Lecture 8
Denotational Semantics

10 September 2012
Announcements

- Homework #2 due tonight at 11:59pm
- Foster office hours today 4-5pm in Upson 4137
- Rajkumar office hours today 5-6pm in 4135
- Homework #3 goes out today
Recap

So far, we’ve:

- Formalized the operational semantics of an imperative language
- Developed the theory of inductive sets
- Used this theory to prove formal properties:
  - Determinism
  - Soundness (via Progress and Preservation)
  - Termination
  - Equivalence of small-step and large-step semantics
- Developed an implementation in OCaml
- Extended to IMP, a more complete imperative language

Today we’ll develop a denotational semantics for IMP
An operational semantics models how a program executes on an idealized machine:

\[
\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \quad \langle \sigma, e \rangle \downarrow \langle \sigma', n \rangle
\]
An operational semantics models how a program executes on an idealized machine:

\[ \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \quad \langle \sigma, e \rangle \Downarrow \langle \sigma', n \rangle \]

A denotational semantics models what a program computes.
Denotational Semantics

An operational semantics models how a program executes on an idealized machine:

\[ \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \quad \langle \sigma, e \rangle \downarrow \langle \sigma', \mathit{n} \rangle \]

A denotational semantics models what a program computes.

More specifically, a denotational semantics defines the meaning of a program directly, as a mathematical function:

\[ C[c] \in \text{Store} \rightarrow \text{Store} \]
Syntax

\[ a \in \text{Aexp} \quad a ::= x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \]

\[ b \in \text{Bexp} \quad b ::= \text{true} \mid \text{false} \mid a_1 < a_2 \]

\[ c \in \text{Com} \quad c ::= \text{skip} \mid x ::= a \mid c_1; c_2 \]

\[ \quad \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \]
IMP

Syntax

\[ a \in Aexp \quad a ::= \ x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \]
\[ b \in Bexp \quad b ::= \ \text{true} \mid \text{false} \mid a_1 < a_2 \]
\[ c \in Com \quad c ::= \ \text{skip} \mid x := a \mid c_1 ; c_2 \]
\[ \quad \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \]

Semantic Domains

\[ C[c] \in \text{Store} \rightarrow \text{Store} \]
\[ A[a] \in \text{Store} \rightarrow \text{Int} \]
\[ B[b] \in \text{Store} \rightarrow \text{Bool} \]
IMP

Syntax

\[ a \in \text{Aexp} \quad a ::= \ x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \]

\[ b \in \text{Bexp} \quad b ::= \ \text{true} \mid \text{false} \mid a_1 < a_2 \]

\[ c \in \text{Com} \quad c ::= \ \text{skip} \mid x := a \mid c_1 ; c_2 \]

\[ \quad \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \]

Semantic Domains

\[ \mathcal{C}[c] \in \text{Store} \rightarrow \text{Store} \]

\[ \mathcal{A}[a] \in \text{Store} \rightarrow \text{Int} \]

\[ \mathcal{B}[b] \in \text{Store} \rightarrow \text{Bool} \]

Why partial functions?
Conventions

Represent functions \( f : A \rightarrow B \) as sets of pairs:

\[
S = \{(a, b) \mid a \in A \text{ and } b = f(a) \in B\}
\]

such that, for each \( a \in A \), there is at most one pair \((a, \_\) in \( S \). That is, \((a, b) \in S\) if and only if \( f(a) = b\).

**Convention #2:** Define functions point-wise.

Equation \( C[c] = S \) defines the denotation function \( C[\_\_] \) on \( c \).
Denotational Semantics of IMP

\[ A[n] = \{(\sigma, n)\} \]
\[ A[x] = \{(\sigma, \sigma(x))\} \]
\[ A[a_1 + a_2] = \{(\sigma, n) \mid (\sigma, n_1) \in A[a_1] \land (\sigma, n_2) \in A[a_2] \land n = n_1 + n_2\} \]

