# *Sequential Nondeterminism*

## *CS 536: Science of Programming, Spring 2023*

## *A. Why*

- Nondeterminism can help us avoid unnecessary determinism.
- Nondeterminism can help us develop programs without worrying about overlapping cases.

## *B. Objectives*

At the end of this class you should know

- The syntax and operational and denotational semantics of nondeterministic statements.

## *C. Avoiding Unnecessary Design Choices Using Nondeterminism*

- When writing programs, it's hard enough concentrating on the decisions we ***must*** make at any given time, so it's helpful to avoid making decisions we don't have to make.
- ***Example 1***: A very simple example is a statement that sets $max$ to the larger of $x$ and $y$. It doesn't really matter which of the following two statements we use. They're written differently but behave the same:
  - ***if*** $x \geq y$ ***then*** $max := x$ ***else*** $max := y$ ***fi***
  - ***if*** $y \geq x$ ***then*** $max := y$ ***else*** $max := x$ ***fi***
- The difference is when $x = y$, the first statement sets $max := x$; the second sets $max := y$. It doesn't matter which one of these we choose, we just have to pick one.
- Our standard ***if-else*** statement is ***deterministic***: It can only behave one way. A nondeterministic ***if-fi*** will specify that one of $max := x$ and $max := y$ has to be run, but it won't say how we choose which one.
  - We don't plan to execute our programs nondeterministically; we design programs using nondeterminism in order to delay making unnecessary decisions about the order in which our code makes choices.
  - When we make the code more concrete by rewriting it using everyday deterministic code, then we'll decide which way to write it.

## *D. Nondeterministic if-fi*

- ***Syntax***: ***if*** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \ldots \ \square \ B_n \rightarrow S_n$ ***fi***
  - The box symbols separate the different arms, like commas in an ordered $n$-tuple.
  - Don't confuse these right arrows with ones in other contexts (implication operator and single-step execution).

 

- ***Definition***: Each $B_i \rightarrow S_i$ clause is a ***guarded command***. The ***guard*** $B_i$ tells us when it's okay to run $S_i$.
- ***Informal semantics***
  - If none of the guard tests $B_1$, $B_2$, ..., $B_n$ are true, abort with a runtime error.
  - If exactly one guard $B_i$ is true then execute $S_i$.
  - If more than one guard is true, then select a corresponding statement and execute it.
    - The selection is made nondeterministically (unpredictably); we'll discuss this more soon.
- ***Example 2***: ***if*** $x \geq y \rightarrow max := x \ \square \ y \geq x \rightarrow max := y$ ***fi*** sets $max$ to the larger of $x$ and $y$.
  - If only one of $x \geq y$ and $y \geq x$ is true, we execute its corresponding assignment.
  - If both are true, we choose one of them and execute its assignment.
- In this example, the two arms set $max$ to the same value when $x = y$, so it doesn't matter which one gets used.
- In more general examples, the different arms might behave differently but as long as each gets us to where we're going, we don't care which one gets chosen.
  - E.g., say we have an ***if-fi*** with two arms; one arm sets a variable $z := 0$; the other arm sets $z := 1$. If, for correctness's sake, we need $z \geq 0$ after the ***if-fi***, then this is fine. (If we needed $even(z)$, for example, we'd have a bug.)
- We can also have ***if-fi*** statements that never have to make a nondeterministic choice.
  - ***Example 3***: Our usual deterministic ***if*** $B$ ***then*** $S_1$ ***else*** $S_2$ ***fi*** can be written as ***if*** $B \rightarrow S_1 \ \square \ \neg B \rightarrow S_2$ ***fi***.

## *E. Nondeterministic Choices are Unpredictable*

- For us, "nondeterministic" means "unpredictable".
- Let *flip* ≡ ***if*** $T \rightarrow x := 0 \ \square \ T \rightarrow x := 1$ ***fi***, which sets $x$ to either $0$ or $1$. I've called it *flip* because it's similar to a coin flip, but it's not identical.
  - With a real coin flip, you expect a 50-50 chance of getting $0$ or $1$, but since *flip* is nondeterministic, its behavior is completely unpredictable.
  - A thousand calls of *flip* might give us anything: all 0's, all 1's, some pattern, random 500 heads and 500 tails, etc.
- ***Nondeterminism shouldn't affect correctness***: We write nondeterministic code when we don't want to worry about how choices are made: We only want to worry about producing correct results given that a choice has been made.
  - E.g., code written using *flip* should produce a correct final state whether we get heads or tails. Of course, eventually, we'll replace *flip* with a deterministic coin-flipping routine, and at that point we'll have to worry about the fairness of the deterministic routine.

