Introduction to Denotational Semantics (1/2)
Gone in Sixty Seconds

- **Denotation semantics** is a formal way of assigning meanings to programs. In it, the meaning of a program is a mathematical object.

- Denotation semantics is *compositional*: the meaning of an expression depends on the meanings of subexpressions.

- Denotational semantics uses $\bot$ ("bottom") to mean non-termination.

- DS uses fixed points and domains to handle while.
Induction on Derivations

Summary

- If you must prove $\forall x \in A. \ P(x) \implies Q(x)$
  - $A$ is some structure (e.g., AST), $P(x)$ is some property
  - 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 $x$
    - $D :: P(x)$ leads to induction on the structure of $D$
  - Generally, the induction on the structure of the derivation is more powerful and a safer bet

- Sometimes there are many choices for induction
  - choosing the right one is a trial-and-error process
  - a bit of practice can help a lot
Summary of Operational Semantics

• Precise specification of dynamic semantics
  - order of evaluation (or that it doesn’t matter)
  - error conditions (sometimes implicitly, by rule applicability; “no applicable rule” = “get stuck”)

• Simple and abstract (vs. implementations)
  - no low-level details such as stack and memory management, data layout, etc.

• Often not compositional (see while)

• Basis for many proofs about a language
  - Especially when combined with type systems!

• Basis for much reasoning about programs

• Point of reference for other semantics
Dueling Semantics

- **Operational semantics** is
  - simple
  - of many flavors (natural, small-step, more or less abstract)
  - not compositional
  - commonly used in the real (modern research) world

- **Denotational semantics** is
  - mathematical (the meaning of a syntactic expression is a mathematical object)
  - compositional

- Denotational semantics is also called: fixed-point semantics, mathematical semantics, Scott-Strachey semantics
Typical Student Reaction To Denotation Semantics
Denotational Semantics
Learning Goals

• DS is \textbf{compositional} (!)
• When should I use DS?
• In DS, meaning is a “\textit{math object}”
• DS uses \(\bot\) (“bottom”) to mean non-termination
• DS uses \textbf{fixed points} and \textbf{domains} to handle while
  - This is the tricky bit
You’re On Jeopardy!

Alex Trebek: “The answer is: this property of denotational semantics ...”
DS In The Real World

• ADA was formally specified with it
• Handy when you want to study non-trivial models of computation
  - e.g., “actor event diagram scenarios”, process calculi
• Nice when you want to compare a program in Language 1 to a program in Language 2
Deno-Challenge

• You may skip homework assignment 3 or 4 if you can find two (2) post-2000 papers in first- or second-tier PL conferences that use denotational semantics and you write me a two paragraph summary of each paper.
Foreshadowing

- **Denotational semantics** assigns meanings to programs
- The meaning will be a mathematical object
  - A number $a \in \mathbb{Z}$
  - A boolean $b \in \{\text{true}, \text{false}\}$
  - A function $c : \Sigma \rightarrow (\Sigma \cup \{\text{non-terminating}\})$
- 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

• ‘Cause, why not?

  \[ \] = “means” or “denotes”

• Example:

  \[ \text{foo} \] = “denotation of foo”
  \[ 3 < 5 \] = true
  \[ 3 + 5 \] = 8

• Sometimes we write A[\cdot] for arith, B[\cdot] for boolean, C[\cdot] for command
Rough Idea of Denotational Semantics

• The meaning of an arithmetic expression $e$ in state $\sigma$ is a number $n$

• So, we try to define $A[e]$ as a function that maps the current state to an integer:

$$A[\cdot] : Aexp \rightarrow (\Sigma \rightarrow \mathbb{Z})$$

• The meaning of boolean expressions is defined in a similar way

$$B[\cdot] : Bexp \rightarrow (\Sigma \rightarrow \{true, false\})$$

• All of these denotational function are total
  - Defined for all syntactic elements
  - For other languages it might be convenient to define the semantics only for well-typed elements
Denotational Semantics of Arithmetic Expressions

• We inductively define a function
  \[ A[\cdot] : \text{Aexp} \rightarrow (\Sigma \rightarrow \mathbb{Z}) \]

\[
\begin{align*}
  A[n] \sigma &= \text{the integer denoted by literal } n \\
  A[x] \sigma &= \sigma(x) \\
  A[e_1 + e_2] \sigma &= A[e_1] \sigma + A[e_2] \sigma \\
  A[e_1 - e_2] \sigma &= A[e_1] \sigma - A[e_2] \sigma \\
  A[e_1 \times e_2] \sigma &= A[e_1] \sigma \times A[e_2] \sigma
\end{align*}
\]

