - Aim to capture “pure meaning” apart from implementation details
- Basis for program analysis
  - Justify program proof methods
    - Soundness of type system, control flow analysis
  - Proof of compiler correctness
  - Language comparisons

Denotational vs. Operational
- Denotational semantics are more “abstract” than operational approaches
- Cannot reason about number of steps of a computation or algorithmic complexity
- Specify what the answer should be not how a computation takes place

Principles of DS
Compositionality
The meaning of a compound program must be defined from the meanings of its parts (not the syntax of its parts).
Examples
P; Q
  composition of two functions, state → state
letrec f(x) = e1 in e2
  meaning of e2 where f denotes function ...

Notation and Definitions
Phrase
variable, expression, statement, declarations
[[ e ]], [[ P ]]
Syntax tree of a phrase
E[[ e ]], C [[ P ]]
Meaning functions (E,C) applied to syntax trees of phrases
Example: Binary Numbers

Syntax
\[
\begin{align*}
b & ::= 0 \mid 1 \\
n & ::= b \mid nb \\
e & ::= n \mid e_1 + e_2
\end{align*}
\]

Semantics
\[
\begin{align*}
E[[0]] &= 0 \\
E[[1]] &= 1 \\
E[[nb]] &= 2 \times E[[n]] + E[[b]] \\
E[[e_1 + e_2]] &= E[[e_1]] + E[[e_2]]
\end{align*}
\]

DS of Regular Expressions

Syntax
\[
\begin{align*}
a, b & \in \Sigma \\
re & ::= a \mid re_1 + re_2 \mid re_1 \cdot re_2 \mid re^*
\end{align*}
\]

Semantics
\[
E : \text{Regular Exp} \rightarrow \Sigma^*
\]

Sets of Strings

Let \(a, b \in \Sigma\)
- \(\Sigma\) is an unspecified alphabet
- \(\Sigma^*\) is the set of strings made up from the alphabet \(\Sigma\)
- \(\varepsilon \in \Sigma^*\)
- \(a \in \Sigma^*\) if \(a \in \Sigma\)
- \(s_1 \cdot s_2 \in \Sigma^*\) if \(s_1 \in \Sigma^*\) and \(s_2 \in \Sigma^*\)

Equality on Strings

Let \(s_1, s_2, s_3 \in \Sigma^*\)
\[
\begin{align*}
a &= a \\
\varepsilon \cdot s &= s \\
s \cdot \varepsilon &= s \\
(s_1 \cdot s_2) \cdot s_3 &= s_1 \cdot (s_2 \cdot s_3) \\
s_1 \cdot s_2 &= s_1' \cdot s_2' & \text{if } s_1 = s_1' \text{ and } s_2 = s_2'
\end{align*}
\]

Operations On Sets of Strings

- \(\{a\}\) is \(\{x \mid x = a\}\)
- \(S_1 \cup S_2\) is \(\{x \mid x \in S_1 \lor x \in S_2\}\)
- \(S_1 \cdot S_2\) is \(\{x \mid s_1 \in S_1 \land s_2 \in S_2 \land x = s_1 \cdot s_2\}\)
- \(S^0\) is \(\{\varepsilon\}\)
- \(S^{n+1}\) is \(\{x \mid x \in S \cdot S^n\}\)
- \(S^*\) is \(\{x \mid \exists n. x \in S^n\}\)

Some Theorems

\[
\begin{align*}
E[a + b] &= \{a, b\} \\
E[a \cdot (b + c)] &= \{ab, ac\} \\
E[[a + b] \cdot c]] &= \{ac, bc\} \\
E[[re_1 \cdot (re_2 + re_3)] &= E[[re_1 \cdot re_2] + (re_1 \cdot re_3)]] \\
E[[a^*]] &= E[[a^*]]
\end{align*}
\]
DS for Regular Expressions

\[ E[\text{a}] \equiv \{ \text{a} \} \]
\[ E[\text{re}_1 + \text{re}_2] \equiv \{ x \mid x \in E[\text{re}_1] \cup E[\text{re}_2] \} \]
\[ E[\text{re}_1 \cdot \text{re}_2] \equiv \{ x \mid x \in E[\text{re}_1] \cdot E[\text{re}_2] \} \]
\[ E[\text{re}^*] \equiv \{ x \mid x \in E[\text{re}]^* \} \]

