Nondeterministic Program (Continue)

(Denotational Semantics of Nondeterministic Program)

- Because of the nature of nondeterminism, we can end up with more than one state when we finish the execution of a nondeterministic program.
  - We use Σ to denote the set of all (well-formed) states. We denote $\Sigma_\perp = \Sigma \cup \{\perp\} = \Sigma \cup \{\perp_d, \perp_e\}$.
  - The denotational semantics of a deterministic program is a "single (possibly pseudo) state": $M(S,\sigma) = \{\tau\}$ where $\langle S,\sigma\rangle \to^* \langle E,\tau\rangle$ and $\tau \in \Sigma_\perp$.
  - The denotational semantics of a nondeterministic program is a set of (possibly pseudo) states: $M(S,\sigma) = \{\tau \in \Sigma_\perp \mid \langle S,\sigma\rangle \to^* \langle E,\tau\rangle\}$.

1. Calculate denotational semantics for each of the following nondeterministic statements and states.
   a. $S \equiv \textbf{if } T \to x \coloneqq 0 \ \square\ T \to x \coloneqq 1 \textbf{ fi}$, $\sigma = \emptyset$
      Since it is possible to have $\langle S,\sigma\rangle \to^* \langle E,\{x=0\}\rangle$ and $\langle S,\sigma\rangle \to^* \langle E,\{x=1\}\rangle$, thus $M(S,\sigma) = \big\{\{x=0\},\{x=1\}\big\}$.

   b. $S \equiv \textbf{if } F \to x \coloneqq 0 \ \square\ F \to x \coloneqq 1 \ \ \square\ T \to \textbf{skip fi}$, $\sigma = \emptyset$
      There is only one true guard, so $\langle S,\sigma\rangle \to \langle \textbf{skip},\sigma\rangle \to \langle E,\emptyset\rangle$, thus $M(S,\sigma) = \{\emptyset\}$.
      Remind that, we cannot omit the "{ }" when there is only $\emptyset$, "$M(S,\sigma) = \emptyset$" looks like we are saying S doesn't have denotational semantics.

   c. $S \equiv \textbf{if } x \ge y \to max \coloneqq x \ \square\ x \le y \to max \coloneqq y \textbf{ fi}$, $\sigma = \{x = 1,\ y = 1\}$
      No matter which arm, we have max gets bind with value 1. Thus $M(S,\sigma) = \big\{\{x = 1, y = 1, max = 1\}\big\}$.

   d. $S \equiv \textbf{do } x + y = 2 \to x \coloneqq y/x \ \square\ x + y = 2 \to x \coloneqq x + 1 \ \square\ x + y = 4 \to y \coloneqq x \ \square\ x + y = 4 \to x \coloneqq x - 1;\ y \coloneqq y - 1 \textbf{ od}$, and $\sigma = \{x = 1,\ y = 3\}$
      - After the first iteration of the $\textbf{do} - \textbf{od}$ loop, we can have these states: $\{x = 1, y = 1\}$, $\{x = 0, y = 2\}$.
      - After the second iteration of the $\textbf{do} - \textbf{od}$ loop, we can have these states: $\{x = 1, y = 1\}$, $\{x = 2, y = 1\}$, $\perp_e$ and $\{x = 1, y = 2\}$. We noticed that state $\{x = 1, y = 1\}$ appears again, and it is the only state that can pass some guard in the next iteration; if we keep evaluating S with $\{x = 1, y = 1\}$ the program will diverge.
      - Thus, $M(S,\sigma) = \big\{\{x = 2, y = 1\},\{x = 1, y = 2\},\perp_e,\perp_d\big\}$.

- From the above examples, we can see that:
  - The denotational semantics of a nondeterministic program can be only one state. Thus, we can say that "if $M(S,\sigma)$ is a set with more than one state, then S is nondeterministic" but its converse is not true.
  - For a nondeterministic program S, $M(S,\sigma)$ might contain pseudo states, and possibly more than one type of pseudo states.
  - We can say $\tau \in M(S,\sigma)$, it means that τ is one of the possible (pseudo) state after evaluating S in σ; there might be other states. Similarly, we can also say $\{\tau_1,\tau_2\} \subseteq M(S,\sigma)$, it means that τ_1,τ_2 are two of the possible (pseudo) states after evaluating S in σ.

2. Given three sorted (into non-decreasing order) $size - n$ arrays b_0, b_1 and b_2, are there valid indices k_0, k_1 and k_2 such that $b_0[k_0] = b_1[k_1] = b_2[k_2]$? Create a program in our language that can find a set of such k_0, k_1 and k_2 if they exist.

  - Here we use the most naïve idea: use k_0, k_1 and k_2 as pointers and scan three arrays until we find a solution or one of them reaches n.
  - If we use a deterministic program, I can expect that there will be many different cases in each iteration: we are looking at three values: $b_0[k_0]$, $b_1[k_1]$ and $b_2[k_2]$, each two of them can be $=, <$ or $>$.

