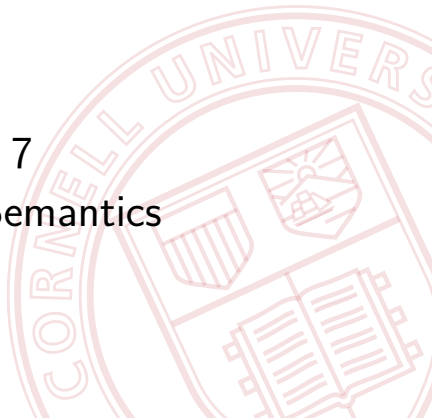


CS 4110

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:

$$\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle$$

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

Denotational Semantics

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

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

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:

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

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:

$$\mathcal{C} \llbracket c \rrbracket \in \mathbf{Store} \rightarrow \mathbf{Store}$$

IMP

Syntax

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

IMP

Syntax

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

Semantic Domains

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

Why partial functions?

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 $\mathcal{C}[\![\cdot]\!]$ is the denotation function, the equation $\mathcal{C}[\![c]\!] = S$ gives its definition for the command c .

Notational Conventions

Convention #1: Represent functions $f: A \rightarrow B$ 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 $\mathcal{C}[\![\cdot]\!]$ 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:

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

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

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

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

Denotational Semantics of IMP

Boolean expressions:

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

Denotational Semantics of IMP

Boolean expressions:

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

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

Denotational Semantics of IMP

Boolean expressions:

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

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

$$\begin{aligned} \mathcal{B}[a_1 < a_2] \triangleq & \{(\sigma, \mathbf{true}) \mid (\sigma, n_1) \in \mathcal{A}[a_1] \wedge (\sigma, n_2) \in \mathcal{A}[a_2] \wedge n_1 < n_2\} \cup \\ & \{(\sigma, \mathbf{false}) \mid (\sigma, n_1) \in \mathcal{A}[a_1] \wedge (\sigma, n_2) \in \mathcal{A}[a_2] \wedge n_1 \geq n_2\} \end{aligned}$$

Denotational Semantics of IMP

Or, using the function-style notation:

$$\mathcal{A}[[n]]\sigma \triangleq n$$

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

$$\mathcal{A}[[a_1 + a_2]]\sigma \triangleq \mathcal{A}[[a_1]]\sigma + \mathcal{A}[[a_2]]\sigma$$

$$\mathcal{A}[[a_1 \times a_2]]\sigma \triangleq \mathcal{A}[[a_1]]\sigma \times \mathcal{A}[[a_2]]\sigma$$

$$\mathcal{B}[[\mathbf{true}]]\sigma \triangleq \mathbf{true}$$

$$\mathcal{B}[[\mathbf{false}]]\sigma \triangleq \mathbf{false}$$

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

Denotational Semantics of IMP

Commands:

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

Denotational Semantics of IMP

Commands:

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

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

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:

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

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

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

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

Denotational Semantics of IMP

In function notation:

$$\mathcal{C}[\text{skip}]\sigma \triangleq \sigma$$

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

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

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

Denotational Semantics of IMP

Commands:

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

Recursive Definitions

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

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

Why?

Recursive Definitions

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

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

Why?

It expresses $\mathcal{C}[\mathbf{while\ } b \mathbf{\ do\ } c]$ 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:

$$f_0 = \emptyset$$

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

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

Solving Recursive Equations

Can build a solution by taking successive approximations:

$$f_0 = \emptyset$$

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

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

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

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:

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

Denotational Semantics for **while**

Now we can complete our denotational semantics:

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

Denotational Semantics for **while**

Now we can complete our denotational semantics:

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

where

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