Another DS for REs

- We can given an equivalent semantics by relating regular expressions to Non-deterministic Finite Automata (NFAs)
- Equivalent means
  - Any theorem about meanings in one semantics is true iff the same theorem is true in the other model

NFA

- States: \( q_0, q_1, \ldots, q_n \in Q \)
- Initial State: \( q_{\text{init}} \in Q \)
- Final States: \( F \subset Q \)
- Transition Relation: \( \delta \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q \)
- \( M = (q_{\text{init}}, F, \delta) \) is an NFA

NFA Accepting a String

- \( \varepsilon \in L(q_{\text{init}}, F, \delta) \)
- if \((q_{\text{init}}, \varepsilon, q_i) \in \delta\) and \(q_i \in F\)
- \( a \in L(q_{\text{init}}, F, \delta) \)
- if \((q_{\text{init}}, a, q_i) \in \delta\) and \(q_i \in F\)
- \( x \cdot s \in L(q_{\text{init}}, F, \delta) \)
- if \((q_{\text{init}}, x, q') \in \delta\) and \(s \in L(q', F, \delta)\)

Semantics of Regular Expressions

- \( \text{NFA}[\text{a}] \equiv (q, \{q_i\}, (q, a, q_i)) \)

Semantics of Regular Expressions

- \( \text{NFA}[\text{re}_1 + \text{re}_2] \equiv \)
- let \( q' \) be a unique new state
- let \((q_1, F_1, \delta_1) = \text{NFA}[\text{re}_1]\)
- let \((q_2, F_2, \delta_2) = \text{NFA}[\text{re}_2]\)
- let \( \delta = \delta_1 \cup \delta_2 \cup \{(q', \varepsilon, q_1), (q, \varepsilon, q_2)\}\)
- \((q', F_1 \cup F_2, \delta)\)
Semantics of Regular Expressions

NFA[[re₁ . re₂]] ≡
let (q₁, F₁, δ₁) = NFA[[re₁]]
let (q₂, F₂, δ₂) = NFA[[re₂]]
let δ = δ₁ ∪ δ₂ ∪ {((q, ε, q₂) | q ∈ F₁)}
(q₁, F₂, δ)

Semantics of Regular Expressions

NFA[[re*]] ≡
let q', qᵢ be new states
let (qᵢ, F, δ) = NFA[[re]]
let δ' = δ ∪ {((q', ε, qᵢ), (q', ε, qᵢ)) ∪ {((q, ε, q) | q ∈ F)}
(q', (q, δ))

Some More Theorems

L(NFA[[a + b]]) ≡ {a . b}
L(NFA[[a . (b + c)]]) ≡ {ab, ac}
L(NFA[[a + b] . c]) ≡ {ac, bc}

E[[re]] = L(NFA[[re]])

- We can prove the NFA semantics is equivalent to the set theoretic semantics via induction on the definition of the meaning function E
- Proof relies on compositional definition of meaning function E!

Which Semantics?

- Set of string semantics clearly easier to reason about!
  - This is what denotation semantics was designed for
- NFA semantics can be transformed into efficient of regular expression matcher
  - Our theorem is a correctness proof about the implementation of our NFA conversion function

DS for Programming Languages

- We can build a DS for a programming language
- Allows us to reason about program equivalence
  - Useful to prove compiler optimizations is safe
- Provides framework for reason about abstract properties of programs statically
DS of Imperative Programs

Syntax

- \( n \in \text{Numbers} \)
- \( x \in \text{Vars} \)
- \( e ::= \text{false} | \text{true} | n | x | e_1 + e_2 | e_1 \leq e_2 \)
- \( P ::= x := e | \text{if } e \text{ then } P_1 \text{ else } P_2 | \text{while } e \text{ do } P \)