 

## F. Nondeterministic Loop

- Nondeterministic loops are very similar to nondeterministic conditionals, both in syntax and semantics. We can derive nondeterministic loops using nondeterministic if and a ***while*** loop.
- ***Syntax***: ***do*** $B_1 \rightarrow S_1 \;\square\; B_2 \rightarrow S_2 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***od***
- ***Informal semantics***:
  - At the top of the loop, check for any true guards.
  - If no guard is true, the loop terminates.
  - If exactly one guard is true, execute its corresponding statement and jump to the top of the loop.
  - If more than one guard is true, select one of the corresponding guarded statements and execute it. (The choice is nondeterministic.) Once we finish the guarded statement, jump to the top of the loop.
- A nondeterministic do loop is equivalent to a regular ***while*** loop (with a nondeterministic test but) with a nondeterministic if body. Let $BB \equiv (B_1 \vee B_2 \ldots \vee B_n)$ be the disjunction of the guards, then ***do*** $B_1 \rightarrow S_1 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***od*** behaves like ***while*** $BB$ ***do if*** $B_1 \rightarrow S_1 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***fi od***.

## G. Operational Semantics of Nondeterministic *if-fi*

- Let $IF \equiv$ ***if*** $B_1 \rightarrow S_1 \;\square\; B_2 \rightarrow S_2 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***fi*** and let $BB \equiv B_1 \vee B_2 \vee \ldots B_n$.
- To evaluate $IF$,
  - If evaluation of any guard fails $(\sigma(BB) = \perp_e)$, then $IF$ causes an error: $\langle IF, \sigma \rangle \rightarrow \langle E, \perp_e \rangle$.
  - If none of the guards are satisfied $(\sigma(BB) = F)$, then $IF$ causes an error: $\langle IF, \sigma \rangle \rightarrow \langle E, \perp_e \rangle$.
  - If one or more guarded commands $B_i \rightarrow S_i$ have $\sigma(B_i) = T$, then one such $i$ is chosen nondeterministically and we jump to the beginning of $S_i$: $\langle IF, \sigma \rangle \rightarrow \langle S_i, \sigma \rangle$.

## H. Operational Semantics of Nondeterministic *do-od*

- Let $DO \equiv$ ***do*** $B_1 \rightarrow S_1 \;\square\; B_2 \rightarrow S_2 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***od*** and let $BB \equiv B_1 \vee B_2 \vee \ldots B_n$.
- Evaluation of $DO$ is very similar to evaluation if $IF$:
  - If evaluation of any guard fails $(\sigma(BB) = \perp_e)$, then $DO$ causes an error: $\langle DO, \sigma \rangle \rightarrow \langle E, \perp_e \rangle$.
  - If none of the guards are satisfied $(\sigma(BB) = F)$, then the loop halts: $\langle DO, \sigma \rangle \rightarrow \langle E, \sigma \rangle$.
  - If one or more guarded commands $B_i \rightarrow S_i$ have $\sigma(B_i) = T$, then one such $i$ is chosen nondeterministically and we jump to the beginning of $S_i$; after it completes, we'll jump back to the top of the loop: $\langle DO, \sigma \rangle \rightarrow \langle S_i; DO, \sigma \rangle$.

## I. Denotational Semantics of Nondeterministic Programs

- ***Notation***:
  - $\Sigma$ is the set of all states (that proper for whatever we happen to be discussing at that time).
  - $\Sigma_\perp = \Sigma \cup \{\text{all flavors of } \perp\} = \Sigma \cup \{\perp_d, \perp_e\}$ right now; other versions can be added later.