  - It will be much easier if we only have two arrays, for example let's look at only $b_0[k_0]$ and $b_1[k_1]$:
    - If $b_0[k_0] = b_1[k_1]$, we are done.
    - If $b_0[k_0] < b_1[k_1]$, since arrays are sorted, increasing k_1 can only get a larger number in b_1, so we should increase k_0.
    - If $b_0[k_0] > b_1[k_1]$, like the previous case, we should increase k_1.

  - Thus, if we ignore b_2 and ignore the array size, we can come up with a partial solution to this question using a nondeterministic $\mathbf{do} - \mathbf{od}$ statement immediately:
$$\mathbf{do}\ b_0[k_0] < b_1[k_1] \to k_0 := k_0 + 1 \ \square\ b_0[k_0] > b_1[k_1] \to k_1 := k_1 + 1\ \mathbf{od}$$

  - Similarly, for the other combinations of values, we can come up with some other partial solutions:
$$\mathbf{do}\ b_0[k_0] < b_2[k_2] \to k_0 := k_0 + 1 \ \square\ b_0[k_0] > b_2[k_2] \to k_2 := k_2 + 1\ \mathbf{od}$$
$$\mathbf{do}\ b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1 \ \square\ b_1[k_1] > b_2[k_2] \to k_2 := k_2 + 1\ \mathbf{od}$$

  - Then, we can see the beauty of using a nondeterministic program: we don't need to worry about the overlapping cases, and it is very easy to combine partial solutions. We can come up with the following program:
$KKK \equiv$
$$\begin{aligned}
&k_0 := 0; k_1 := 0; k_2 := 0;\\
&\mathbf{do}\ b_0[k_0] < b_1[k_1] \to k_0 := k_0 + 1\\
&\ \ \square\ b_0[k_0] > b_1[k_1] \to k_1 := k_1 + 1\\
&\ \ \square\ b_0[k_0] < b_2[k_2] \to k_0 := k_0 + 1\\
&\ \ \square\ b_0[k_0] > b_2[k_2] \to k_2 := k_2 + 1\\
&\ \ \square\ b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1\\
&\ \ \square\ b_1[k_1] > b_2[k_2] \to k_2 := k_2 + 1\ \mathbf{od}
\end{aligned}$$

  - Before discussing the possible outcomes, let's modify KKK. First, let's merge some arms, since each $k_i := k_i + 1$ appears twice. And we have:
$KKK_1 \equiv$
$$\begin{aligned}
&k_0 := 0; k_1 := 0; k_2 := 0;\\
&\mathbf{do}\ b_0[k_0] < b_1[k_1] \lor b_0[k_0] < b_2[k_2] \to k_0 := k_0 + 1\\
&\ \ \square\ b_0[k_0] > b_1[k_1] \lor b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1\\
&\ \ \square\ b_0[k_0] > b_2[k_2] \lor b_1[k_1] > b_2[k_2] \to k_2 := k_2 + 1\ \ \mathbf{od}
\end{aligned}$$

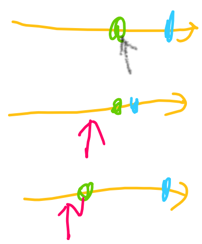
  - KKK_1 is already pretty good, but we can still simplify each guard. For example, in the first arm, we only need $b_0[k_0]$ less than something to increase k_0, so we can come up with:
$KKK_2 \equiv$
$$\begin{aligned}
&k_0 := 0; k_1 := 0; k_2 := 0;\\
&\mathbf{do}\ b_0[k_0] < b_1[k_1] \to k_0 := k_0 + 1\\
&\ \ \square\ b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1\\
&\ \ \square\ b_2[k_2] < b_0[k_0] \to k_2 := k_2 + 1\ \ \mathbf{od}
\end{aligned}$$

If all three guards are False, then we have $b_0[k_0] \geq b_1[k_1] \geq b_2[k_2] \geq b_0[k_0]$ which implies an equality among all three values.

- Does $\perp_e \in M(KKK_2, \sigma)$?
  It is possible. We can increase the value some k_i from $n-1$ to n and have runtime error in the next iteration.

- Now, let's take care of $\perp_e$. We can simply add some bounds checks in each guard, then:
  $KKK_3 \equiv$
  $$\begin{aligned} & k_0 := 0; k_1 := 0; k_2 := 0; \\ & \textbf{do } k_0 < n \wedge k_1 < n \wedge b_0[k_0] < b_1[k_1] \rightarrow k_0 := k_0 + 1 \\ & \ \ \square\ k_1 < n \wedge k_2 < n \wedge b_1[k_1] < b_2[k_2] \rightarrow k_1 := k_1 + 1 \\ & \ \ \square\ k_2 < n \wedge k_0 < n \wedge b_2[k_2] < b_0[k_0] \rightarrow k_2 := k_2 + 1 \ \textbf{od} \end{aligned}$$

- What can we get if we calculate $M(KKK_3, \sigma)$, where b_0, b_1 and b_2 are already bind in the state σ?
  - If there is at least one set of k_0, k_1 and k_2 that satisfies the requirements, why is there a $\tau \in M(KKK_3, \sigma)$ with $\tau(b_0[k_0]) = \tau(b_1[k_1]) = \tau(b_2[k_2])$?
    For any $\tau \in M(KKK_3, \sigma)$, where none of $\tau(k_i) = n$; we must have $\tau(b_0[k_0]) = \tau(b_1[k_1]) = \tau(b_2[k_2])$, otherwise at least one of the guards is true.