Semantics for Expression

Meaning function for expressions

- \( \text{State } = \text{Vars} \rightarrow \text{Numbers} \)
- \( E : \text{Expressions} \rightarrow \text{State} \rightarrow \text{Numbers} \)

- \( E[\text{false}](s) = 0 \)
- \( E[\text{true}](s) = 1 \)
- \( E[n](s) = n \)
- \( E[x](s) = s(x) \)
- \( E[e_1 + e_2](s) = E[e_1](s) + E[e_2](s) \)
- \( E[e_1 \leq e_2](s) = \text{if } E[e_1](s) \leq E[e_2](s) \text{ then } 1 \text{ else } 0 \)

Semantics for Programs

Meaning function for programs

- \( \text{Command } = \text{State} \rightarrow \text{State} \)
- \( C : \text{Programs} \rightarrow \text{Command} \)

- \( C[\text{P;Q}](s) = C[Q](C[P](s)) \)
- \( C[\text{if } e \text{ then } P \text{ else } Q](s) = \)
  - \( C[P](s) \) if \( E[e](s) = 1 \)
  - \( C[Q](s) \) if \( E[e](s) = 0 \)

Semantics of Assignment

- \( \text{modify} : \text{State} \times \text{Vars} \times \text{Numbers} \rightarrow \text{State} \)
- \( \text{modify}(s,x,a) = \lambda y. \text{if } y = x \text{ then } a \text{ else } s(y) \)

- \( C[\text{x:= e}](s) = \text{modify}(s,x,E[e](s)) \)

Semantics of Iteration

- \( C[\text{while } e \text{ do } P](s) = \)
  - \( \text{The function } f \text{ such that} \)
  - \( f(s) = s \) if \( E[e](s) = 0 \)
  - \( f(s) = f(C[P](s)) \) if \( E[e](s) = 1 \)

Mathematics of denotational semantics:

- prove that there is such a function and that it is uniquely determined. “Beyond scope of this course.”

Mathematical Foundations

A full DS for imperative programs requires the definition of a special class of sets called “domains.”

From Wikipedia, the free encyclopedia.

Dana S. Scott is the incumbent Hillman University Professor of Computer Science, Philosophy, and Mathematical Logic at Carnegie Mellon University. His contributions include early work in automata theory, for which he received the ACM Turing Award in 1976, and the independence of the Boolean prime ideal theorem.

Scott is also the founder of domain theory, a branch of order theory that is used to model computation and approximation, and that provides the denotational semantics for the lambda calculus.

He received his Bachelor's degree from the University of California, Berkeley in 1954, and his Ph.D. from Princeton University in 1960.
Abstract Interpretation

Abstract interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g. control structure, flow of information) without performing all the calculations.

Abstract interpretation was formalized by Patrick Cousot.

Abstract vs. Standard Semantics

Given a standard semantics abstract interpretation "approximates" the standard semantics in a way that guarantees the abstract interpretation is correct with respect to the original semantics.

The original DS semantics again is used as part of a correctness criterion for the abstract semantics.

The Meaning of Meaning

- The denotational relates of syntax trees to objects in mathematical functions expressed using a metalanguage for defining functions.
- Strachey and Scott used the mathematical theory of continuous function over partially ordered sets (domains) as their "target" semantics.
- We can explain the meaning of programs in using the Strachey and Scott metalanguage.

DS Scheme

Formal semantics of Scheme is defined using the Strachey and Scott metalanguage. See [http://www.schemers.org/Documents/Standards/R5RS/r5rs.pdf](http://www.schemers.org/Documents/Standards/R5RS/r5rs.pdf)

A non-trivial semantics language

Abstract Syntax of Scheme

\[
\begin{align*}
K & \in \text{Con} & \text{constants, including quotations} \\
I & \in \text{Ide} & \text{identifiers (variables)} \\
E & \in \text{Exp} & \text{expressions} \\
\Gamma & \in \text{Com} & \text{commands} \\
\text{Exp} & \to K | I | (E_0 \ E^*) \\
 & | \text{lambda} (I^* \ \Gamma^* \ E_0) \\
 & | \text{lambda} (I^* \ D \ \Gamma^* \ E_0) \\
 & | \text{lambda} I \ I^* \ E_0 \\
 & | \text{if} E_0 \ E_1 \ E_2 \\
 & | \text{if} E_0 \ E_1 \\
 & | \text{set!} \ I \ E
\end{align*}
\]