\[ B[\text{true}] = \{(\sigma, \text{true})\} \]
\[ B[\text{false}] = \{(\sigma, \text{false})\} \]
\[ B[a_1 < a_2] = \{(\sigma, \text{true}) \mid (\sigma, n_1) \in A[a_1] \land (\sigma, n_2) \in A[a_2] \land n_1 < n_2\} \cup \{(\sigma, \text{false}) \mid (\sigma, n_1) \in A[a_1] \land (\sigma, n_2) \in A[a_2] \land n_1 \geq n_2\} \]

\[ C[\text{skip}] = \{(\sigma, \sigma)\} \]
\[ C[x := a] = \{(\sigma, \sigma[x \mapsto n]) \mid (\sigma, n) \in A[a]\} \]
\[ C[c_1; c_2] = \{(\sigma, \sigma') \mid \exists \sigma''. ((\sigma, \sigma'') \in C[c_1] \land (\sigma'', \sigma') \in C[c_2])\} \]
\[ C[\text{if } b \text{ then } c_1 \text{ else } c_2] = \{(\sigma, \sigma') \mid (\sigma, \text{true}) \in B[b] \land (\sigma, \sigma') \in C[c_1]\} \cup \{(\sigma, \sigma') \mid (\sigma, \text{false}) \in B[b] \land (\sigma, \sigma') \in C[c_2]\} \]
\[ C[\text{while } b \text{ do } c] = \{(\sigma, \sigma) \mid (\sigma, \text{false}) \in B[b]\} \cup \{(\sigma, \sigma') \mid (\sigma, \text{true}) \in B[b] \land \exists \sigma''. ((\sigma, \sigma'') \in C[c] \land (\sigma'', \sigma') \in C[\text{while } b \text{ do } c])\} \]
Recursive Definitions

Problem: the last “definition” in our semantics is not really a definition!

\[
C[\text{while } b \text{ do } c] = \{(\sigma, \sigma) \mid (\sigma, \text{false}) \in B[b]\} \cup \\
\{(\sigma, \sigma') \mid (\sigma, \text{true}) \in B[b] \land \exists \sigma''. ((\sigma, \sigma'') \in C[c] \land \\
(\sigma'', \sigma') \in C[\text{while } b \text{ do } c])\}
\]

Why?
Recursive Definitions

**Problem:** the last “definition” in our semantics is not really a definition!

\[
\mathcal{C}[\textbf{while } b \textbf{ do } c] = \{(\sigma, \sigma) \mid (\sigma, \text{false}) \in \mathcal{B}[b]\} \cup \\
\{(\sigma, \sigma') \mid (\sigma, \text{true}) \in \mathcal{B}[b] \land \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[c] \land \\
(\sigma'', \sigma') \in \mathcal{C}[\textbf{while } b \textbf{ do } c])\}
\]

Why?

It expresses \(\mathcal{C}[\textbf{while } b \textbf{ do } c]\) in terms of itself.

So this is not a definition but a recursive equation.

What we want is the solution to this equation.
Recursive Equations

Example:

\[ f(x) = \begin{cases} 
0 & \text{if } x = 0 \\
 f(x - 1) + 2x - 1 & \text{otherwise} 
\end{cases} \]
Recursive Equations

Example:

\[ f(x) = \begin{cases} 
0 & \text{if } x = 0 \\
 f(x - 1) + 2x - 1 & \text{otherwise} 
\end{cases} \]

Question: What functions satisfy this equation?
Recursive Equations

Example:

\[ f(x) = \begin{cases} 
0 & \text{if } x = 0 \\
 f(x - 1) + 2x - 1 & \text{otherwise} 
\end{cases} \]

Question: What functions satisfy this equation?

