Denotational Semantics; Runtime Errors

CS 536: Science of Programming, Fall 2021

A. Why

• A program or statement can be viewed as denoting a state transformation.
• Infinite loops and runtime errors cause failure of normal program execution.

B. Outcomes

At the end of today, you should know how to

• Use denotational semantics to describe overall execution of programs in our language
• Determine that evaluation of an expression or program fails due to a runtime error.

C. Denotational Semantics Definition and Rules

• In addition to the small step-by-step operational semantics for our programs, we’ll also introduce a version of semantics that concentrates only on the beginning and end of the evaluation process (hence he name “large-step” semantics).
• Definition: The denotational semantics of \( S \) in \( \sigma \) is \( \tau \) if in state \( \sigma \), program \( S \) terminates in \( \tau \). (I.e., \( \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \).) Symbolically, we write \( M(S, \sigma) = \{\tau\} \).
  • The reason we have a singleton set containing \( \tau \) instead of just \( \tau \) is that later, we’ll look at non-deterministic computations, which can have more than one possible final state.

• Notation: If you slip up and write \( M(S, \sigma) = \tau \) instead of \( \{\tau\} \), it’s not a big deal.
• Example 1: Let \( \sigma \) be a state and let \( S = x := 1; y := 2 \). Since \( \langle x := 1; y := 2, \sigma \rangle \rightarrow \langle y := 2, \sigma[x \mapsto 1] \rangle \rightarrow \langle E, \sigma[x \mapsto 1][y \mapsto 2] \rangle \), we know \( M(S, \sigma) = \{\sigma[x \mapsto 1][y \mapsto 2]\} \).
• Notation: In the literature, some people write hollow square brackets around arguments that are syntactic to emphasize that they are indeed syntactic. Other notations for \( M(S, \sigma) \) include \( M[S](\sigma) \) and \( M[S] \) \( \sigma \) and \( M(S) \) \( \sigma \). In the last two cases, \( M[S] \) and \( M(S) \) are viewed as functions that transform memory states, so \( M[S](\sigma) = \tau \) means function \( M[S] \) maps \( \sigma \) to \( \tau \). Our notation \( \sigma[e] \) would be written \( \sigma[e] \).

Denotational Semantics Rules

• Since \( M(S, \sigma) = \tau \) means \( \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \), we can give specific rules for \( M(S, \sigma) \) depending on the kind of \( S \).
• Skip and Assignment: These statements complete in only one step, so the operational semantics rules give the denotational semantics immediately.
  • \( M(skip, \sigma) = \{\sigma\} \)
• $M(\nu := e, \sigma) = \{\sigma[\nu \mapsto \sigma(e)]\}$
• $M(b[e_1] := e, \sigma) = \{\sigma[b[\alpha \mapsto \beta]]\}$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e)$.

**Composition:** $M(S_i; S_2, \sigma) = M(S_2, \tau)$ where $\{\tau\} = M(S_1, \sigma)$. To justify this, say we have $\langle S_i; S_2, \sigma \rangle \rightarrow^* \langle S_2, \tau \rangle \rightarrow^* \langle E, \tau \rangle$. Since $M(S_1, \sigma) = \{\tau\}$, we run $S_2$ starting in state $\tau$, so $M(S_i; S_2, \sigma) = M(S_2, \tau) = M(S_2, M(S_1, \sigma))$.

**Notation:** We’ll bend the notation a bit and write $M(S_2, M(S_1, \sigma))$ as short for $M(S_2, \tau)$ where $\{\tau\} = M(S_1, \sigma)$. (Note: In $M(S_2, M(S_1, \sigma))$, the subscripts appear as 2 then 1, not 1 then 2.)

**Conditional:** One way to definite the meaning of $W = \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}$ is either the meaning of the true branch or the meaning of the false branch.

