Computer Science 131
Programming Languages

November 28, 2000

Denotational Semantics

Review

- Operational Semantics
  - Describes interpreter for programs
  - Models execution in a computer
    - Can choose how detailed to make the model
  - We have seen several flavors; others are possible

$$\mathbf{e} \rightarrow \mathbf{e'}$$

$$(\mathbf{e},\mathbf{M}) \rightarrow (\mathbf{e'},\mathbf{M'})$$

$$\mathbf{e} \Downarrow \mathbf{v}$$

$$\mathbf{e}, \rho \Downarrow \mathbf{v}$$

Denotational Semantics

- Every program has a *meaning* or *denotation*.
  - The meaning is a mathematical object
    - e.g., an integer or a real number or a function or …
  - In typed languages:
    - Every type **`t`** corresponds to a set
    - The meaning of a program of type **`t`** is a member of the corresponding set.
  - In untyped languages:
    - Everything is the same type
    - Meanings taken from a single set.

Denotational Semantics

- Denotations are defined *compositionally*.
  - The meaning of $\mathbf{e_1+e_2}$ can be determined from the meanings of $\mathbf{e_1}$ and of $\mathbf{e_2}$.
  - The meaning of **`while b do e`** can be determined from the meanings of **`b`** and **`e`**.

Notation

- We write $[\![\mathbf{e}]\!]$ to represent the meaning of **`e`**.
  - That is, $[\![\cdot]\!]$ is a function from expressions to meanings.

- We also write $[\![\mathbf{t}]\!]$ to represent the set of possible meanings for expressions of type **`t`**.

- We will use the following mathematical sets
  - The set $\mathbb{Z}$ of integers
  - The set $\mathbb{B}$ `= {true,false}.`

Operational vs. Denotational

- When are two expressions equal?
- Operational equivalence: if replacing one with the other *in a complete program* does not change the answer.
  - Usually only consider complete programs that evaluate to an "observable" result (i.e., a base type).
- Denotational equivalence: if they have the same meaning.
  - Frequently easier to reason about.

Simple Integer Expressions

- Abstract Syntax

v ::= <u>n</u>	(values)
e ::= v | $\mathbf{e_1}$ + $\mathbf{e_2}$	(expressions)
p ::= e	(programs)
t ::= Int	(types)

- What does a program in this language mean?

Denotational Semantics

$$[\![\mathbf{Int}]\!] = \mathbb{Z}$$

$$[\![\underline{\mathbf{n}}]\!] = n$$

$$[\![\mathbf{e_1+e_2}]\!] = [\![\mathbf{e_1}]\!] + [\![\mathbf{e_2}]\!]$$

Soundness

- A denotational semantics is *sound* with respect to the operational semantics if denotational equivalence implies operational equivalence.
  - Soundness = "does not equate any observably different programs"
  - Might still give different meanings to two expressions that always act the same.

Adequacy and Full Abstraction

- Denotational semantics is *adequate* if:
  - for all **`M`** and **`N`**, **`M`** evaluates to result **`N`** under the operational semantics iff **`M`** and **`N`** are denotationally equivalent.

- It is *fully abstract* if operational and denotational semantics agree for arbitrary expressions.

Including Booleans

- Abstract Syntax

$$
\begin{array}{lll}
\mathbf{v} ::= \underline{\mathbf{n}} \mid \underline{\mathbf{tt}} \mid \underline{\mathbf{ff}} & & \text{(values)} \\
\mathbf{e} ::= \mathbf{v} \mid \mathbf{e}_1 + \mathbf{e}_2 \mid \mathbf{e}_1 < \mathbf{e}_2 & & \text{(expressions)} \\
\qquad \mid \mathbf{if}\ \mathbf{e}_1\ \mathbf{then}\ \mathbf{e}_2\ \mathbf{else}\ \mathbf{e}_3 & & \\
\mathbf{p} ::= \mathbf{e} & & \text{(programs)} \\
\mathbf{t} ::= \mathbf{Int} \mid \mathbf{Bool} & & \text{(types)}
\end{array}
$$

