Denotational Semantics; Runtime Errors

CS 536: Science of Programming, Spring 2023

2023-01-31: pp. 1,5,6

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

- We've seen the "small" step-by-step operational semantics for our programs. Today we'll look at a "large" step semantics.
- ***Definition***: If in state σ, program S terminates in τ, then τ is the ***denotational semantics*** of S in σ. Symbolically, if $\langle S, \sigma \rangle \to^* \langle E, \tau \rangle$, then we write $M(S, \sigma) = \{\tau\}$.
  - The reason we have a singleton set containing τ instead of just τ is that later, we'll look at non-deterministic computations, which can have more than one possible final state.
- ***Notation***: We'll often write $M(S, \sigma) = \tau$ instead of $\{\tau\}$. The only (obscure) time the difference is important is when $\tau = \varnothing$; in that case we should write $M(S, \sigma) = \{\varnothing\}$.
- ***Example 1***: Let σ be a state and let $S \equiv x := 1; y := 2$. Since $\langle x := 1; y := 2, \sigma \rangle \to \langle y := 2, \sigma[x \mapsto 1] \rangle \to \langle E, \sigma[x \mapsto 1][y \mapsto 2] \rangle$, we have $M(S, \sigma) = \{\sigma[x \mapsto 1][y \mapsto 2]\}$.

  [2023-01-31]

  $$\begin{aligned} & M(x := 1; y := 2, \sigma) & \\ = \; & M(y := 2, \sigma[x \mapsto 1]) & \textit{asgt } x{:=}1 \\ = \; & M(E, \sigma[x \mapsto 1][y \mapsto 2]) & \textit{asgt } y{:=}2 \\ = \; & \{\sigma[x \mapsto 1][y \mapsto 2]\} & \textit{defn } M \end{aligned}$$

- ***Notation***: In the literature, some people write hollow square brackets around arguments that are syntactic to emphasize that they are indeed syntactic. E.g, our $\sigma(e)$ would be written $\sigma[\![e]\!]$.
- ***Notation:*** Another notation defines $M[\![S]\!]$ or $M(S)$ as a state transformation function, which you can apply to a state σ to get τ. One writes $\tau = M[\![S]\!](\sigma)$ or $M(S)(\sigma)$ or $M[\![S]\!]\sigma$ or $M(S)\sigma$. In yet another notation we pass a set of possible start states to $M(S, \ldots)$ and get a set of possible end states; we'd write $M(S, \{\sigma\}) = \{\tau\}$.

Denotational Semantics Rules

- Since $M(S,\sigma)=\tau$ means $\langle S,\sigma\rangle \to^* \langle E,\tau\rangle$, we can give specific rules for $M(S,\sigma)$ depending on the kind of S.
- ***Skip and Assignment Statements:*** These statements complete in only one step, so the operational semantics rules give the denotational semantics immediately.
  - $M(\textbf{skip},\sigma)=\{\sigma\}$.
  - $M(v:=e,\sigma)=\{\sigma[v\mapsto\sigma(e)]\}$.
  - $M(b[e_1]:=e,\sigma)=\{\sigma[b[\alpha]\mapsto\beta]\}$ where $\alpha=\sigma(e_1)$ and $\beta=\sigma(e)$.
- ***Composition Statements***: $M(S_1;\ S_2,\sigma)=M(S_2,\tau)$ where $\{\tau\}=M(S_1,\sigma)$. To justify this, say we have $\langle S_1;\ S_2,\sigma\rangle\to^*\langle S_2,\tau\rangle\to^*\langle E,\tau'\rangle$. Since $M(S_1,\sigma)=\{\tau\}$, we run S_2 starting in state τ, so $M(S_1;\ S_2,\sigma)=M(S_2,\tau)=M(S_2,M(S_1,\sigma))$. Note: In $M(S_2,M(S_1,\sigma))$, the subscripts appear as *2* then *1*, not *1* then *2*.
- ***Notation***: We'll bend the notation a bit and allow $M(S_2,M(S_1,\sigma))$ as short for $M(S_2,\tau)$ when $M(S_1,\sigma)=\{\tau\}$.
- ***Conditional Statements***: The meaning of an ***if-else*** statement is either the meaning of the true branch or the meaning of the false branch.
  - If $\sigma(B)=T$, then $M(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi},\sigma)=M(S_1,\sigma)$
  - If $\sigma(B)=F$, then $M(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi},\sigma)=M(S_2,\sigma)$
- ***Example 2***: Let $S\equiv$ ***if*** y ***then*** $x:=x+1$ ***else*** $z:=x+2$ ***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 Statements***: One way to definite the meaning of $W\equiv$ ***while*** B ***do*** S ***od*** is recursively. The definition is appealing intuitively but is not well-formed if W leads to an infinite loop.
  - If $\sigma(B)=F$, then $M(W,\sigma)=\{\sigma\}$
  - If $\sigma(B)=T$, then $M(W,\sigma)=M(S;W,\sigma)=M(W,M(S,\sigma))$.
- 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 $k=0,\ 1,\ \ldots$, 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: The k'th time we evaluate the loop test, we use state σ_k.
  - Now we can define $M(W,\sigma)=\{\sigma_k\}$ for the least k in which B is false, if there is one. If there isn't, we have an infinite loop.
  - Then $M(W,\sigma)$ is the (set containing the) first state in this sequence that satisfies $\neg B$, assuming there is such a state. (If there isn't, we have an infinite loop.)
- ***Example 3***: Let $W\equiv$ ***while*** $x<n$ ***do*** S ***od***, where the loop body $S\equiv x:=x+1;\ y:=y+y$, and let's calculate $M(W,\sigma)$ where $\sigma=\{x=0,\ n=3,\ y=1\}$.