Answer: \( f(x) = x^2 \)
Recursive Equations

Example:

\[ g(x) = g(x) + 1 \]
Recursive Equations

Example:

\[ g(x) = g(x) + 1 \]

Question: Which functions satisfy this equation?
Recursive Equations

Example:

\[ g(x) = g(x) + 1 \]

**Question**: Which functions satisfy this equation?

**Answer**: None!
Recursive Equations

Example:

\[ h(x) = 4 \times h \left( \frac{x}{2} \right) \]
Recursive Equations

Example:

\[ h(x) = 4 \times h \left( \frac{x}{2} \right) \]

Question: Which functions satisfy this equation?
Recursive Equations

Example:

\[ h(x) = 4 \times h \left( \frac{x}{2} \right) \]

**Question:** Which functions satisfy this equation?

**Answer:** There are multiple solutions.
Solving Recursive Equations

Returning the first example...

\[ f(x) = \begin{cases} 
0 & \text{if } x = 0 \\
 f(x - 1) + 2x - 1 & \text{otherwise} 
\end{cases} \]
Solving Recursive Equations

Can build a solution by taking successive approximations:

\[ f_0 = \emptyset \]
Solving Recursive Equations

Can build a solution by taking successive approximations:

\[ f_0 = \emptyset \]

\[ f_1 = \begin{cases} 
0 & \text{if } x = 0 \\
 f_0(x - 1) + 2x - 1 & \text{otherwise} 
\end{cases} \]

\[ = \{(0, 0)\} \]
Solving Recursive Equations

Can build a solution by taking successive approximations:

\[ f_0 = \emptyset \]

\[ f_1 = \begin{cases} 
0 & \text{if } x = 0 \\
 f_0(x - 1) + 2x - 1 & \text{otherwise}
\end{cases} \]

\[ f_1 = \{(0, 0)\} \]

\[ f_2 = \begin{cases} 
0 & \text{if } x = 0 \\
 f_1(x - 1) + 2x - 1 & \text{otherwise}
\end{cases} \]

\[ f_2 = \{(0, 0), (1, 1)\} \]
Solving Recursive Equations

Can build a solution by taking successive approximations:

\[ f_0 = \emptyset \]

\[ f_1 = \begin{cases} 
0 & \text{if } x = 0 \\
 f_0(x - 1) + 2x - 1 & \text{otherwise}
\end{cases} \]

\[ = \{(0, 0)\} \]

\[ f_2 = \begin{cases} 
0 & \text{if } x = 0 \\
 f_1(x - 1) + 2x - 1 & \text{otherwise}
\end{cases} \]

\[ = \{(0, 0), (1, 1)\} \]

\[ f_3 = \begin{cases} 
0 & \text{if } x = 0 \\
 f_2(x - 1) + 2x - 1 & \text{otherwise}
\end{cases} \]

\[ = \{(0, 0), (1, 1), (2, 4)\} \]
Solving Recursive Equations

We can model this process using a higher-order function $F$ that takes one approximation $f_k$ and returns the next approximation $f_{k+1}$:

$$F : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$$

where

$$(F(f))(x) = \begin{cases} 
0 & \text{if } x = 0 \\
 f(x - 1) + 2x - 1 & \text{otherwise}
\end{cases}$$
Fixed Points

A solution to the recursive equation is an $f$ such that $f = F(f)$.

**Definition:** Given a function $F : A \to A$, we have that $a \in A$ is a fixed point of $F$ if and only if $F(a) = a$.

**Notation:** Write $a = \text{fix}(F)$ to indicate that $a$ is a fixed point of $F$.

**Idea:** Compute fixed points iteratively, starting from the completely undefined function. The fixed point is the limit of this process:

\[
  f = \text{fix}(F) \\
  = f_0 \cup f_1 \cup f_2 \cup f_3 \cup \ldots \\
  = \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup F(F(F(\emptyset))) \cup \ldots \\
  = \bigcup_{i \geq 0} F^i(\emptyset)
\]
Denotational Semantics for **while**

Now we can complete our denotational semantics:

\[ C[\textbf{while } b \textbf{ do } c] = \text{fix}(F) \]
Now we can complete our denotational semantics:

\[ C[\text{while } b \text{ do } c] = \text{fix}(F) \]

where

\[
F(f) = \{ (\sigma, \sigma) \mid (\sigma, \text{false}) \in B[b] \} \cup \\
\{ (\sigma, \sigma') \mid (\sigma, \text{true}) \in B[b] \land \\
\exists \sigma''. ((\sigma, \sigma'') \in C[c] \land (\sigma'', \sigma') \in f) \}
\]