Announcements

- Homework 2 is out, due Mon Feb 9
- OCaml part like Homework 1
- Should be easier this time!

- Course feedback at the end of class

---

Recall: Proving Com Eval. is Deterministic

\[
\begin{align*}
&\text{If } D: <c, \sigma> \Downarrow \sigma' \text{ and } D': <c, \sigma> \Downarrow \sigma'', \text{ then } \sigma' = \sigma''. \\
&\text{Where do we get stuck?}
\end{align*}
\]

- Assume we have proven Bexp and Aexp evaluation deterministic
- Cannot use induction on the structure of the command \(c\).
  - Case \(c = \text{while} \ b \text{ do } c_1\), Subcase:
    - \(b \Rightarrow \text{true}\):
      \[
      \begin{align*}
      D &:: <b, \sigma> \Downarrow \text{true} \quad <c_1, \sigma> \Downarrow \sigma_1 \\
      & <\text{while} \ b \text{ do } c_1, \sigma> \Downarrow \sigma' \\
      & D' :: <b, \sigma> \Downarrow \text{true} \quad <c_1, \sigma> \Downarrow \sigma_1' \\
      & <\text{while} \ b \text{ do } c_1, \sigma> \Downarrow \sigma'' \\
      \end{align*}
      \]

---

**Induction on the Structure of Derivations**

- To prove that for all derivations \(D\) of a judgment, property \(P\) holds

- For each derivation rule of the form
  \[
  H_1 \ldots H_n \quad \vdash C
  \]
  - Assume \(P\) holds for derivations of \(H_i (i = 1..n)\)
  - Prove the the property holds for the derivation obtained from the derivations of \(H_i\) using the given rule.
Proving Com Evaluation is Deterministic

If $D :: \langle c, \sigma \rangle \Downarrow \sigma'$ and $D' :: \langle c, \sigma \rangle \Downarrow \sigma''$, then $\sigma' = \sigma''$.

- Case
  - By inversion and determinism of Bexp evaluation, $D'$ uses the rule for "while true" with subderivations $D'_1 :: \langle c, \sigma \rangle \Downarrow \sigma''_1$ and $D'_2 :: \langle W, \sigma''_1 \rangle \Downarrow \sigma''_1$
  - By induction hypothesis on $D_1$ (with $D'_1$): $\sigma_1 = \sigma''_1$
  - By induction hypothesis on $D_2$ (with $D'_2$): $\sigma_2 = \sigma''_2$

Try to do this on a piece of paper. In a moment, we'll have some lucky winners come on down!

Summary: Induction on Derivations

- If you must prove $\forall x \in A. P(x) \Rightarrow Q(x)$
  - with $A$ inductively defined and $P(x)$ rule-defined
  - we pick arbitrary $x \in A$ and $D :: P(x)$
  - we could do induction on both facts
    - $x \in A$ leads to induction on the structure of $x$
      - $c \in \text{Com}$
    - $D :: P(x)$ leads to induction on the structure of $D$
      - $D :: \langle c, \sigma \rangle \Downarrow \sigma'$
  - Generally, the induction on the structure of the derivation is more powerful and a safer bet

End of Review

On to Denotational Semantics

Also called:
- fixed-point semantics
- mathematical semantics
- Scott-Strachey semantics

Dueling Semantics

- Operational semantics is
  - simple
  - of many flavors (big-step, small-step, more or less abstract)
  - basis for many proofs about a language
    - commonly used in the real (modern research) world
    - not compositional
  - Denotational semantics is
    - mathematical (the meaning of a syntactic expression is a mathematical object)
    - compositional

Recall truth vs. provability discussion
Typical Student Reaction to Denotational Semantics

Denotational Semantics Learning Goals

• Key features of denotational semantics
  - Compositional
  - Meaning is a "math object"
  - When to use DS?

• DS uses ⊥ ("bottom") to mean non-termination
• DS uses fixed points and domains to handle while
  - This is the tricky bit

Remember SLAM and BLAST?

