CS 4110

Programming Languages & Logics

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:

$$\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \qquad\qquad \langle \sigma, e \rangle \Downarrow \langle \sigma', n \rangle$$

Denotational Semantics

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

$$\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \qquad\qquad \langle \sigma, e \rangle \Downarrow \langle \sigma', n \rangle$$

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:

$$\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \qquad\qquad \langle \sigma, e \rangle \Downarrow \langle \sigma', n \rangle$$

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:

$$\mathcal{C}[\![c]\!] \in \textbf{Store} \rightharpoonup \textbf{Store}$$

IMP

Syntax

$$
\begin{array}{ll}
a \in \textbf{Aexp} & a ::= x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
b \in \textbf{Bexp} & b ::= \textbf{true} \mid \textbf{false} \mid a_1 < a_2 \\
c \in \textbf{Com} & c ::= \textbf{skip} \mid x := a \mid c_1 ; c_2 \\
 & \quad\;\; \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c
\end{array}
$$

IMP

Syntax

$$
\begin{array}{lrl}
a \in \textbf{Aexp} & a ::= & x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
b \in \textbf{Bexp} & b ::= & \textbf{true} \mid \textbf{false} \mid a_1 < a_2 \\
c \in \textbf{Com} & c ::= & \textbf{skip} \mid x := a \mid c_1 ; c_2 \\
 & \mid & \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c
\end{array}
$$

Semantic Domains

$$
\begin{array}{rcl}
\mathcal{C}[\![c]\!] & \in & \textbf{Store} \rightharpoonup \textbf{Store} \\
\mathcal{A}[\![a]\!] & \in & \textbf{Store} \rightharpoonup \textbf{Int} \\
\mathcal{B}[\![b]\!] & \in & \textbf{Store} \rightharpoonup \textbf{Bool}
\end{array}
$$

Why partial functions?

Notational Conventions

Convention #1: Represent functions $f : A \rightharpoonup B$ 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 $\mathcal{C}[\![\cdot]\!]$ is the denotation function, the equation $\mathcal{C}[\![c]\!] = S$ gives its definition for the command c.

Notational Conventions

Convention #1: Represent functions $f : A \rightharpoonup B$ 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 $\mathcal{C}[\![\cdot]\!]$ is the denotation function, the equation $\mathcal{C}[\![c]\!] = S$ gives its definition for the command c.

Applying this notation twice, $\mathcal{C}[\![c]\!]\sigma = \sigma'$ gives the value for the $\mathcal{C}[\![c]\!]$ function at σ.

Denotational Semantics of IMP

Arithmetic expressions:

$$\mathcal{A}[\![n]\!] \triangleq$$
$$\{(\sigma, n)\}$$

Denotational Semantics of IMP

Arithmetic expressions:

$$\mathcal{A}[\![n]\!] \triangleq \{(\sigma, n)\}$$

$$\mathcal{A}[\![x]\!] \triangleq \{(\sigma, \sigma(x))\}$$

Denotational Semantics of IMP

Arithmetic expressions:

$$\mathcal{A}[\![n]\!] \triangleq$$
$$\{(\sigma, n)\}$$

$$\mathcal{A}[\![x]\!] \triangleq$$
$$\{(\sigma, \sigma(x))\}$$

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

$$\mathcal{A}[\![a_1 \times a_2]\!] \triangleq$$
$$\{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \land (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \land n = n_1 \times n_2\}$$

Denotational Semantics of IMP

Boolean expressions:

$$\mathcal{B}[\![\textbf{true}]\!] \triangleq \{(\sigma, \textbf{true})\}$$

Denotational Semantics of IMP

Boolean expressions:

$$\mathcal{B}[\![\textbf{true}]\!] \triangleq$$
$$\{(\sigma, \textbf{true})\}$$

$$\mathcal{B}[\![\textbf{false}]\!] \triangleq$$
$$\{(\sigma, \textbf{false})\}$$

Denotational Semantics of IMP

Boolean expressions:

$$\mathcal{B}[\![\textbf{true}]\!] \triangleq \{(\sigma, \textbf{true})\}$$

$$\mathcal{B}[\![\textbf{false}]\!] \triangleq \{(\sigma, \textbf{false})\}$$

$$\mathcal{B}[\![a_1 < a_2]\!] \triangleq \{(\sigma, \textbf{true}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 < n_2\} \cup \{(\sigma, \textbf{false}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 \geq n_2\}$$

Denotational Semantics of IMP

Or, using the function-style notation:

$$\begin{aligned}
\mathcal{A}[\![n]\!]\sigma &\triangleq n \\
\mathcal{A}[\![x]\!]\sigma &\triangleq \sigma(x) \\
\mathcal{A}[\![a_1 + a_2]\!]\sigma &\triangleq \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{A}[\![a_1 \times a_2]\!]\sigma &\triangleq \mathcal{A}[\![a_1]\!]\sigma \times \mathcal{A}[\![a_2]\!]\sigma
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![\mathbf{true}]\!]\sigma &\triangleq \mathbf{true} \\
\mathcal{B}[\![\mathbf{false}]\!]\sigma &\triangleq \mathbf{false} \\
\mathcal{B}[\![a_1 < a_2]\!]\sigma &\triangleq \begin{cases} \mathbf{true} & \text{if } \mathcal{A}[\![a_1]\!]\sigma < \mathcal{A}[\![a_2]\!]\sigma \\ \mathbf{false} & \text{otherwise} \end{cases}
\end{aligned}$$

