

CS 4110

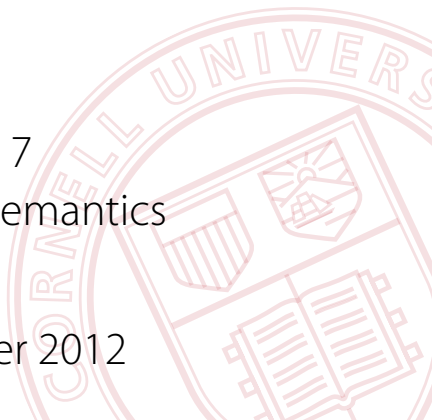
# Programming Languages & Logics

---

Lecture 7

Denotational Semantics

12 September 2012



# Announcements

---

- Homework #2 out
- Foster office hours today 11am-12pm
- Please make sure you have an iClicker or a subscription!

# 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

# Denotational Semantics

---

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

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

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

# Denotational Semantics

---

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

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

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

A **denotational semantics** models *what* a program computes.

# Denotational Semantics

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

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

A **denotational semantics** models *what* a program computes.

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

$$\mathcal{C}[[c]] \in \mathbf{Store} \rightarrow \mathbf{Store}$$

## Syntax

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

# 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



# 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

$\mathcal{C}[[c]] \in \mathbf{Store} \rightarrow \mathbf{Store}$

Why partial functions?

## 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

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

Why partial functions?

# 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, 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  $\mathcal{C}[[c]] = S$  gives the definition of the denotation function  $\mathcal{C}[[\cdot]]$  on command  $c$ .

# Denotational Semantics of IMP

---

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

# Denotational Semantics of IMP

---

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

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

# Denotational Semantics of IMP

---

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

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

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

# Denotational Semantics of IMP

---

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



# Denotational Semantics of IMP

---

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

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

# Denotational Semantics of IMP

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

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

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

# Denotational Semantics of IMP

---

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

# Denotational Semantics of IMP

---

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

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

# Denotational Semantics of IMP

$$\mathcal{C}[\mathbf{skip}] =$$

$$\{(\sigma, \sigma)\}$$

$$\mathcal{C}[x := a] =$$

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

$$\mathcal{C}[c_1; c_2] =$$

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

# Denotational Semantics of IMP

$$\mathcal{C}[\mathbf{skip}] =$$

$$\{(\sigma, \sigma)\}$$

$$\mathcal{C}[x := a] =$$

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

$$\mathcal{C}[c_1; c_2] =$$

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

$$\mathcal{C}[\mathbf{if } b \mathbf{ then } c_1 \mathbf{ else } c_2] =$$

$$\{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_1]\} \cup$$

$$\{(\sigma, \sigma') \mid (\sigma, \mathbf{false}) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_2]\}$$

# Denotational Semantics of IMP

$$\mathcal{C}[\mathbf{skip}] =$$

$$\{(\sigma, \sigma)\}$$

$$\mathcal{C}[x := a] =$$

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

$$\mathcal{C}[c_1; c_2] =$$

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

$$\mathcal{C}[\mathbf{if } b \mathbf{ then } c_1 \mathbf{ else } c_2] =$$

$$\{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_1]\} \cup$$

$$\{(\sigma, \sigma') \mid (\sigma, \mathbf{false}) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_2]\}$$

$$\mathcal{C}[\mathbf{while } b \mathbf{ do } c] =$$

$$\{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[b]\} \cup$$

$$\{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[b] \wedge \exists \sigma''. ((\sigma, \sigma'') \in \mathcal{C}[c] \wedge (\sigma'', \sigma') \in \mathcal{C}[\mathbf{while } b \mathbf{ do } c])\}$$

# Recursive Definitions

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

$$\mathcal{C}[\mathbf{while\ } b \mathbf{\ do\ } c] =$$

$$\mathcal{C}[\mathbf{while\ } b \mathbf{\ do\ } c] =$$

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

Why?



# Recursive Definitions

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

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

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 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:

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

# Denotational Semantics for **while**

---

Now we can complete our denotational semantics:

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

# Denotational Semantics for **while**

Now we can complete our denotational semantics:

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

where

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