- The behavior of S in an arbitrary state τ is $M(S, \tau[x \mapsto \alpha][y \mapsto \beta]) = \{\tau[x \mapsto \alpha + 1][y \mapsto 2\beta]\}$. Then 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, σ_3 is the first state that satisfies $x \geq n$, so $M(W, \sigma) = \{\sigma_3\} = \{\{x = 3, n = 3, y = 8\}\}$.
- Since we stop at σ_3, there's no need to calculate $M(S, \sigma_3) = \{\sigma_4\}$ to find that $\sigma_4 = \{x = 4, n = 3, y = 16\}$. (It's not incorrect, it's just not useful.)

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.
- (Recall that S starting in σ ***converges*** to τ / ***terminates*** in τ if $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ (operationally) or $M(S, \sigma) = \{\tau\}$ (denotationally). If S does not converge, it's said to ***diverge***.)
- ***Note***: Divergence is one way in which a program fails to successfully terminate.
- Rather than write $M(S, \sigma) = \varnothing$ for a divergent calculation, we'll use a "pseudo state" $\perp$ (pronounced "bottom"), so $M(S, \sigma) = \{\perp\}$ means that S doesn't terminate successfully in a final state. (Either it doesn't terminate, i.e., diverges, or it gets some sort of runtime error and halts.)
- We'll introduce other flavors of $\perp$ as we look at other ways to not get successful termination.

Divergence: The pseudo-state $\perp_d$ ("bottom sub-d")

- ***Notation***: Denotationally, $M(S, \sigma) = \{\perp_d\}$ means S diverges in σ. Note that although we're writing it in a place where you'd expect a memory state, $\perp_d$ is not an actual memory state; we'll call it a ***pseudo-state*** as apposed to an ***actual*** or ***real*** memory state like σ and τ.
- ***Notation***: Operationally, $\langle S, \sigma\rangle \rightarrow^* \langle E, \perp_d\rangle$ means that S starting in σ diverges. Again, we're not using $\perp_d$ 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 $M(S, \sigma) = \{\perp_d\}$ to say that S diverges, writing $\langle S, \sigma\rangle \rightarrow^* \langle E, \perp_d\rangle$ is notationally consistent[1].
- To determine when $M(W, \sigma) = \{\perp_d\}$, recall that in the previous section we looked at the series of states $\sigma_0, \sigma_1, \sigma_2, \ldots$ 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

[1] We should let $\rightarrow^*$ include a countably infinite number of steps, since you can only infer divergence by standing back after watching that may steps.

satisfies $\neg B$. We can now write $M(W, \sigma) = \{\perp_d\}$ to indicate that no state in the sequence satisfies $\neg B$.

- ***Example 4***: Let $W \equiv$ ***while*** T ***do skip od*** and σ be any state. Then $\langle W, \sigma\rangle \to \langle$ ***skip*** $; W, \sigma\rangle \to \langle W, \sigma\rangle$. Hence $M(W, \sigma) = \{\perp_d\}$. As a directed graph, this is a two-node cycle, $\langle W, \sigma\rangle \rightleftarrows \langle$ ***skip*** $; W, \sigma\rangle$.
- ***Example 5***: Let $W \equiv$ ***while*** $x \neq n$ ***do*** $x := x - 1$ ***od*** and let $\sigma = \{x = -1, n = 0\}$.
  - Let $\sigma_0 = \sigma = \{x = -1, n = 0\}$
  - Let $\{\sigma_1\} = M(x := x - 1, \sigma_0) = \{\sigma_0[x \mapsto -2]\} = \{\{x = -2, n = 0\}\}$
  - Let $\{\sigma_2\} = M(x := x - 1, \sigma_1) = \{\sigma_1[x \mapsto -3]\} = \{\{x = -3, n = 0\}\}$
  - In general, let $\{\sigma_k\} = M(x := x - 1, \sigma_{k-1}) = \{\{x = -k-1, n = 0\}\}$
  - Since every $\sigma_k \models x \neq n$, we have $M(W, \sigma) = \{\perp_d\}$.