• If $\sigma(B) = T$, then $M(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) = M(S_1, \sigma)$
• If $\sigma(B) = F$, then $M(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) = M(S_2, \sigma)$

**Example 2:** Let $S = \text{if } y \text{ then } x := x+1 \text{ else } z := x+2 \text{ fi}$, then

• If $\sigma(y) = T$, then $M(S, \sigma) = \{\sigma[x \mapsto \sigma(x)+1]\}$
• If $\sigma(y) = F$, then $M(S, \sigma) = \{\sigma[z \mapsto \sigma(x)+2]\}$

**Iterative:** One way to definite the meaning of $W = \text{while } B \text{ do } S \text{ od}$ is recursively:

• If $\sigma(B) = F$ then $M(W, \sigma) = \{\sigma\}$
• If $\sigma(B) = T$ then $M(W, \sigma) = M(S; W, \sigma) = M(W, M(S, \sigma))$.

Unfortunately, this definition is not well-formed if $W$ leads to an infinite loop.

**Another way to characterize $M(W, \sigma)$ involves looking at the series of states in which we evaluate the test.**

• Let $\sigma_0 = \sigma$, and for for $k = 0, 1, \ldots$, let $\{\sigma_{k+1}\} = M(S, \sigma_k)$. Then $\sigma_0, \sigma_1, \sigma_2, \ldots$ is the sequence of states seen at successive while loop tests: $\sigma_k$ is the state in effect the $k$’th time we evaluate the loop test.

• Then $M(W, \sigma)$ is the (set containing the) first state in this sequence that satisfies $\lnot B$, assuming there is such a state. (If there isn’t, we have an infinite loop.)

**Example 3:** Let $W = \text{while } x < n \text{ do } S \text{ od}$, where the loop body $S = x := x+1; y := y+y$. The general case for the behavior of $S$ is (for any $\tau$), $M(S, \tau[x \mapsto \alpha][y \mapsto \beta]) = \{\tau[x \mapsto \alpha+1][y \mapsto 2 \beta]\}$. Say we start execution of $W$ in state $\sigma = \{x = 0, n = 3, y = 1\}$. Our sequence of states is

• $\sigma_0 = \sigma = \{x = 0, n = 3, y = 1\}$
• $M(S, \sigma_0) = \{\sigma_1\}$ where $\sigma_1 = \{x = 1, n = 3, y = 2\}$
• $M(S, \sigma_1) = \{\sigma_2\}$ where $\sigma_2 = \{x = 2, n = 3, y = 4\}$, and
• $M(S, \sigma_2) = \{\sigma_3\}$ where $\sigma_3 = \{x = 3, n = 3, y = 8\}$.

Of this sequence, $\sigma_3$ is the first state that satisfies $x \geq n$, so $M(W, \sigma) = \{\sigma_3\} = \{\{x = 3, n = 3, y = 8\}\}$.

**D. Convergence and Divergence of Loops**

• Not all loops terminate. Evaluation of an infinite loop yields an unending path of $\rightarrow$ steps: Either an infinite sequence of different configurations or a finite-length cycle of configurations. More
generally in computer science we can also also have infinite recursion, which we won't study in detail but is treated similarly to infinite iteration.

- **Definition:** Execution of $S$ starting in $\sigma$ **diverges** if it doesn't converge; i.e., $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$ for no $\tau$.

- **Notation:** $M(S, \sigma) = \{\bot_d\}$ (**bottom sub-d**) means $S$ diverges in $\sigma$. Note that although we're writing it in a place where you'd expect a memory state, $\bot_d$ is not an actual memory state; we'll call it a **pseudo-state** as opposed to an actual or real memory state like $\sigma$ and $\tau$.

- **Note:** Divergence is one way in which a program doesn't successfully terminate. We'll introduce other flavors of $\bot$ as we look at other ways to not get successful termination.

- **Notation:** $\langle S, \sigma \rangle \rightarrow^* \langle E, \bot_d \rangle$ means that $S$ starting in $\sigma$ diverges. Again, we're not using $\bot_d$ as an actual memory state here, but since $M(S, \sigma) = \{\tau\}$ means $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$, if we're going to write $M(S, \sigma) = \{\bot_d\}$ to say that $S$ diverges, it's consistent to write $\langle S, \sigma \rangle \rightarrow^* \langle E, \bot_d \rangle$.

