Computer Science 131
Programming Languages

November 28, 2000
Denotational Semantics
• 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

\[
e \rightarrow e' \]

\[
(e, M) \rightarrow (e', M')
\]

\[
e \Downarrow v
\]

\[
e, \rho \Downarrow v
\]
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 $\tau$ corresponds to a set
    - The meaning of a program of type $\tau$ 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 $e_1 + e_2$ can be determined from the meanings of $e_1$ and of $e_2$.
  - The meaning of \texttt{while b do e} can be determined from the meanings of \texttt{b} and \texttt{e}. 
Notation

• We write \([e]\) to represent the meaning of \(e\).
  - That is, \([\cdot]\) is a function from expressions to meanings.

• We also write \([t]\) 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 \(\mathbb{B} = \{\text{true}, \text{false}\}\).
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

\[ v ::= n \] (values)
\[ e ::= v \mid e_1 + e_2 \] (expressions)
\[ p ::= e \] (programs)
\[ t ::= \text{Int} \] (types)

• What does a program in this language mean?
Denotational Semantics

\[
[\text{Int}] = \mathbb{Z}
\]

\[
[n] = n
\]

\[
[e_1 + e_2] = [e_1] + [e_2]
\]
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

\[ v ::= n \mid tt \mid ff \] (values)

\[ e ::= v \mid e_1 + e_2 \mid e_1 < e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \] (expressions)

\[ p ::= e \] (programs)

\[ t ::= \text{Int} \mid \text{Bool} \] (types)
Choice Point

- What do we do about \( 3 + \texttt{tt} \)?
- Two choices
  1. Only give meanings to well-typed expressions.
  2. Give meanings to all expressions
     - Then \( \cdot \) is function from expressions to \( \mathbb{Z} \cup \mathbb{B} \cup \{\text{typeerror}\} \)
Definition

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

\[
\begin{align*}
[n] & = n \\
[tt] & = \text{true} \\
[ff] & = \text{false} \\
[e_1 + e_2] & = [e_1] + [e_2] \\
[e_1 < e_2] & = [e_1] < [e_2] \\
\text{[if } e_1 \text{ then } e_2 \text{ else } e_3 \text{]} & = \\
& \quad \text{if } [e_1] \text{ then } [e_2] \text{ else } [e_3]
\end{align*}
\]
Alternate Definition

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

\[
\begin{align*}
[n] &= n \\
[tt] &= 0 \\
[ff] &= 1 \\
[e_1 + e_2] &= [e_1] + [e_2] \\
[e_1 < e_2] &= \text{if } [e_1] < [e_2] \text{ then } 0 \text{ else } 1 \\
[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] &= \\
&\quad \text{if } ([e_1] = 0) \text{ then } [e_2] \text{ else } [e_3]
\end{align*}
\]
Including variables

• Abstract Syntax

\[
\begin{align*}
v & ::= \text{n} \mid \text{tt} \mid \text{ff} \quad \text{(values)} \\
e & ::= v \mid e_1 + e_2 \mid e_1 < e_2 \quad \text{(expressions)} \\
& \quad | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
& \quad | x \\
& \quad | \text{let } x \text{ be } e_1 \text{ in } e_2 \\
p & ::= e \quad \text{(programs)} \\
t & ::= \text{Int} \mid \text{Bool} \quad \text{(types)}
\end{align*}
\]
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 \( \Gamma \).
  \[
  \{ \rho \mid \forall x \in \text{dom}(\Gamma). \rho(x) \in [\Gamma(x)] \}
  \]