Choice Point

- What do we do about $\underline{\mathtt{3}}\mathtt{+}\underline{\mathtt{tt}}$?
- Two choices
  1. Only give meanings to well-typed expressions.
  2. Give meanings to all expressions
     - Then $[\![\cdot]\!]$ is function from expressions to $\mathbb{Z} \cup \mathbb{B} \cup \{\mathtt{typeerror}\}$

Definition

- For simplicity, we will assume well-typedness.
- Then

$$
\begin{aligned}
[\![\underline{\mathbf{n}}]\!] &= \mathtt{n} \\
[\![\underline{\mathbf{tt}}]\!] &= \mathtt{true} \\
[\![\underline{\mathbf{ff}}]\!] &= \mathtt{false} \\
[\![\mathbf{e_1+e_2}]\!] &= [\![\mathbf{e_1}]\!] + [\![\mathbf{e_2}]\!] \\
[\![\mathbf{e_1<e_2}]\!] &= [\![\mathbf{e_1}]\!] < [\![\mathbf{e_2}]\!] \\
[\![\mathbf{if}\ \mathbf{e_1}\ \mathbf{then}\ \mathbf{e_2}\ \mathbf{else}\ \mathbf{e_3}]\!] &= \\
&\quad \mathtt{if}\ [\![\mathbf{e_1}]\!]\ \mathtt{then}\ [\![\mathbf{e_2}]\!]\ \mathtt{else}\ [\![\mathbf{e_3}]\!]
\end{aligned}
$$

Alternate Definition

- Could take *all* meanings from $\mathbb{Z}$:

$$
\begin{aligned}
[\![\underline{\mathbf{n}}]\!] &= \mathtt{n} \\
[\![\underline{\mathbf{tt}}]\!] &= \mathtt{0} \\
[\![\underline{\mathbf{ff}}]\!] &= \mathtt{1} \\
[\![\mathbf{e_1+e_2}]\!] &= [\![\mathbf{e_1}]\!] + [\![\mathbf{e_2}]\!] \\
[\![\mathbf{e_1<e_2}]\!] &= \mathtt{if}\ [\![\mathbf{e_1}]\!] < [\![\mathbf{e_2}]\!]\ \mathtt{then\ 0\ else\ 1} \\
[\![\mathbf{if\ e_1\ then\ e_2\ else\ e_3}]\!] &= \\
&\quad \mathtt{if}\ ([\![\mathbf{e_1}]\!] = \mathtt{0})\ \mathtt{then}\ [\![\mathbf{e_2}]\!]\ \mathtt{else}\ [\![\mathbf{e_3}]\!]
\end{aligned}
$$

Including variables

- Abstract Syntax

$$
\begin{array}{lll}
\mathbf{v} ::= \underline{\mathbf{n}} \mid \underline{\mathbf{tt}} \mid \underline{\mathbf{ff}} & & \text{(values)} \\
\mathbf{e} ::= \mathbf{v} \mid \mathbf{e_1} + \mathbf{e_2} \mid \mathbf{e_1} < \mathbf{e_2} & & \text{(expressions)} \\
\quad\quad \mid \mathbf{if}\ \mathbf{e_1}\ \mathbf{then}\ \mathbf{e_2}\ \mathbf{else}\ \mathbf{e_3} & & \\
\quad\quad \mid \mathbf{x} & & \\
\quad\quad \mid \mathbf{let}\ \mathbf{x}\ \mathbf{be}\ \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2} & & \\
\mathbf{p} ::= \mathbf{e} & & \text{(programs)} \\
\mathbf{t} ::= \mathbf{Int} \mid \mathbf{Bool} & & \text{(types)}
\end{array}
$$

Meanings of Open Expressions

- Now programs have subexpressions that contain free variables.
  - Recall: denotational semantics is compositional
- What is the meaning of the expression **x+7** ?
  - Given a value for **x**, return **7** more than that.
  - i.e., meaning is a *function* of the values of free variables.
- Let $[\![\Gamma]\!]$ denote the set of environments suitable for Γ.