- To determine when $M(W, \sigma) = \{\bot_d\}$, recall that in the previous section we looked at the series of states $\sigma_0, \sigma_n, \sigma_n, ...$ in which we evaluate the loop test. For this sequence, $\sigma_0 = \sigma$, and $\sigma_{k+1} = M(S, \sigma_k)$ for $k \geq 0$. For terminating loops, $M(W, \sigma)$ is the first state in the sequence that satisfies $\neg B$. We can now write $M(W, \sigma) = \{\bot_d\}$ to indicate that no state in the sequence satisfies $\neg B$.

- **Example 4:** Let $W = \textbf{while } T \textbf{ do skip od}$ and $\sigma$ be any state. Then $\langle W, \sigma \rangle \rightarrow \langle \textbf{skip ; } W, \sigma \rangle$ but $\langle \textbf{skip ; } W, \sigma \rangle \rightarrow \langle W, \sigma \rangle$. (As a directed graph, this is a two-node cycle, $\langle W, \sigma \rangle \Leftrightarrow \langle \textbf{skip ; } W, \sigma \rangle$.) Hence $M(W, \sigma) = \{\bot_d\}$.

- **Example 5:** Let $W = \textbf{while } x \neq n \textbf{ do } x := x-1 \textbf{ od}$ and let $\sigma = \{x = -1, n = 0\}$.
  - Let $\sigma_0 = \sigma = \{x = -1, n = 0\}$
  - Let $\sigma_1 = M(x := x-1, \sigma_0) = \{x = -2, n = 0\}$
  - Let $\sigma_2 = M(x := x-1, \sigma_1) = \{x = -3, n = 0\}$
  - In general, let $\sigma_k = M(x := x-1, \sigma_{k-1}) = \{x = -k, n = 0\}$
  - Since every $\sigma_k \models x \neq n$, we have $M(W, \sigma) = \{\bot_d\}$.

### E. Expressions With Runtime Errors

- Using $\bot_d$ lets us talk about a program not successfully terminating because it simply doesn't terminate at all.

- Runtime errors cause a program to terminate, but unsuccessfully. E.g, in $\sigma$, the assignment $z := x/y$ fails if $\sigma(y) = 0$ because evaluation of $\sigma(x/y)$ fails. There are two notions of failure here: The expression fails, and this causes the statement to fail.

- **Definition:** $\sigma(e) = \bot_e$ means evaluation of $e$ in state $\sigma$ causes a runtime error.

  - Here, $\bot_e$ is used as a pseudo-value of an expression, to indicate an error. It's not a value; we're writing it in place of an actual value.

  - If $e$ can fail at runtime, then instead of $\sigma(e) \in V$ for some set of values $V$, we now have $\sigma(e) \in V \cup \{\bot_e\}$. Of course, some expressions never fail: $\sigma(2+2) \in \mathbb{Z} \cup \{\bot_e\}$ but more specifically, $\sigma(2+2) \in \mathbb{Z}$.
• **Primary errors**: The primitive values and operations being supported determines what basic runtime errors can occur. For us, let’s include:
  - *Array index out of bounds*: \( \sigma[b[e]] = \bot_e \) if \( \sigma(e) < 0 \) or \( \geq \sigma(\text{size}(b)) \); similar for multiple dimensions.
  - *Division by zero*: \( \sigma(e_1/e_2) = \sigma(e_1, \% e_2) = \bot_e \) if \( \sigma(e_2) = 0 \).
  - *Square root of negative number*: \( \sigma(\sqrt{e}) = \bot_e \) if \( \sigma(e) < 0 \).

• **Example 6**: \( b[-1], n/0, \) and \( \sqrt{-1} \) fail for all \( \sigma \). \( b[k] \) fails in state \( \{b = (2, 3, 5, 8), k = 4\} \) but not in state \( \{b = (6), k = 0\} \).