DS in the Real World

• Ada was formally specified with it

• Nice when you want to compare a program in Language 1 to a program in Language 2

Foreshadowing

• Denotational semantics assigns meanings to programs
  • The meaning will be a mathematical object
    - A number ∈ ℤ
    - A boolean ∈ {true, false}
    - A state transformer : Σ → (Σ ∪ {⊥})
  • The meaning will be determined compositionally
    - Denotation of a command is based on the denotations of its immediate sub-commands (≠ more than merely syntax-directed)

New Notation

[ ] = "means" or "denotes"

• Examples:
  [foo] = "denotation of foo"
  [3 < 5] = true
  [3 + 5] = 8

• Sometimes we write
  - A[] for Aexp, B[] for Bexp, C[] for Com

Rough Idea of Denotational Semantics

• The meaning of an arithmetic expression e in state σ is a number n
  • So, we try to define A[e] as a function that maps the current state to an integer:
    A[ ] : Aexp → (Σ → ℤ)
  • The meaning of boolean expressions is defined in a similar way
    B[ ] : Bexp → (Σ → {true, false})
  • All of these denotational functions are total
    - Defined for all syntactic elements
    - For other languages it might be convenient to define the semantics only for well-typed elements
The document contains sections titled "Denotational Semantics of Arithmetic Expressions" and "Denotational Semantics of Boolean Expressions.

**Denotational Semantics of Arithmetic Expressions**
- We inductively define a function
  \[ A[\cdot] : \text{Aexp} \rightarrow (\Sigma \rightarrow \mathbb{Z}) \]
  - \[ A[n]\sigma = \text{the integer denoted by literal } n \]
  - \[ A[x]\sigma = \sigma(x) \]
  - \[ A[e_1 + e_2]\sigma = A[e_1]\sigma + A[e_2]\sigma \]
  - \[ A[e_1 - e_2]\sigma = A[e_1]\sigma - A[e_2]\sigma \]
  - \[ A[e_1 \times e_2]\sigma = A[e_1]\sigma \times A[e_2]\sigma \]
- This is a total function (defined for all expressions)

**Denotational Semantics of Boolean Expressions**
- We inductively define a function
  \[ B[\cdot] : \text{Bexp} \rightarrow (\Sigma \rightarrow \{\text{true}, \text{false}\}) \]
  - \[ B[\text{true}]\sigma = \text{true} \]
  - \[ B[\text{false}]\sigma = \text{false} \]
  - \[ B[b_1 \land b_2]\sigma = B[b_1]\sigma \land B[b_2]\sigma \]
  - \[ B[e_1 = e_2]\sigma = \text{if } A[e_1]\sigma = A[e_2]\sigma \text{ then true else false} \]

**Seems Easy So Far**
- \[ \llbracket \text{SEMANTICS} \rrbracket \]
- [\text{carrot}] = carrot
- [\text{bowling pin}] = bowling pin

**Denotational Semantics for Commands**
- Running a command \( c \) starting from a state \( \sigma \) yields another state \( \sigma' \)
- So, we try to define \( C[\cdot] \) as a function that maps \( \sigma \) to \( \sigma' \)
  \[ C[\cdot] : \text{Com} \rightarrow (\Sigma \rightarrow \Sigma) \]
- Will this work? **No, we have non-termination!**

**\( \bot \) = Non-Termination**
- We introduce the special element \( \bot \) ("bottom") to denote a special resulting state that stands for **non-termination**
- Notation: for any set \( X \), we write \( X_\bot \) to denote \( X \cup \{\bot\} \)
- Convention:
  whenever \( f : X \rightarrow X_\bot \), we extend \( f \) to \( X_\bot \rightarrow X_\bot \) so that \( f(\bot) = \bot \)
- This is called **strictness**
**Denotational Semantics of Commands**