$$\{\rho \mid \forall \mathbf{x} \in \mathrm{dom}(\Gamma).\ \rho(\mathbf{x}) \in [\![\Gamma(\mathbf{x})]\!]\}$$

- Then we expect

if

Definition

$\llbracket \underline{\mathbf{n}} \rrbracket = \lambda\rho.\mathrm{n}$

$\llbracket \mathbf{e_1+e_2} \rrbracket = \lambda\rho.(\llbracket \mathbf{e_1} \rrbracket\rho + \llbracket \mathbf{e_2} \rrbracket\rho)$

$\llbracket \mathbf{if\ e_1\ then\ e_2\ else\ e_3} \rrbracket =$

$\quad l\rho.(\mathrm{if}\ \llbracket \mathbf{e_1} \rrbracket\rho\ \mathrm{then}\ \llbracket \mathbf{e_2} \rrbracket\rho\ \mathrm{else}\ \llbracket \mathbf{e_3} \rrbracket\rho)$

...

$\llbracket \mathbf{x} \rrbracket = \lambda\rho.\ \rho(\mathrm{x})$

$\llbracket \mathbf{let\ x\ be\ e_1\ in\ e_2} \rrbracket = \lambda\rho.\ \llbracket \mathbf{e_2} \rrbracket\ (\rho, \mathbf{x}$ ® $\llbracket \mathbf{e_1} \rrbracket\rho)$

Definition

$$[\![\underline{\mathbf{n}}]\!]\rho \quad = \quad n$$

$$[\![\mathbf{e_1+e_2}]\!]\rho \quad = \quad [\![\mathbf{e_1}]\!]\rho + [\![\mathbf{e_2}]\!]\rho$$

$$[\![\mathbf{if}\ \mathbf{e_1}\ \mathbf{then}\ \mathbf{e_2}\ \mathbf{else}\ \mathbf{e_3}]\!]\rho =$$

$$\quad \text{if } [\![\mathbf{e_1}]\!]\rho \text{ then } [\![\mathbf{e_2}]\!]\rho \text{ else } [\![\mathbf{e_3}]\!]\rho$$

...

$$[\![\mathbf{x}]\!]\rho \quad = \rho(\mathbf{x})$$

$$[\![\mathbf{let}\ \mathbf{x}\ \mathbf{be}\ \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2}]\!] = [\![\mathbf{e_2}]\!](\rho, \mathbf{x}\circledR[\![\mathbf{e_1}]\!]\rho)$$

Including Functions

- Abstract Syntax

$$
\begin{array}{lll}
\mathtt{v} ::= \underline{\mathtt{n}} \mid \underline{\mathtt{tt}} \mid \underline{\mathtt{ff}} & & \text{(values)} \\
\quad\quad \mid \mathtt{fn\ x{:}t => e} & & \\
\mathtt{e} ::= \mathtt{v} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{e}_1 < \mathtt{e}_2 & & \text{(expressions)} \\
\quad\quad \mid \mathtt{if\ e}_1\ \mathtt{then\ e}_2\ \mathtt{else\ e}_3 & & \\
\quad\quad \mid \mathtt{x} \mid \mathtt{let\ x\ be\ e}_1\ \mathtt{in\ e}_2 & & \\
\quad\quad \mid \mathtt{e}_1\mathtt{e}_2 & & \\
\mathtt{p} ::= \mathtt{e} & & \text{(programs)} \\
\mathtt{t} ::= \mathtt{Int} \mid \mathtt{Bool} \mid \mathtt{t}_1\mathtt{->t}_2 & & \text{(types)}
\end{array}
$$

Definition

- Define

$$[\![\mathbf{t_1\text{->}t_2}]\!] \quad = \quad \{\mathtt{f} : [\![\mathbf{t_1}]\!] \to [\![\mathbf{t_2}]\!]\}$$

- Then

$$[\![\mathbf{fn\ x\ =>\ e}]\!]\rho \quad = \quad \lambda z{:}[\![\mathbf{t_1}]\!].\ [\![\mathbf{e}]\!](\rho,\mathbf{x}\to z)$$

$$[\![\mathbf{e_1 e_2}]\!]\rho \quad = \quad ([\![\mathbf{e_1}]\!]\rho)([\![\mathbf{e_2}]\!]\rho)$$

Adding Divergence

$\mathbf{e ::= \ldots}$

$\quad\quad \mathbf{|\ \texttt{diverge}_t}$

where

$$\texttt{diverge}_t : \texttt{t}$$

and operationally

$$\texttt{diverge}_t \text{ ® } \texttt{diverge}_t$$

That is, $\texttt{diverge}_t$ never terminates.