• **Hereditary Failure**: If evaluating a subexpression fails, then the overall expression fails.
  - If \( op \) is a unary operator, then \( \sigma(op\ e) = \bot_e \) if \( \sigma(e) = \bot_e \).
  - If \( op \) is a binary operator, then \( \sigma(e_1\ op\ e_2) = \bot_e \) if \( \sigma(e_1) \) or \( \sigma(e_2) = \bot_e \).
  - For a conditional expression, \( \sigma(\text{if } B \ \text{then } e_1 \ \text{else } e_2 \ \text{fi}) = \bot_e \) if one of the following three situations occurs: (1) \( \sigma(B) = \bot_e \) (2) \( \sigma(B) = T \) and \( \sigma(e_1) = \bot_e \) or (3) \( \sigma(B) = F \) and \( \sigma(e_2) = \bot_e \). We don’t worry about a hypothetical failure of the branch we don’t evaluate.

• **Example 7**: \( \sigma(x/y) = \bot_e \) when \( \sigma(y) = 0 \), but \( \sigma(y = 0 \ ? \ 0 : x/y) \) never = \( \bot_e \).

F. **Statements With Runtime Errors**

An expression that causes a runtime error causes the statement it appears in to terminate unsuccessfully. We’ll write \( \langle S, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \) for the operational semantics of such a statement. This use of \( \bot_e \) as a (pseudo)-state is different from its use as a pseudo-value (\( \sigma(e) = \bot_e \)).

• **Definition** (Statements with expressions with runtime errors) If a statement evaluates an expression that causes a runtime error, then the statement terminates unsuccessfully. To the operational semantics, we add:
  - If \( \sigma(e) = \bot_e \), then \( \langle v := e, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \).
  - If \( \sigma(e_1) \) or \( \sigma(e_2) = \bot_e \), then \( \langle b[e_1] := e_2, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \).
  - If \( \sigma(B) = \bot_e \), then \( \langle \text{while } B \text{ do } S \text{ od}, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \).
  - If \( \sigma(B) = \bot_e \), then \( \langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \).
  - If \( \langle S_1, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \) then \( \langle S_1; S_2, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \) where \( T_1 \) either is a statement or \( E \).

The pseudo-states \( \bot_d \) and \( \bot_e \) share some properties, so it’s helpful to have a more general notation for “error”.

• **Notation**: \( \bot \) refers generically to \( \bot_d \) and/or \( \bot_e \). For example, \( \langle S, \sigma \rangle \rightarrow^* \langle E, \bot \rangle \) means either \( \langle S, \sigma \rangle \rightarrow^* \langle E, \bot_e \rangle \) (evaluation of \( S \) causes a runtime error) or \( \langle S, \sigma \rangle \rightarrow^* \langle E, \bot_d \rangle \) (evaluation of \( S \) diverges).

• **Notation**: \( \bot \in M(S, \sigma) \) means \( \langle S, \sigma \rangle \rightarrow^* \langle E, \bot \rangle \). (Here, \( \bot \) can be \( \bot_d \) or \( \bot_e \).)

• **Trying to use** \( \bot \): Since we are writing \( \bot \) in some of the places where an actual memory state would appear, it’s good to be thorough, look at the other places states appear, and extend those notions or notations.
  - When we say “for all states...” or “for some state...,” we don’t include \( \bot \).
• We can’t add a binding to \( \perp \): \( \perp [v \mapsto \alpha] = \perp \).

• We can’t bind a variable to \( \perp \): \( \sigma(v) \neq \perp \) and \( \sigma[\perp] = \perp \).

• We can’t take the value of a variable or expression in \( \perp \): If \( \sigma = \perp \) then \( \sigma(v) = \sigma(e) = \perp \). (More succinctly, \( \perp(v) = \perp(e) = \perp \).)

• Operationally, execution halts as soon we generate \( \perp \) as a “state”: \( \langle S, \perp \rangle \rightarrow^0 \langle E, \perp \rangle \).