• This is a total function (= defined for all expressions)
Denotational Semantics of Boolean Expressions

• We inductively define a function

$$B[] : Bexp \rightarrow (\Sigma \rightarrow \{\text{true, false}\})$$

$$B[\text{true}] \sigma = \text{true}$$

$$B[\text{false}] \sigma = \text{false}$$

$$B[b_1 \land b_2] \sigma = B[b_1] \sigma \land B[b_2] \sigma$$

$$B[e_1 = e_2] \sigma = \text{if } A[e_1] \sigma = A[e_2] \sigma \text{ then true else false}$$
Seems Easy So Far

Semantics
of a structure

\[
\begin{align*}
\left[ \text{\rotatebox{90}{\text{\Large \text{\textit{}}} \text{\rotatebox{-90}{\text{\Large \textit{}}}}}} \right] &= \text{carrot} \\
\left[ \text{\rotatebox{180}{\text{\Large \textit{}}} \text{\rotatebox{0}{\text{\Large \textit{}}} \text{\rotatebox{180}{\text{\Large \textit{}}} \text{\rotatebox{0}{\text{\Large \textit{}}}}} \right] &= \text{bowling pin}
\end{align*}
\]

By Tom 7
Denotational Semantics for Commands

• Running a command $c$ starting from a state $\sigma$ yields another state $\sigma'$

• So, we try to define $\mathbb{C}[\cdot]c$ as a function that maps $\sigma$ to $\sigma'$

$$\mathbb{C}[\cdot] : \text{Comm} \rightarrow (\Sigma \rightarrow \Sigma)$$

• Will this work? Bueller?
\( \bot = \text{Non-Termination} \)

- We introduce the special element \( \bot \) ("bottom") to denote a special resulting state that stands for **non-termination**.
- For any set \( X \), we write \( X\bot \) to denote \( X \cup \{ \bot \} \).

Convention:
whenever \( f \in X \rightarrow X\bot \) we extend \( f \) to \( X\bot \rightarrow X\bot \) so that \( f(\bot) = \bot \).
- This is called **strictness**.
Denotational Semantics of Commands

• We try:

\[ C[\cdot] : \text{Comm} \rightarrow (\Sigma \rightarrow \Sigma_\bot) \]

\[
\begin{align*}
C[\text{skip}] \sigma &= \sigma \\
C[x := e] \sigma &= \sigma[x := A[e] \sigma] \\
C[c_1; c_2] \sigma &= C[c_2] (C[c_1] \sigma) \\
C[\text{if } b \text{ then } c_1 \text{ else } c_2] \sigma &= \\
&\quad \text{if } B[b]\sigma \text{ then } C[c_1]\sigma \text{ else } C[c_2]\sigma \\
C[\text{while } b \text{ do } c] \sigma &= ?
\end{align*}
\]
Examples

- \( C[x:=2; x:=1] \sigma = \sigma[x := 1] \)
- \( C[\text{if true then } x:=2; x:=1 \text{ else } ...] \sigma = \sigma[x := 1] \)
- The semantics does not care about intermediate states (cf. "big-step")
- We haven’t used \( \bot \) yet
Q: Theatre (012 / 842)

• Name the author or the 1953 play about McCarthyism that features John Proctor's famous cry of "More weight!".
Q: General (450 / 842)

- Identify the children's dance here parodied in faux-Shakespearean English:
  - *O proud left foot, that ventures quick within*
  - *Then soon upon a backward journey lithe.*
  - *Anon, once more the gesture, then begin:*
  - *Command sinistral pedestal to writhe.*
Q: Games (557 / 842)

• Name the company that manufactures Barbie (a $1.9 billion dollar a year industry in 2005 with two dolls being bought every second).
In 1995 the Swedish eurodance group Rednex released a version of this late 1800's American bluegrass tune about an attractive man of unknown provenance.
Denotational Semantics of WHILE

- Notation: \( W = C[\text{while } b \text{ do } c] \)
- Idea: rely on the equivalence (see end of notes)
  \[ \text{while } b \text{ do } c \approx \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip} \]
- 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
More on WHILE

• The unwinding equation does not specify W uniquely

• Take \( C[\text{while true do skip}] \)

• The unwinding equation reduces to \( W(\sigma) = W(\sigma) \), which is satisfied by every function!