Changes to the Setup

- We add a new item $\perp$ to represent the meaning of a non-terminating expression.
- If $G \vdash$ **`e : t`** then

  $$[\![\mathbf{e}]\!] : [\![G]\!] \text{ -> } ([\![\mathbf{t}]\!] \cup \{\perp\})$$

  where

  $$[\![\mathbf{Int}]\!] = \mathbb{Z}$$

  $$[\![\mathbf{Bool}]\!] = \mathbb{B}$$

  $$[\![\mathbf{t_1\text{->}t_2}]\!] = \{f : [\![\mathbf{t_1}]\!] \rightarrow ([\![\mathbf{t_2}]\!] \cup \{\perp\})\}$$

Definition

- Clearly

$$[\![\mathtt{\textbf{diverge}}_{\mathtt{\textbf{t}}}]\!] \quad = \quad \perp$$

- But the other equations need to change too.
  - What is the meaning of $\mathtt{\textbf{3 + diverge}}_{\mathtt{\textbf{Int}}}$?

Recursive Functions

- Abstract Syntax

$$
\begin{array}{lll}
\mathbf{v} ::= \underline{\mathbf{n}} \mid \underline{\mathbf{tt}} \mid \underline{\mathbf{ff}} & & \text{(values)} \\
\quad \mid \mathbf{fun}\ \mathbf{f(x{:}t_1){:}t_2}\ \mathbf{is}\ \mathbf{e} & & \\
\mathbf{e} ::= \mathbf{v} \mid \mathbf{e_1 + e_2} \mid \mathbf{e_1 < e_2} & & \text{(expressions)} \\
\quad \mid \mathbf{if}\ \mathbf{e_1}\ \mathbf{then}\ \mathbf{e_2}\ \mathbf{else}\ \mathbf{e_3} & & \\
\quad \mid \mathbf{x} \mid \mathbf{let}\ \mathbf{x}\ \mathbf{be}\ \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2} & & \\
\quad \mid \mathbf{e_1 e_2} & & \\
 & & \\
\mathbf{p} ::= \mathbf{e} & & \text{(programs)}
\end{array}
$$

First Try

$$[\![\mathbf{fun\ f(x{:}t_1){:}t_2\ is\ e}]\!]\rho \;\; \mathbf{=}$$

$$\lambda z{:}[\![\mathbf{t_1}]\!].\ [\![\mathbf{e}]\!](\rho, \mathbf{x}\to z, \mathbf{f}\to([\![\mathbf{fun\ f(x{:}t_1){:}t_2\ is\ e}]\!]\rho))$$

- But this "definition" is circular.
- Can we find a definition not in terms of

$$[\![\mathbf{fun\ f(x{:}t_1){:}t_2\ is\ e}]\!]\ ?$$

Second Try

$[\![$**fun f(x:t$_1$):t$_2$ is e**$]\!]\rho$ **:=**

the mathematical function g *such that*

g = λz:$[\![$**t$_1$**$]\!]$. $[\![$**e**$]\!]$(ρ,**x**$\to$z,**f**$\to$g))

- This is a little better
  - But it really says the same thing.
  - Why should such a g exist? Is it unique?
  - What about non-terminating functions?

Domain Theory

- Idea: the meanings of types will be special *partially-ordered* sets, called domains.
  - Ordered by information content.
- Meanings of programs will turn out to be *continuous*
  - Monotone (increasing)
  - Preserves limits
- Theorem: continuous functions have unique least fixed-points.
  - Chosen as the meaning of recursive/looping programs.