Lecture 7
Denotational Semantics

9 September 2016
Announcements

- Homework #2 out
- Please consider finding a partner!
- Advance warning: the next homework (#3) will involve OCaml programming
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
- Extended to IMP, a more complete imperative language

Today we’ll develop a denotational semantics for IMP
Denotational Semantics

An operational semantics, like an interpreter, describes how to evaluate a program:

\[ \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \quad \langle \sigma, e \rangle \downarrow \langle \sigma', n \rangle \]
Denotational Semantics

An operational semantics, like an interpreter, describes how to evaluate a program:

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

A denotational semantics, like a compiler, describes a translation into a different language with known semantics—namely, math.
Denotational Semantics

An operational semantics, like an interpreter, describes how to evaluate a program:

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

A denotational semantics, like a compiler, describes a translation into a different language with known semantics—namely, math.

A denotational semantics defines what a program means as a mathematical function:

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

\[ a \in Aexp \quad a ::= \ x \ | \ n \ | \ a_1 + a_2 \ | \ a_1 \times a_2 \]

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

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

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

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

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

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

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

Semantic Domains
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

\[ C[c] \in \text{Store} \to \text{Store} \]

Why partial functions?
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

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

Why partial functions?
Syntax

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

Semantic Domains

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

Why partial functions?
Convention #1: 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 $(a, b) \in S$ if and only if $f(a) = b$.

(For each $a \in A$, there is at most one pair $(a, \_)$ in $S$.)

Convention #2: Define functions point-wise.

Where $C[\cdot]$ is the denotation function, the equation $C[c] = S$ gives its definition for the command $c$. 
Denotational Semantics of IMP

\[ A[n] = \{(\sigma, n)\} \]
Denotational Semantics of IMP

\[ A[n] = \{(\sigma, n)\} \]

\[ A[x] = \{(\sigma, \sigma(x))\} \]
Denotational Semantics of IMP

\[ A[n] = \{(\sigma, n)\} \]

\[ A[x] = \{(\sigma, \sigma(x))\} \]

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

\[ A[a_1 \times a_2] = \{(\sigma, n) \mid (\sigma, n_1) \in A[a_1] \land (\sigma, n_2) \in A[a_2] \land n = n_1 \times n_2\} \]
\[ B[\text{true}] = \{(\sigma, \text{true})\} \]
Denotational Semantics of IMP

\[ B[\text{true}] = \{(\sigma, \text{true})\} \]

\[ B[\text{false}] = \{(\sigma, \text{false})\} \]
Denotational Semantics of IMP

\[ B[\text{true}] = \{(\sigma, \text{true})\} \]

\[ B[\text{false}] = \{(\sigma, \text{false})\} \]

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

\[ C[\text{skip}] = \{(\sigma, \sigma)\} \]
Denotational Semantics of IMP

\[
\begin{align*}
C[\text{skip}] &= \{(\sigma, \sigma)\} \\
C[x := a] &= \{(\sigma, \sigma[x \mapsto n]) \mid (\sigma, n) \in A[a]\}
\end{align*}
\]
Denotational Semantics of IMP

\[ C[\text{skip}] = \{(\sigma, \sigma)\} \]

\[ C[x := a] = \{(\sigma, \sigma[x \mapsto n]) | (\sigma, n) \in A[a]\} \]

\[ C[c_1; c_2] = \{(\sigma, \sigma') | \exists \sigma'' . ((\sigma, \sigma'') \in C[c_1] \land (\sigma'', \sigma') \in C[c_2])\} \]
Denotational Semantics of IMP

\[ C[\text{skip}] = \{(\sigma, \sigma)\} \]

\[ C[x := a] = \{(\sigma, \sigma[x \rightarrow n]) | (\sigma, n) \in A[a]\} \]

\[ C[c_1; c_2] = \{(\sigma, \sigma') | \exists \sigma''. ((\sigma, \sigma'') \in C[c_1] \land (\sigma'', \sigma') \in C[c_2])\} \]

\[ C[\text{if } b \text{ then } c_1 \text{ else } c_2] = \{(\sigma, \sigma') | (\sigma, \text{true}) \in B[b] \land (\sigma, \sigma') \in C[c_1]\} \cup \{(\sigma, \sigma') | (\sigma, \text{false}) \in B[b] \land (\sigma, \sigma') \in C[c_2]\} \]
Denotational Semantics of IMP

\[ C[\text{skip}] = \{(\sigma, \sigma)\} \]

\[ C[x := a] = \{(\sigma, \sigma[x \mapsto n]) \mid (\sigma, n) \in A[a]\} \]

\[ C[c_1; c_2] = \{(\sigma, \sigma') \mid \exists \sigma''. ((\sigma, \sigma'') \in C[c_1] \land (\sigma'', \sigma') \in C[c_2])\} \]

\[ C[\text{if } b \text{ then } c_1 \text{ else } c_2] = \]
\[ \{(\sigma, \sigma') \mid (\sigma, \text{true}) \in B[b] \land (\sigma, \sigma') \in C[c_1]\} \cup \]
\[ \{(\sigma, \sigma') \mid (\sigma, \text{false}) \in B[b] \land (\sigma, \sigma') \in C[c_2]\} \]

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

It expresses \( C[\text{while } b \text{ do } c] \) 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?
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:

\[ f_0 = \emptyset \]

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

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

Can build a solution by taking successive approximations:

\[ f_0 = \emptyset \]

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

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

\[ f_3 = \begin{cases} 
0 & \text{if } x = 0 \\
 f_2(x - 1) + 2x - 1 & \text{otherwise} 
\end{cases} = \{(0, 0), (1, 1), (2, 4)\} \]

\ldots
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 $f_{k+1}$:

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

where

$$(F(f))(x) = \begin{cases} 
0 & \text{if } x = 0 \\
 f(x - 1) + 2x - 1 & \text{otherwise}
\end{cases}$$
A solution to the recursive equation is an \( f \) such that \( f = F(f) \).

**Definition:** Given a function \( F : A \rightarrow A \), we say 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 \texttt{while}

Now we can complete our denotational semantics:

\[ C[\texttt{while } b \texttt{ do } c] = \text{fix}(F) \]
Denotational Semantics for \textbf{while}

Now we can complete our denotational semantics:

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

where

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