• Take \( C[\text{while } x \neq 0 \text{ do } x := x - 2] \)

• The following solution satisfies equation (for any \( \sigma' \))

\[
W(\sigma) = \begin{cases} 
\sigma[x := 0] & \text{if } \sigma(x) = 2k \land \sigma(x) \geq 0 \\
\sigma' & \text{otherwise}
\end{cases}
\]
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 \( W(\sigma) = F(W(\sigma)) \) == meaning of “while”
WHILE $k$-steps Semantics

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

\[
W_k(\sigma) = \begin{cases} 
\sigma' & \text{if “while } b \text{ do } c \text{” in state } \sigma \text{ terminates in fewer than } k \\ 
\bot & \text{otherwise} 
\end{cases}
\]

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

- Try the examples again:
  - For $C[\text{while true do skip}]$
    \[ W_k(\sigma) = \bot \text{ for all } k, \text{ thus } W(\sigma) = \bot \]
  - For $C[\text{while } x \neq 0 \text{ do } x := x - 2]$
    \[ W(\sigma) = \begin{cases} 
  \sigma[x:=0] & \text{if } \sigma(x) = 2n \land \sigma(x) \geq 0 \\
  \bot & \text{otherwise}
\end{cases} \]
More on WHILE

- The solution is not quite satisfactory because
  - 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
That Wasn’t Good Enough!?
Simple Domain Theory

- Consider programs in an eager, deterministic language with one variable called “x”
  - All these restrictions are just to simplify the examples
- A state $\sigma$ is just the value of $x$
  - Thus we can use $\mathbb{Z}$ instead of $\Sigma$
- The semantics of a command give the value of final $x$ as a function of input $x$

$$C[ c ] : \mathbb{Z} \rightarrow \mathbb{Z}_\perp$$
Examples - Revisited

• Take $C[\text{while true do skip}]$
  - Unwinding equation reduces to $W(x) = W(x)$
  - Any function satisfies the unwinding equation
  - Desired solution is $W(x) = \perp$

• Take $C[\text{while } x \neq 0 \text{ do } x := x - 2]$
  - Unwinding equation:
    \[ W(x) = \begin{cases} W(x - 2) & \text{if } x \neq 0 \\ x & \text{else} \end{cases} \]
  - Solutions (for all values $n, m \in \mathbb{Z}_\perp$):
    \[ W(x) = \begin{cases} 0 & \text{if } x \geq 0 \text{ and } x \text{ even} \\ n & \text{else} \end{cases} \]
  - Desired solution: $W(x) = \begin{cases} 0 & \text{if } x \geq 0 \land x \text{ even} \\ \perp & \text{else} \end{cases}$
An Ordering of Solutions

- The desired solution is the one in which all the arbitrariness is replaced with non-termination.
  - The arbitrary values in a solution are not uniquely determined by the semantics of the code.
- We introduce an ordering of semantic functions.
- Let $f, g \in \mathbb{Z} \rightarrow \mathbb{Z}_\perp$.
- Define $f \sqsubseteq g$ as
  \[ \forall x \in \mathbb{Z}. \ f(x) = \perp \text{ or } f(x) = g(x) \]
  - A “smaller” function terminates at most as often, and when it terminates it produces the same result.
Alternative Views of Function Ordering

- A semantic function $f \in \mathbb{Z} \rightarrow \mathbb{Z}_\perp$ can be written as $S_f \subseteq \mathbb{Z} \times \mathbb{Z}$ as follows:

  $$S_f = \{ (x, y) \mid x \in \mathbb{Z}, f(x) = y \neq \bot \}$$

  - set of “terminating” values for the function

- If $f \sqsubseteq g$ then
  - $S_f \subseteq S_g$ (and vice-versa)
  - We say that $g$ refines $f$
  - We say that $f$ approximates $g$
  - We say that $g$ provides more information than $f$
The “Best” Solution

• Consider again \( \text{C[while } x \neq 0 \text{ do } x := x - 2] \)
  - Unwinding equation:
    \[ W(x) = \text{if } x \neq 0 \text{ then } W(x - 2) \text{ else } x \]

• Not all solutions are comparable:
  \[ W(x) = \text{if } x \geq 0 \text{ then if } x \text{ even then } 0 \text{ else } 1 \text{ else } 2 \]
  \[ W(x) = \text{if } x \geq 0 \text{ then if } x \text{ even then } 0 \text{ else } \bot \text{ else } 3 \]
  \[ W(x) = \text{if } x \geq 0 \text{ then if } x \text{ even then } 0 \text{ else } \bot \text{ else } \bot \]
  (last one is least and best)

• Is there always a least solution?
• How do we find it?
• *If only we had a general framework* for answering these questions ...
Fixed-Point Equations

• Consider the general unwinding equation for \texttt{while}
  \texttt{while b do c ≡ if b then c; while b do c else skip}