Denotational Semantics of IMP

Commands:

$$\mathcal{C}[\![\textbf{skip}]\!] \triangleq \{(\sigma, \sigma)\}$$

Denotational Semantics of IMP

Commands:

$$\mathcal{C}[\![\textbf{skip}]\!] \triangleq \{(\sigma, \sigma)\}$$

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

Denotational Semantics of IMP

Commands:

$$\mathcal{C}[\![\mathbf{skip}]\!] \triangleq \{(\sigma, \sigma)\}$$

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

$$\mathcal{C}[\![c_1 ; c_2]\!] \triangleq \{(\sigma, \sigma') \mid \exists \sigma''. \, ((\sigma, \sigma'') \in \mathcal{C}[\![c_1]\!] \wedge (\sigma'', \sigma') \in \mathcal{C}[\![c_2]\!])\}$$

Denotational Semantics of IMP

Commands:

$$\mathcal{C}[\![\textbf{skip}]\!] \triangleq \{(\sigma, \sigma)\}$$

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

$$\mathcal{C}[\![c_1; c_2]\!] \triangleq \{(\sigma, \sigma') \mid \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[\![c_1]\!] \wedge (\sigma'', \sigma') \in \mathcal{C}[\![c_2]\!])\}$$

$$\begin{aligned}\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] \triangleq\ & \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \cup \\ & \{(\sigma, \sigma') \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\}\end{aligned}$$

Denotational Semantics of IMP

In function notation:

$$\mathcal{C}[\![\textbf{skip}]\!]\sigma \triangleq \sigma$$

$$\mathcal{C}[\![x := a]\!]\sigma \triangleq \sigma[x \mapsto (\mathcal{A}[\![a]\!]\sigma)]$$

$$\mathcal{C}[\![c_1; c_2]\!] \triangleq \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!]$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma \triangleq \begin{cases} \mathcal{C}[\![c_1]\!]\sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \textbf{true} \\ \mathcal{C}[\![c_2]\!]\sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \textbf{false} \end{cases}$$

Denotational Semantics of IMP

Commands:

$$
\begin{aligned}
&\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \triangleq \\
&\qquad \{(\sigma, \sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \;\cup \\
&\qquad \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \land \exists \sigma''.\, ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \;\land \\
&\qquad\qquad\qquad (\sigma'', \sigma') \in \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])\}
\end{aligned}
$$

Recursive Definitions

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

$$
\begin{aligned}
&\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \triangleq \\
&\qquad \{(\sigma, \sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\
&\qquad \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \land \exists \sigma''. \, ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \land \\
&\qquad\qquad\qquad (\sigma'', \sigma') \in \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])\}
\end{aligned}
$$

Why?

Recursive Definitions

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

$$
\begin{aligned}
&\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \triangleq \\
&\qquad \{(\sigma, \sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\
&\qquad \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma''.\, ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge \\
&\qquad\qquad\qquad (\sigma'', \sigma') \in \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])\}
\end{aligned}
$$

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:

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

Solving Recursive Equations

Can build a solution by taking successive approximations:

$$
\begin{aligned}
f_0 &= \emptyset \\
f_1 &= \begin{cases} 0 & \text{if } x = 0 \\ f_0(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0)\} \\
f_2 &= \begin{cases} 0 & \text{if } x = 0 \\ f_1(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0),(1,1)\}
\end{aligned}
$$

Solving Recursive Equations

Can build a solution by taking successive approximations:

$$
\begin{aligned}
f_0 &= \emptyset \\
f_1 &= \begin{cases} 0 & \text{if } x = 0 \\ f_0(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0)\} \\
f_2 &= \begin{cases} 0 & \text{if } x = 0 \\ f_1(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0), (1,1)\} \\
f_3 &= \begin{cases} 0 & \text{if } x = 0 \\ f_2(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0), (1,1), (2,4)\} \\
&\vdots
\end{aligned}
$$

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}:

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

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 say that $a \in A$ is a fixed point of F if and only if $F(a) = a$.

Notation: Write $a = \mathrm{fix}(F)$ 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:

$$\begin{aligned} f &= \mathrm{fix}(F) \\ &= f_0 \cup f_1 \cup f_2 \cup f_3 \cup \ldots \\ &= \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup F(F(F(\emptyset))) \cup \ldots \\ &= \bigcup_{i \geq 0}^{\infty} F^i(\emptyset) \end{aligned}$$

Denotational Semantics for **while**

Now we can complete our denotational semantics:

$$\mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!] \triangleq \mathrm{fix}(F)$$

Denotational Semantics for **while**

Now we can complete our denotational semantics:

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \triangleq \text{fix}(F)$$

where

$$\begin{aligned} F(f) \triangleq \{(\sigma, \sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \\ \exists \sigma''.\, ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'', \sigma') \in f)\} \end{aligned}$$