- We try:
  \[ C[\cdot] : \text{Com} \to (\Sigma \to \Sigma^\perp) \]

\[
\begin{align*}
C[\text{skip}] \sigma & = \emptyset \\
C[x := e] \sigma & = \sigma \left[ x := A e \sigma \right] \\
C[c_1; c_2] \sigma & = C[c_2; C[c_1] \sigma] \\
C[\text{if } b \text{ then } c_1 \text{ else } c_2] \sigma & = \\
& \begin{cases} 
C[c_1] \sigma & \text{if } B[b] \sigma \text{ holds} \\
C[c_2] \sigma & \text{otherwise}
\end{cases}
\end{align*}
\]

**Examples**

- \[ C[x:=2; x:=1] \sigma = \sigma[x := 1] \]
- \[ C[\text{if true then } x:=2; x:=1 \text{ else } \ldots] \sigma = \sigma[x := 1] \]
- The semantics does not care about intermediate states (cf. "big-step")
- We haven't used \( \perp \) yet

**DS of While: Unwinding**

- Notation: \( W = C[\text{while } b \text{ do } c] \)
- Idea: rely on the equivalence (justification?)
  \[ \text{while } b \text{ do } c = \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else } \text{skip} \]
- Try
  \[ W(\sigma) = \text{if } B[b] \sigma \text{ then } W(C[c] \sigma) \text{ else } \sigma \]
- This is called the **unwinding equation**
- It is not a good denotation of \( W \) because:
  - It defines \( W \) in terms of itself
  - It is not evident that such a \( W \) exists
  - It does not describe \( W \) uniquely
  - It is not compositional

**Unwinding Equation Does Not Specify \( W \) Uniquely**

- Take \( C[\text{while true do } c] \)
  \[ W(\sigma) = \text{if } B[b] \sigma \text{ then } W(C[c] \sigma) \text{ else } \sigma \]
- Take \( C[\text{while } x \neq 0 \text{ do } x := x - 2] \)
  \[ W(\sigma) = \sigma[x := 0] \text{ if } \sigma(x) \text{ even and } \sigma(x) \geq 0 \]
  \[ \sigma' \text{ otherwise} \]
- for any \( \sigma' \)!
**Denotational Game Plan**

- Since while is recursive
  - always have something like: $W(\sigma) = F(W(\sigma))$
- Admits many possible values for $W(\sigma)$
- We will order them
  - With respect to non-termination = "least"
- And then find the least fixed point

$LFP \ W(\sigma) = F(W(\sigma)) \Rightarrow$ meaning of "while"

---

**Preview: while $k$-steps Semantics**

- Define $W_k: \Sigma \rightarrow \Sigma_\bot$ (for $k \in \mathbb{N}$) such that

  $$
  W_k(\sigma) = \begin{cases} 
  \sigma' & \text{if "while } b \text{ do } c \text{" in state } \sigma \\
  \bot & \text{otherwise}
  \end{cases}
  $$

- We can define the $W_k$ functions as follows:

  $$
  W_0(\sigma) = \bot \\
  W_k(\sigma) = \begin{cases} 
  W_{k-1}(C[c]_{\sigma}) & \text{if } B[b]_{\sigma} \text{ for } k \geq 1 \\
  \sigma & \text{otherwise}
  \end{cases}
  $$

---

**while Semantics**

- How do we get $W$ from $W_k$?

  $$
  W(\sigma) = \begin{cases} 
  \sigma' & \text{if } \exists k. W_k(\sigma) = \sigma' \neq \bot \\
  \bot & \text{otherwise}
  \end{cases}
  $$

- This is a compositional definition of $W$
  - Depends only on $C[c]$ and $B[b]$

---

**It Works, But …**

- Do you feel cheated? Why?

  "Keep executing, operational flavor, worked around the problem."

---

**It Works, But …**

- This solution is not quite satisfactory
  - It has an operational flavor (= "run the loop")
  - It does not generalize easily to more complicated semantics (e.g., higher-order functions)
- However, precisely due to the operational flavor this solution is easy to prove sound w.r.t operational semantics

---

... to be continued
For Next Time

• Homework 2 due Mon Feb 9
• Finish Winskel, Chapter 5
• Get to know Winskel, Chapter 8