Lecture 7
Denotational Semantics
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:

\[ \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \quad \langle \sigma, e \rangle \downarrow \langle \sigma', n \rangle \]
An operational semantics, like an interpreter, describes *how* to evaluate a program:

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

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

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

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

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

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

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

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

Why partial functions? well-formed
Notational Conventions

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 $c$. 
Notational Conventions

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 \(c\).

Applying this notation twice, \(C[C[c]]_\sigma = \sigma'\) gives the value for the \(C[C[c]]\) function at \(\sigma\).
Denotational Semantics of IMP

Arithmetic expressions:

\[ A[n] \triangleq \{(\sigma, n)\} \quad A \downarrow n \sigma \triangleq n \]
Denotational Semantics of IMP

Arithmetic expressions:

\[
\begin{align*}
\mathcal{A}[n] & \triangleq \\
& \{ (\sigma, n) \} \\
\mathcal{A}[x] & \triangleq \\
& \{ (\sigma, \sigma(x)) \}
\end{align*}
\]

\[
\mathcal{A}[x] \sigma \triangleq \sigma \langle x \rangle
\]
Denotational Semantics of IMP

Arithmetic expressions:

\[
\begin{align*}
\mathcal{A}[n] & \triangleq \{(\sigma, n)\} \\
\mathcal{A}[x] & \triangleq \{(\sigma, \sigma(x))\} \\
\mathcal{A}[a_1 + a_2] & \triangleq \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[a_1] \land (\sigma, n_2) \in \mathcal{A}[a_2] \land n = n_1 + n_2\} \\
\mathcal{A}[a_1 \times a_2] & \triangleq \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[a_1] \land (\sigma, n_2) \in \mathcal{A}[a_2] \land n = n_1 \times n_2\}
\end{align*}
\]
Denotational Semantics of IMP

Boolean expressions:

\[ \mathcal{B}[\text{true}] \triangleq \{(\sigma, \text{true})\} \]
Denotational Semantics of IMP

Boolean expressions:

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

\[ B[6] \models \sigma \models b \]
Denotational Semantics of IMP

Boolean expressions:

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

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

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

Or, using the function-style notation:

\[ A[n]_\sigma \triangleq n \]
\[ A[x]_\sigma \triangleq \sigma(x) \]
\[ A[a_1 + a_2]_\sigma \triangleq A[a_1]_\sigma + A[a_2]_\sigma \]
\[ A[a_1 \times a_2]_\sigma \triangleq A[a_1]_\sigma \times A[a_2]_\sigma \]

\[ B[true]_\sigma \triangleq true \]
\[ B[false]_\sigma \triangleq false \]
\[ B[a_1 < a_2]_\sigma \triangleq \begin{cases} true & \text{if } A[a_1]_\sigma < A[a_2]_\sigma \\ false & \text{otherwise} \end{cases} \]
Denotational Semantics of IMP

Commands:

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

\[ C[\text{skip}] \sigma \triangleq \sigma \]
Denotational Semantics of IMP

Commands:

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

Commands:

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

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

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

Commands:

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

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

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

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

In function notation:

\[ C[\text{skip}]\sigma \triangleq \sigma \]
\[ C[x := a]\sigma \triangleq \sigma[x \mapsto (A[a]\sigma)] \]
\[ C[c_1; c_2] \triangleq C[c_2] \circ C[c_1] \]
\[ C[\text{if } b \text{ then } c_1 \text{ else } c_2]\sigma \triangleq \begin{cases} C[c_1]\sigma & \text{if } B[b]\sigma = \text{true} \\ C[c_2]\sigma & \text{if } B[b]\sigma = \text{false} \end{cases} \]
Denotational Semantics of IMP

Commands:

\[ \mathcal{C}[\textbf{while } b \textbf{ do } c] \triangleq \]

\[ \{ (\sigma, \sigma) \mid (\sigma, \text{false}) \in B[b] \} \cup \]

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

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

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

Why?
Recursive Definitions

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

\[
\mathcal{C}[\text{while } b \text{ do } c] \not\equiv \{ (\sigma, \sigma) \mid (\sigma, \text{false}) \in B[b] \} \cup \{ (\sigma, \sigma') \mid (\sigma, \text{true}) \in B[b] \land \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[c] \land (\sigma'', \sigma') \in \mathcal{C}[\text{while } b \text{ do } c]) \}
\]

Why?

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

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

Solve for \( f \)?

\[ f(x) = x^2 \]

\[ f(0) = 0 \]

\[ f(1) = f(0) + 2 - 1 = 1 \]
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.

General takeaway: Recursive equations can have one, multiple, or no 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:

\[ f_0 = \emptyset \]

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

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

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

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

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

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

\[ \vdots \]
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 $F : A \rightarrow A$, 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 **while**

Now we can complete our denotational semantics:

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

Now we can complete our denotational semantics:

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

where

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

\textit{Approximation better?} \quad F

\[
A \cap B \quad A \cup B = B
\]