 

- As before, writing $\tau = \perp$ means $\tau \in \{\perp_d, \perp_e\}$, so it refers ambiguously to one or the other.
- For a nondeterministic program, to get its denotational semantics, we have to collect all the possible final states (or the pseudo-state $\perp$): $M(S, \sigma) = \{\tau \in \Sigma_\perp \mid \langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle\}$.
- For a deterministic program, there is only one such $\tau$, so this simplifies to our earlier definition: $M(S, \sigma) = \{\tau\}$ where $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ and $\tau \in \Sigma_\perp$.
- ***Example 4***: Let $S \equiv$ ***if*** $T \rightarrow x := 0 \ \square \ T \rightarrow x := 1$ ***fi***. Then $\langle S, \varnothing\rangle \rightarrow^* \langle E, x = 0\rangle$ and $\langle S, \varnothing\rangle \rightarrow^* \langle E, x = 1\rangle$ are both possible, and $M(S, \sigma) = \{\{x = 0\}, \{x = 1\}\}$. (Be careful not to write this as $\{\{x = 0, x = 1\}\}$, which is a set containing a single, ill-formed state.) For any particular execution of $S$ in a σ, we'll get exactly one of these final states.
- ***Notation***: For convenience, most times we can still abbreviate $M(S, \sigma) = \{\tau\}$ to $M(S, \sigma) = \tau$. But let's agree not to shorten $M($***skip***$, \varnothing) = \{\varnothing\}$ to $M($***skip***$, \varnothing) = \varnothing$, since it might look like we're claiming that $M($***skip***$, \varnothing)$ has no final state — it does, the empty state.
- A nondeterministic program doesn't have to have multiple final states.
  - ***Example 5***: The *max* program from Example 2 has only one final state. Let $Max \equiv$ ***if*** $x \geq y \rightarrow max := x \ \square \ y \geq x \rightarrow max := y$ ***fi***, in the nondeterministic case, where $x = y$, both possible execution paths take us to the same state: $M(Max, \{x = \alpha, y = \alpha\}) = \{\{x = \alpha, y = \alpha, max = \alpha\}\}$.
  - ***Note***: To keep from confusing the graders, avoid writing things that look like multisets, such as "$\{\tau, \tau\}$ where $\tau = \{x = \alpha, y = \alpha, max = \alpha\}$".
- For arbitrary $S$, if $M(S, \sigma)$ has more than one member, then $S$ is nondeterministic. The *Max* program shows us that the converse doesn't hold: If $M(S, \sigma)$ has just one member, $S$ still could be nondeterministic. Note also that the size of $M(S, \sigma)$ can vary depending on σ.
  - ***Example 6***: If $S \equiv$ ***if*** $x \geq 0 \rightarrow x := x * x \ \square \ x \leq 8 \rightarrow x := -x$ ***fi***, then $M(S, \{x = 0\}) = \{\{x = 0\}\}$, but $M(S, \{x = 3\}) = \{\{x = 9\}, \{x = -3\}\}$.

### *Difference between M(S,σ) = {τ} and τ ∈ M(S,σ)*

- There's a difference between $M(S, \sigma) = \{\tau\}$ and $\tau \in M(S, \sigma)$. They both say that $\tau$ can be a final state, but $M(S, \sigma) = \{\tau\}$ says there's only one final state, but $\tau \in M(S, \sigma)$ leaves open the possibility that there are other final states.
- In particular, $M(S, \sigma) = \{\perp\}$ says $S$ always causes an error whereas $\perp \in M(S, \sigma)$ says that $S$ might cause an error. Remember that when we write $\perp$, we're being ambiguous as to whether we mean $\perp_d$ or $\perp_e$. If multiple kinds of failure are possible, we should say so, as in $M(S, \sigma) = \{\perp_d, \perp_e\}$ or $\{\perp_d, \perp_e\} \subseteq M(S, \sigma)$.

## *J. Why Use Nondeterministic programs?*

- Without having defined program correctness yet, the discussion here will be informal.

## *Reason 1: Nondeterminism makes it easer to combine partial solutions*

- With nondeterministic code, it's straightforward to combine partial solutions to a problem to form a larger solution. This means we can solve a large problem by solving smaller instances of it and combining them.
- ***Example 7:*** Let's solve the *Max* problem. Say we specify "*Max* takes $x$ and $y$ and (without changing them), sets *max* to the larger of $x$ and $y$."
  - Since the program has to end with $max = x$ or $max = y$, one way to approach is to ask "When does $max := x$ work?" and "When does $max := y$ work?".
  - Since $max := x$ is correct exactly when $x \geq y$, the program ***if*** $x \geq y \rightarrow max := x$ ***fi*** is (partially) correct. (It's not totally correct, since it fails if $x < y$.)
  - Similarly, since $max := y$ is correct exactly when $x \leq y$, the program ***if*** $x \leq y \rightarrow max := y$ ***fi*** is also (partially) correct.
  - We can combine the two partial solutions and get

    ***if*** $x \geq y \rightarrow max := x$
    $\square$ $x \leq y \rightarrow max := y$
    ***fi***
  - This program works when $x \geq y$ or $x \leq y$, and since that covers all possibilities, our program is done.