E. Expressions With Runtime Errors: The pseudo-state $\perp_e$

- Using $\perp_d$ 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 σ, 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***: $\sigma(e) = \perp_e$ means evaluation of expression e in state σ causes a runtime error.
  - Here, $\perp_e$ 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 $\sigma(e) \in V \cup \{\perp_e\}$. Of course, some expressions never fail: $\sigma(2+2) \in \mathbb{Z}$, not just $\sigma(2+2) \in \mathbb{Z} \cup \{\perp_e\}$.
- ***Primary Failure***: The primitive values and operations being supported determine some set of basic runtime errors. For us, let's include:
  - *Array index out of bounds*: $\sigma(b[e]) = \perp_e$ if $\sigma(e) < 0$ or $\geq \sigma(size(b))$; similar for multiple dimensions.
  - *Division by zero*: $\sigma(e_1/e_2) = \sigma(e_1 \% e_2) = \perp_e$ if $\sigma(e_2) = 0$.
  - *Square root of negative number*: $\sigma(sqrt(e)) = \perp_e$ if $\sigma(e) < 0$.
  - ***Example 6***: $b[-1]$, $n/0$, and $sqrt(-1)$ fail for all σ. $b[k]$ 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 $\sigma(op\ e) = \perp_e$ if $\sigma(e) = \perp_e$.
  - If op is a binary operator, then $\sigma(e_1\ op\ e_2) = \perp_e$ if $\sigma(e_1)$ or $\sigma(e_2) = \perp_e$.
  - For a conditional expression, $\sigma($***if*** B ***then*** e_1 ***else*** e_2 ***fi*** $) = \perp_e$ if one of the following three situations occurs: (1) $\sigma(B) = \perp_e$ (2) $\sigma(B) = T$ and $\sigma(e_1) = \perp_e$ or (3) $\sigma(B) = F$ and $\sigma(e_2) = \perp_e$.

- As usual, ***if*** expressions are executed lazily: We don't worry about a hypothetical failure of the branch we don't evaluate.
- [2023-01-31] if σ(e) = $\perp_e$ then b[e]=$\perp_e$.

- ***Example 7***: $\sigma(x/y) = \perp_e$ when $\sigma(y) = 0$, but $\sigma(\textbf{\textit{if}}\ y = 0\ \textbf{\textit{then}}\ 0\ \textbf{\textit{else}}\ x/y\ \textbf{\textit{fi}})$ never=$\perp_e$.

F. Statements With Runtime Errors

- An expression that causes a runtime error causes the statement it appears in terminate unsuccessfully. We'll write $\langle S, \sigma\rangle \to \langle E, \perp_e\rangle$ for the operational semantics of such a statement. This use of $\perp_e$ as a (pseudo)-state is different from its use as a pseudo-value in $\sigma(e) = \perp_e$.
- ***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 $\sigma(e) = \perp_e$, then $\langle v := e, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(b[e_1])$ or $\sigma(e_2) = \perp_e$, then $\langle b[e_1] := e_2, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(B) = \perp_e$, then $\langle \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2\ \textbf{\textit{fi}}, \sigma\rangle \to \langle E, \perp_e\rangle$.
    - If $\sigma(B)$ does not fail, we continue with $\langle S_1, \sigma\rangle$ or $\langle S_2, \sigma\rangle$. Failure of those automatically cause failure of the overall conditional, so there's no need to treat failure of the true or false branch as a separate case.
  - If $\langle S_1, \sigma\rangle \to \langle E, \perp_e\rangle$ then $\langle S_1;\ S_2, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(B) = \perp_e$, then $\langle \textbf{\textit{while}}\ B\ \textbf{\textit{do}}\ S\ \textbf{\textit{od}}, \sigma\rangle \to \langle E, \perp_e\rangle$.
    - If $\sigma(B)$ is true, we continue with $\langle S;\ \textbf{\textit{while}}\ B\ \textbf{\textit{do}}\ S\ \textbf{\textit{od}}, \sigma\rangle$, and failure of this causes failure of the loop, so we don't need to treat failure of the loop body as a separate case.
- The pseudo-states $\perp_d$ and $\perp_e$ share some properties, so it's helpful to have a more general notation for "error".
- ***Notation***: $\perp$ refers generically to $\perp_d$ and/or $\perp_e$. In particular we use $\langle S, \sigma\rangle \to^* \langle E, \perp\rangle$ when it's not important which of $\perp_e$ or $\perp_d$ can occur. Similarly, $\perp \in M(S, \sigma)$ means $\langle S, \sigma\rangle$ leads to $\perp_d$ or $\perp_e$.