• We define a context \( C \) (command with a hole)
  \[ C = \texttt{if b then c; \bullet else skip} \]
  \texttt{while b do c ≡ C[while b do c]}
  \begin{itemize}
    \item The grammar for \( C \) does not contain “while b do c”
  \end{itemize}

• We can find such a (recursive) context for any looping construct
  \begin{itemize}
    \item Consider: \texttt{fact n = if n = 0 then 1 else n * fact (n - 1)}
    \item \( C(n) = \texttt{if n = 0 then 1 else n * \bullet (n - 1)} \)
    \item \texttt{fact = C [ fact ]}
  \end{itemize}
Fixed-Point Equations

• The meaning of a context is a semantic functional
  \( F : (\mathbb{Z} \rightarrow \mathbb{Z}_\perp) \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}_\perp) \) such that
  \[ F [C[w]] = F [w] \]
• For “while”: \( C = \text{if } b \text{ then } c; \bullet \text{ else skip} \)
  \[ F w x = \text{if } [b] x \text{ then } w ([c] x) \text{ else } x \]
  - \( F \) depends only on \([c]\) and \([b]\)
• We can rewrite the unwinding equation for while
  - \( W(x) = \text{if } [b] x \text{ then } W([c] x) \text{ else } x \)
  - or, \( W x = F W x \) for all \( x \),
  - or, \( W = F W \) (by function equality)
Fixed-Point Equations

• The meaning of “while” is a solution for $W = F W$
• Such a $W$ is called a fixed point of $F$
• We want the least fixed point
  - We need a general way to find least fixed points
• Whether such a least fixed point exists depends on the properties of function $F$
  - Counterexample: $F w x = \text{if } w x = \bot \text{ then } 0 \text{ else } \bot$
  - Assume $W$ is a fixed point
  - $F W x = W x = \text{if } W x = \bot \text{ then } 0 \text{ else } \bot$
  - Pick an $x$, then $\text{if } W x = \bot \text{ then } W x = 0 \text{ else } W x = \bot$
  - Contradiction. This $F$ has no fixed point!
Can We Solve This?

• Good news: the functions $F$ that correspond to contexts in our language have least fixed points!
• The only way $F \, w \, x$ uses $w$ is by invoking it.
• If any such invocation diverges, then $F \, w \, x$ diverges!
• It turns out: $F$ is monotonic, continuous
  - Not shown here!
New Notation: \( \lambda \)

- \( \lambda x. e \)
  - an anonymous function with body \( e \) and argument \( x \)

- Example: \( \text{double}(x) = x+x \)
  \[
  \text{double} = \lambda x. x+x
  \]

- Example: \( \text{allFalse}(x) = \text{false} \)
  \[
  \text{allFalse} = \lambda x. \text{false}
  \]

- Example: \( \text{multiply}(x,y) = x*y \)
  \[
  \text{multiply} = \lambda x. \lambda y. x*y
  \]
The Fixed-Point Theorem

- If $F$ is a semantic function corresponding to a context in our language
  - $F$ is monotonic and continuous (we assert)
  - For any fixed-point $G$ of $F$ and $k \in \mathbb{N}$
    \[ F^k(\lambda x. \perp) \subseteq G \]
  - The least of all fixed points is
    \[ \bigsqcup_k F^k(\lambda x. \perp) \]

- Proof (not detailed in the lecture):
  1. By mathematical induction on $k$.
     Base: $F^0(\lambda x. \perp) = \lambda x. \perp \subseteq G$
     Inductive: $F^{k+1}(\lambda x. \perp) = F(F^k(\lambda x. \perp)) \subseteq F(G) = G$
  - Suffices to show that $\bigsqcup_k F^k(\lambda x. \perp)$ is a fixed-point
    \[ F(\bigsqcup_k F^k(\lambda x. \perp)) = \bigsqcup_k F^{k+1}(\lambda x. \perp) = \bigsqcup_k F^k(\lambda x. \perp) \]
WHILE Semantics

- We can use the fixed-point theorem to write the denotational semantics of while:

  \[ \text{while } b \text{ do } c = \bigcup_k F^k (\lambda x. \perp) \]

  where \( F f x = \text{if } \llbracket b \rrbracket x \text{ then } f (\llbracket c \rrbracket x) \text{ else } x \)

- Example: \( \llbracket \text{while true do skip} \rrbracket = \lambda x. \perp \)