Semantic Values

\[
\begin{align*}
\alpha & \in L & \text{locations} \\
\nu & \in \mathbb{N} & \text{natural numbers} \\
T & = \{\text{false, true}\} & \text{booleans} \\
Q & \text{symbols} \\
H & \text{characters} \\
R & \text{numbers} \\
E_p & = L \times L \times T & \text{pairs} \\
E_v & = L^* \times T & \text{vectors} \\
E_s & = L^* \times T & \text{strings} \\
M & = \{\text{false, true, null, undefined, unspecified}\} & \text{miscellaneous}
\end{align*}
\]
Semantic Values (cont.)

$\phi \in F = L \times (E^* \to K \to C)$ procedure values
$\epsilon \in E = Q + H + R + E_v + E_e + E_e + H + F$

Semantic Functions (cont.)

$\mathcal{K} : \text{Con} \to \mathbb{E}$
$\mathcal{E} : \text{Exp} \to \mathbb{U} \to K \to C$
$\mathcal{E}^* : \text{Exp}^* \to \mathbb{U} \to K \to C$
$\mathcal{C} : \text{Com}^* \to \mathbb{U} \to C \to C$

$\mathcal{E}[\rho] \equiv \lambda \rho. \text{hold}(\text{lookup } \rho) ($

Semantic Functions (cont.)

$\mathcal{E}[(\text{set} \; t \; \text{I} \; \text{E}_0)] =$

$\lambda \rho. \mathcal{E}[\text{E}_0] \rho (\text{single}(\lambda e. \text{assign}(\text{lookup } \rho) \epsilon$n

$\mathcal{E}[(\text{if} \; \text{E}_0 \; \text{E}_1 \; \text{E}_2)] =$

$\lambda \rho. \mathcal{E}[\text{E}_1] \rho (\text{single}(\lambda e. \text{truish } \epsilon \rightarrow \mathcal{E}[\text{E}_1] \rho \epsilon, \mathcal{E}[\text{E}_2] \rho \epsilon))$

$\mathcal{E}[(\text{E}_0 \; \text{E}_1 \; \text{E}_2)] =$

$\lambda \rho. \mathcal{E}^*(\text{permute}(\langle \text{E}_0 \rangle \; \text{E}_1 \; \text{E}_2))$

SML as a Metalanguage

- Scheme semantics looks awfully like a complicated program written in an obscure language of functions which happen to have a precise well-understood meaning (If you’re a mathematician that is)
- We could also use SML as a metalanguage to express semantics!

Semantics of Expressions in SML

```
type var = string
datatype exp =
    True
| False
| N of int
| V of var
| Plus of exp * exp
| Minus of exp * exp
| Leq of exp * exp
```
Semantics of Expressions in SML

type value = int

val type state = var -> value

(* val expSem : exp -> state -> value *)

fun expSem (True)(s) = 1
| expSem (False)(s) = 0
| expSem (N i)(s) = i
| expSem (V v)(s:state) = s v
| expSem (Plus(e1,e2))(s) = expSem(e1)(s) + expSem(e2)(s)
| expSem (Minus(e1,e2))(s) = expSem(e1)(s) - expSem(e2)(s)
| expSem (Leq(e1,e2))(s) = if expSem(e1)(s) <= expSem(e2)(s)
| then 1 else 0

Summary

• Denotational techniques provide giving meaning to programs by relating syntax to the semantics of some well-defined metalanguage
• DS are abstract semantics that are good to reason about correctness of implementations or static analysis

Next Lecture

• Using denotational techniques to specify the semantics of resolution independent graphical objects
• Homework for this week turn our semantics into SML code that takes a simple picture language into an image!