Properties and Consequences of $\perp$

- ***Trying to use*** $\perp$: Since we are writing $\perp$ in some of the places where an actual memory state would appear, we should be thorough and look at the other places states appear so we can extend those notions or notations.
  - $\perp$ is not a well-formed state.
  - When we say "for all states…" or "for some state…," we don't include $\perp$.
  - We can't add a binding to $\perp$: $\perp[v \mapsto \beta] = \perp$.
  - We can't bind a variable to $\perp$: $\sigma(v) \neq \perp$ and $\sigma[v \mapsto \perp] = \perp$.
  - 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": $\langle S, \perp\rangle \to^0 \langle E, \perp\rangle$.
  - Denotationally, we can't run a program in $\perp$: $M(S, \perp) = \{\perp\}$.
- From the properties we have, it follows that we can't evaluate something after generating $\perp$.
  - If $\langle S_1, \sigma\rangle \to \langle E, \perp\rangle$, then $\langle S_1; S_2, \sigma\rangle \to \langle E, \perp\rangle$.
  - 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 $W \equiv$ ***while*** B ***do*** S_1 ***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*** $\perp$: Note: $\perp$ never satisfies a predicate: $\perp \not\models p$ for all p, even if p is the constant T. In general, we now have three possibilities for a state trying to satisfy a predicate: $\sigma \models p$, $\sigma \models \neg p$, or $\sigma = \perp$. So $\sigma \not\models p$ implies $\sigma \models \neg p$ or $\sigma = \perp$, not just $\sigma \models \neg p$.
  - Note if $\sigma = \perp$, then $\sigma \not\models p$ and $\sigma \not\models \neg p$, both. The converse doesn't hold, though. E.g., if p is the conditional expression ***if*** $x = 0$ ***then*** $x/x = 0$ ***else*** $2 = 2$ ***fi***, then $\{x = 0\} \not\models p$ because $0/0$ causes a runtime error. For the false branch, $\{x = 1\} \models 2 = 2$, so $\{x = 1\} \not\models \neg(2 = 2)$, so $\{x = 1\} \not\models \neg p$.
- ***Logical negation and*** $\perp$: We still have that $\sigma \models \neg p$ implies $\sigma \not\models p$, but the converse no longer holds. It's possible now for the meaning of p to be $\perp$ (we'll look at this more in a moment), so $\sigma \not\models p$ doesn't imply $\sigma \models \neg p$. We need a new answer to the question of what $\sigma \models \neg p$ means. The solution is to treat $\neg p$ as shorthand for $p \to F$ 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 $\sigma \neq \perp$. (We can also derive F by defining $F \equiv T \neq T$). For all σ that are not $\perp$, we get $\sigma \models F \to F$, so $\sigma \models \neg F$.
- ***Generating*** $\perp$ ***while testing for satisfaction***: We certainly don't want to say $\{y = 0\} \models y/y = 1$. To handle the situation of $\sigma \models p$ when evaluation of p causes an error, we can add $\perp$ to the semantics of basic operations and tests.
  - For any *relation* (like less than, etc), we have $(\beta\ relation\ \delta)$ yields $\perp$ if β or δ are $\perp$.
  - For any binary *operation* (like addition, etc), we have $(\alpha\ \ operation\ \beta)$ yields $\perp$ if β or $\delta = \perp$.
  - Similarly for a unary operation, we have $(operation\ \perp)$ yields $\perp$.
  - [2023-01-31] this also applies to logical relations/operations.
- Some of the implications of this are reasonably intuitive: $(\perp\ plus\ one)$ 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 have $\sigma \models y/y = 1$ iff $\sigma(y/y) = \sigma(1)$ iff $\sigma(y) \div \sigma(y) = 1$, so if $\sigma(y) =$ some $\beta \neq 0$, then $\sigma \models y/y = 1$ iff $\beta \div \beta = 1$ iff $1 = 1$ iff T.
    - But if $\sigma(y) = 0$, then $\sigma \models y/y = 1$ iff $0 \div 0 = 1$ iff $\perp = 1$ iff $\perp$. Hence $\sigma \not\models y/y = 1$.
    - But also, since $\perp \neq 1$ is $\perp$, we also have $\sigma \not\models y/y \neq 1$.
    - So as expected, here σ satisfies neither $y/y = 1$ nor $y/y \neq 1$.=