- Example: \( \llbracket \text{while } x \neq 0 \text{ then } x := x - 1 \rrbracket \)
  - \( F (\lambda x. \perp) x = \text{if } x = 0 \text{ then } x \text{ else } \perp \)
  - \( F^2 (\lambda x. \perp) x = \text{if } x = 0 \text{ then } x \text{ else if } x-1 = 0 \text{ then } x-1 \text{ else } \perp \)
  = \( \text{if } 1 \geq x \geq 0 \text{ then } 0 \text{ else } \perp \)
  - \( F^3 (\lambda x. \perp) x = \text{if } 2 \geq x \geq 0 \text{ then } 0 \text{ else } \perp \)
  - \( \text{LFP}_F = \text{if } x \geq 0 \text{ then } 0 \text{ else } \perp \)

- Not easy to find the closed form for general LFPs!
Discussion

• We can write the denotational semantics but we cannot always compute it.
  - Otherwise, we could decide the halting problem
  - $H$ is halting for input 0 iff $\llbracket H \rrbracket 0 \neq \perp$

• We have derived this for programs with one variable
  - Generalize to multiple variables, even to variables ranging over richer data types, even higher-order functions: domain theory
Can You Remember?

You just survived the hardest lectures in 615.
It’s all downhill from here.
Recall: Learning Goals

• DS is **compositional**
• When should I use DS?
• In DS, meaning is a “math object”
• DS uses \(\bot\) (“bottom”) to mean non-termination
• DS uses **fixed points** and **domains** to handle while
  - This is the tricky bit
Homework

- Homework 2 Due Thursday
- Homework 3
  - Not as long as it looks - separated out every exercise sub-part for clarity.
  - Your denotational answers must be compositional (e.g., $W_k(\sigma)$ or LFP)
- Read Winskel Chapter 6
- Read Hoare article
- Read Floyd article
Equivalence

- Two expressions (commands) are equivalent if they yield the same result from all states

\[ e_1 \approx e_2 \iff \forall \sigma \in \Sigma. \forall n \in \mathbb{N}. \]
\[ <e_1, \sigma> \Downarrow n \iff <e_2, \sigma> \Downarrow n \]

and for commands

\[ c_1 \approx c_2 \iff \forall \sigma, \sigma' \in \Sigma. \]
\[ <c_1, \sigma> \Downarrow \sigma' \iff <c_2, \sigma> \Downarrow \sigma' \]
Notes on Equivalence

- Equivalence is like logical validity
  - It must hold in all states (= all valuations)
  - $2 \approx 1 + 1$ is like “$2 = 1 + 1$ is valid”
  - $2 \approx 1 + x$ might or might not hold.
    - So, $2$ is not equivalent to $1 + x$
- Equivalence (for IMP) is **undecidable**
  - If it were decidable we could solve the halting problem for IMP. *How?*
- Equivalence justifies code transformations
  - compiler optimizations
  - code instrumentation
  - abstract modeling
- **Semantics** is the basis for proving equivalence
Equivalence Examples

- skip; c \approx c
- while b do c \approx if b then c; while b do c else skip
- If e_1 \approx e_2 then x := e_1 \approx x := e_2
- while true do skip \approx while true do x := x + 1
- If c is
  while x \neq y do
    if x \geq y then x := x - y else y := y - x
  then
  (x := 221; y := 527; c) \approx (x := 17; y := 17)
Potential Equivalence

- \((x := e_1; x := e_2) \approx x := e_2\)

- Is this a valid equivalence?
Not An Equivalence

\( (x := e_1; x := e_2) \sim x := e_2 \)

lie. Chigau yo. Dame desu!

Not a valid equivalence for all \( e_1, e_2 \).

Consider:

\(- (x := x+1; x := x+2) \sim x := x+2 \)

But for \( n_1, n_2 \) it’s fine:

\(- (x := n_1; x := n_2) \approx x := n_2 \)
Proving An Equivalence

• Prove that “skip; c ≈ c” for all c
• Assume that D :: <skip; c, σ> ↓ σ'
• By inversion (twice) we have that

\[ D :: \frac{\langle \text{skip}, \sigma \rangle \downarrow \sigma \quad D_1 :: \langle c, \sigma \rangle \downarrow \sigma'}{\langle \text{skip}; c, \sigma \rangle \downarrow \sigma'} \]

• Thus, we have D_1 :: <c,σ> ↓ σ'
• The other direction is similar
Proving An Inequivalence

- Prove that $x := y \not\sim x := z$ when $y \neq z$
- It suffices to exhibit a $\sigma$ in which the two commands yield different results

- Let $\sigma(y) = 0$ and $\sigma(z) = 1$
- Then
  
  $$<x := y, \sigma> \Downarrow \sigma[x := 0]$$
  $$<x := z, \sigma> \Downarrow \sigma[x := 1]$$