• Denotationally, we can’t run a program in \( \perp \): \( M(S, \perp) = \{\perp\} \)

• From the properties above, it follows that we can’t evaluate something after generating \( \perp \).
  - If \( \langle S_1, \sigma \rangle \rightarrow \langle E, \perp \rangle \), then \( \langle S_1; S_2, \sigma \rangle \rightarrow \langle E, \perp \rangle \).
  - If \( M(S_1, \sigma) = \{\perp\} \), then \( M(S_1; S_2, \sigma) = M(S_2, M(S_1, \sigma)) = M(S_2, \perp) = \{\perp\} \).
  - If \( W = \text{while } B \text{ do } S_1 \text{ od} \) and \( \sigma(B) = T \) but \( M(S_1, \sigma) = \{\perp\} \), then \( M(W, \sigma) = \{\perp\} \).
    - (In detail, \( M(W, \sigma) = M(S_1; W, \sigma) = M(W, M(S_1, \sigma)) = M(W, \perp) = \{\perp\} \).)

**Satisfaction and Validity and** \( \perp \): \( \perp \) never satisfies a predicate: \( \perp \not\models p \) for all \( p \), even if \( p = \text{the constant } T \). In general, we now have three possibilities: \( \sigma \models p, \sigma \models \neg p, \text{ or } \sigma = \perp \). So \( \sigma \not\models p \) is now equivalent to \( \sigma \models \neg p \) or \( \sigma = \perp \), not just \( \sigma \models \neg p \). We can also have \( \sigma \not\models p \) and \( \sigma \not\models \neg p \) simultaneously (when \( \sigma = \perp \)).

**Logical negation and** \( \perp \): Since \( \sigma \models \neg p \) is no longer equivalent to \( \sigma \not\models p \), we need a better notion of what \( \neg p \) means. The solution is to treat \( \neg p \) as shorthand for \( p \rightarrow F \) where \( F \) is the predicate “false”.

  - Just a quick note: For the meaning of \( T \) and \( F \), we have \( \sigma \models T \) and \( \sigma \not\models F \) for all \( \sigma \). (We can also derive \( F \) by defining \( F = T \neq T \).) For all \( \sigma \) (i.e., unless \( \sigma = \perp \)), \( \sigma \models F \rightarrow F \), so \( \sigma \not\models \neg F \).

**Generating** \( \perp \) **while testing for satisfaction:** Another problem to worry about is what to do if evaluation of a predicate causes an error? Clearly, we can’t allow things like \( \{y = 0\} \models y/y = 1 \). To handle this, we’ll add \( \perp \) to the semantics of basic operations and tests:
  - For any relation (like less than, etc), we have \( (\alpha \text{ relation } \beta) \text{ yields } \perp \) if \( \alpha \) or \( \beta = \perp \).
  - For any binary operation (like addition, etc), we have \( (\alpha \text{ operation } \beta) \text{ yields } \perp \) if \( \alpha \) or \( \beta = \perp \).
  - Similarly for a unary operation, we have \( (\text{operation } \perp) \) yields \( \perp \).
  - Some of the implications of this are reasonably intuitive: \( (\perp \text{ plus one}) \) yields \( \perp \).
  - But some implications are less intuitive: Semantic operations and tests like \( \perp \neq 2, \perp < \perp, \perp = \perp, \) and \( \perp \neq \perp \) all yield \( \perp \) (not \( T \) or \( F \)).
  - Returning to \( y/y = 1 \), we still have \( \sigma \models y/y = 1 \) iff \( \sigma(y/y) = \sigma(1) \) iff \( (\sigma(y) \text{ divided by } \sigma(y)) = \text{one} \), so
    - If \( \sigma(y) \neq 0 \), then \( \sigma \models y/y = 1 \) iff \( (\text{divided by } \alpha = \text{one}) \) iff \( (\text{one} = \text{one}) \) iff true
    - But if \( \sigma(y) = 0 \), then \( \sigma \models y/y = 1 \) iff \( (0 \text{ divided by } 0 = \text{one}) \) iff \( (\perp = \text{one}) \) iff \( \perp \).
    - So if \( \sigma(y) = 0 \), then \( \sigma \neq y/y = 1 \) and similarly (since \( (\perp \neq \text{one}) \) yields \( \perp \), \( \sigma \neq y/y \neq 1 \).

---

* I’m using “yields” here for “semantically evaluates to”. “Equals” or “=” are okay as long as we remember we mean semantic equality.