  - If there are multiple solutions, does the program find all of them?
    No. For each k_i, its value can increase by at most 1 after each iteration, so to reach the set of "larger" solutions we must go through the set of "smaller" solutions. This is true for all i, so every k_i will reach the smaller solution first, and when they reach the smaller solution, the loop terminates.

  - Does $\perp_e \in M(KKK_3, \sigma)$?
    No.

  - For any $\tau \in M(KKK_3, \sigma)$, is it possible that $\exists 0 \leq i \leq 2. \tau(k_i) > n$?
    None of k_0, k_1 and k_2 can be bind with a value $> n$; because the value of some k_i can only increases after it passes the guard, and if the evaluation of $k_i \geq n$, then evaluating a guard will return $\perp_e$.

  - Does $M(KKK_3, \sigma)$ contain more than one state?
    If there is at least one solution, then $M(KKK_3, \sigma) = \{\tau\}$ is only one state, and it contains the smallest solution. If there is no solution, then $M(KKK_3, \sigma)$ will be a set of states, where each state in the set contains one k_i binds with n.

Correctness Triples

- A **correctness triple** (a.k.a. "Hoare triple," after C.A.R. Hoare; or usually simplified to **"triple"**), written as $\{p\}\ S\ \{q\}$ is a program S plus its specification predicates p and q. Note that, $\{p\}$ and $\{q\}$ are not states, they are predicates wrapped in "{ }".
  - The **precondition** p (not "$\{p\}$") describes what we're assuming is true about the state before the program begins.
  - The **postcondition** q (not "$\{q\}$") describes what should be true about the state after the program terminates.
  - Informally, a triple $\{p\}\ S\ \{q\}$ means "if program S runs in a state that satisfies p, then we can expect the execution of S satisfies q".

3. Here are some correctness triple examples.
   a. $\{x = 2\}\ x \coloneqq x + 3\ \{x < 6\}$
   b. $\{y = 2\}\ x \coloneqq 2; x \coloneqq 2 + x\ \{x = 4\}$
   c. $\{x \geq 0\}\ S\ \{y^2 \leq x < (y + 1)^2\}$

(Satisfaction and Validity)

- Informally, a state σ **satisfies** a triple $\{p\}\ S\ \{q\}$, written as $\sigma \vDash \{p\}\ S\ \{q\}$, it means "**if** σ satisfies p, **then** after running S in σ we can get a state τ who satisfies q".
  - If $\sigma \nvDash p$, we don't claim anything about the execution of S in σ. For example, if have state $\{x = -5\}$ and triple $\{x \geq 0\}\ S\ \{y^2 \leq x < (y + 1)^2\}$. Running S with $\{x = -5\}$ might give a runtime error or diverge or give a state that doesn't satisfy the postcondition, but here we don't consider any of those situations since $\{x = -5\}$ doesn't satisfy the precondition $x \geq 0$, so $\{x = -5\} \vDash \{x \geq 0\}\ S\ \{y^2 \leq x < (y + 1)^2\}$.
  - From the above example, we can see that "$\sigma \vDash \{p\}\ S\ \{q\}$" might not give us much information about executing S in σ. But on the other hand, "$\sigma \nvDash \{p\}\ S\ \{q\}$" shows that $\sigma \vDash p$ and the execution of S in σ doesn't give us a state satisfies q.

- If triple $\{p\}\ S\ \{q\}$ is satisfied by all states, then we say $\{p\}\ S\ \{q\}$ is **valid**, written as $\vDash \{p\}\ S\ \{q\}$.

- To sum up, we have:
  - $\sigma \vDash \{p\}\ S\ \{q\}$ means σ satisfies the triple.
  - $\sigma \nvDash \{p\}\ S\ \{q\}$ means σ does not satisfy the triple.
  - $\vDash \{p\}\ S\ \{q\}$ means the triple is valid: $\forall \sigma. \sigma \vDash \{p\}\ S\ \{q\}$.
  - $\nvDash \{p\}\ S\ \{q\}$ means the triple is invalid: $\exists \sigma. \sigma \nvDash \{p\}\ S\ \{q\}$.

4. True or False
   a. $\{x = -5\} \vDash \{x > 0\}\ x := x + 1\ \{x > 0\}$ True
   b. $\vDash \{x > 0\}\ x := x + 1\ \{x > 0\}$ True
   c. $\{x = -5\} \vDash \{x > 0\}\ x := x - 1\ \{x > 0\}$ True
   d. $\vDash \{x > 0\}\ x := x - 1\ \{x > 0\}$ False, we can find $\{x = 1\} \nvDash \{x > 0\}\ x := x - 1\ \{x > 0\}$