• Then we expect

if
Definition

\[ n = \lambda \rho. n \]
\[ e_1 + e_2 = \lambda \rho. ([e_1]_\rho + [e_2]_\rho) \]
\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 = \lambda \rho. (\text{if } [e_1]_\rho \text{ then } [e_2]_\rho \text{ else } [e_3]_\rho) \]
\[ \ldots \]

\[ x = \lambda \rho. \rho(x) \]
\[ \text{let } x \text{ be } e_1 \text{ in } e_2 = \lambda \rho. [e_2](\rho, x \rightarrow [e_1]_\rho) \]
Definition

\[
\begin{align*}
[n]_\rho &= n \\
[e_1 + e_2]_\rho &= (e_1)_\rho + (e_2)_\rho \\
[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]_\rho &= \\
&\quad \text{if } (e_1)_\rho \text{ then } (e_2)_\rho \text{ else } (e_3)_\rho \\
&\quad \ldots
\end{align*}
\]

\[
\begin{align*}
[x]_\rho &= \rho(x) \\
[\text{let } x \text{ be } e_1 \text{ in } e_2] &= (e_2)(\rho, x \mapsto (e_1)_\rho)
\end{align*}
\]
Including Functions

• Abstract Syntax

\[
\begin{align*}
  v & ::= \mathsf{\underline{n}} \mid \mathsf{\underline{tt}} \mid \mathsf{\underline{ff}} \quad \text{(values)} \\
  & \quad \mid \mathsf{fn} \; x : t \Rightarrow e \\
  e & ::= v \mid e_1 + e_2 \mid e_1 < e_2 \quad \text{(expressions)} \\
  & \quad \mid \text{if} \; e_1 \; \text{then} \; e_2 \; \text{else} \; e_3 \\
  & \quad \mid x \mid \text{let} \; x \; \text{be} \; e_1 \; \text{in} \; e_2 \\
  & \quad \mid e_1 e_2 \\
  p & ::= e \quad \text{(programs)} \\
  t & ::= \text{Int} \mid \text{Bool} \mid t_1 \rightarrow t_2 \quad \text{(types)}
\end{align*}
\]
Definition

• Define

\[
[t_1 \rightarrow t_2] = \{ f : \{ t_1 \} \rightarrow \{ t_2 \} \}
\]

• Then

\[
[\text{fn } x \Rightarrow e] \rho = \lambda z : \{ t_1 \}. \{ e \}(\rho, x \rightarrow z)
\]

\[
[e_1 e_2] \rho = ([e_1] \rho)([e_2] \rho)
\]
Adding Divergence

e ::= ... | diverge_t

where

diverge_t : t

and operationally

diverge_t → diverge_t

That is, \text{diverge}_t never terminates.
Changes to the Setup

• We add a new item $\perp$ to represent the meaning of a non-terminating expression.

• If $\Gamma \vdash e : t$ then

\[
[e] : \Gamma \rightarrow ([t] \cup \{\perp\})
\]

where

\[
\begin{align*}
[\text{Int}] &= \mathbb{Z} \\
[\text{Bool}] &= \mathbb{B} \\
[t_1 \rightarrow t_2] &= \{ f : [t_1] \rightarrow ([t_2] \cup \{\perp\}) \}
\end{align*}
\]
Definition

- Clearly
  \[
  [\text{diverge}_t] = \bot
  \]

- But the other equations need to change too.
  - What is the meaning of \(3 + \text{diverge}_{\text{Int}}\)?
Recursive Functions

• Abstract Syntax

\[ v ::= n \mid \text{tt} \mid \text{ff} \quad \text{(values)} \]
\[ v ::= \text{fun } f(x:t_1):t_2 \text{ is } e \]
\[ e ::= v \mid e_1 + e_2 \mid e_1 < e_2 \quad \text{(expressions)} \]
\[ e ::= \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]
\[ e ::= x \mid \text{let } x \text{ be } e_1 \text{ in } e_2 \]
\[ e ::= e_1 e_2 \]
\[ p ::= e \quad \text{(programs)} \]
First Try

\[
[\text{fun } f(x: t_1): t_2 \text{ is } e]\rho = \\
\lambda z: [t_1]. \ [e](\rho, x \mapsto z, f \mapsto ([\text{fun } f(x: t_1): t_2 \text{ is } e]\rho))
\]

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

\[
[\text{fun } f(x: t_1): t_2 \text{ is } e]
\]
Second Try

\[
\text{fun } f(x:t_1):t_2 \text{ is } e]\rho :=
\]

the mathematical function \(g\) such that
\[
g = \lambda z:[t_1]. \ [e](\rho, x \rightarrow z, f \rightarrow g)
\]

- 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.