## *Reason 2: Nondeterminism makes it easier to find overlapping solutions but delay worrying about them*

- ***Find overlapping solutions***: When approaching a problem nondeterministically, you can concentrate on discovering partial solutions but put off decisions about which ones might be better. Here's a vague example: Say we want to process a stream of widgets; the red ones can be processed using techniques A or B; the blue ones can be processed using using techniques B or C. Very roughly, ***do*** $red \rightarrow A \square red \rightarrow B \square blue \rightarrow$ B $\square$ $blue \rightarrow C$ ***od***.
  - Since we're using a nondeterministic approach, we can improve the individual cases separately because we don't have to decide immediately about which process we want to run. For example, we might discover that red widgets can also be handled by process $B_1$: ***do*** $red \rightarrow A \square red \rightarrow B \square red \rightarrow B_1 \square blue \rightarrow$ B $\square$ $blue \rightarrow C$ ***od***.
- ***Delay worry about overlapping cases***: In nondeterministic ***if***/***do***, the order of the guarded commands makes no difference, so we can it doesn't matter if guards overlap in what states satisfy them. That means we can write the code nondeterministically, with overlapping cases, and not worry about the overlap. We can remove the overlap when we rewrite the code deterministically to run it.
  - Going back to the widget example, we found $B_1$ for red widgets but don't have to immediately ponder questions like "Should we use B or $B_1$ for red widgets?" and "Can we improve B for blue widgets (using the insight that gave us $B_1$ for red widgets?"

 

- ***Example 8:*** For a simpler example, let's take the *Max* program yet again.
  - Both of these programs are correct:

    ***if*** $x \geq y \rightarrow max := x \;\square\; x \leq y \rightarrow max := y$ ***fi***

    ***if*** $x \leq y \rightarrow max := y \;\square\; x \geq y \rightarrow max := x$ ***fi***
  - Since the programs behave identically when $x = y$, it doesn't matter if we drop that case from one of the tests, say the second, which yields

    ***if*** $x \geq y \rightarrow max := x \;\square\; x < y \rightarrow max := y$ ***fi***

    ***if*** $x \leq y \rightarrow max := y \;\square\; x > y \rightarrow max := x$ ***fi***
  - Introducing the asymmetry makes the code correspond to the deterministic statements

    ***if*** $x \geq y$ ***then*** $max := x$ ***else*** $max := y$ ***fi***

    ***if*** $x \leq y$ ***then*** $max := y$ ***else*** $max := x$ ***fi***
- ***Example 9:*** A similar pair of examples of introducing asymmetry takes

  ***if*** $x \geq 0 \rightarrow y := sqrt(x) \;\square\; x \leq 0 \rightarrow y := 0$ ***fi***

  to ***if*** $x \geq 0$ ***then*** $y := sqrt(x)$ ***else*** $y := 0$ ***fi*** or

  ***if*** $x > 0$ ***then*** $y := sqrt(x)$ ***else*** $y := 0$ ***fi***

## *K. Example 10: Array Value Matching*

- As an example of how nondeterministic code can help us write programs, let's look at an array-matching problem. We're given three arrays, *b0, b1*, and *b2*, all of length $n$ and all sorted in non-descending order. The goal is to find indexes *k0, k1*, and *k2* such that $b0[k0] = b1[k1] = b2[k2]$ if such values exist.
  - But what if no such *k0*, *k1*, and *k2* exist? One solution is to terminate with $k0 = k1 = k2 = n$. This certainly works, but we need to test each index before testing its value. Assuming "∧" is short-circuiting, we test $k0 < n \wedge k1 < n \wedge b0[k0] < b1[k1]$ to make sure that *k0* and *k1* are in range before using them as indexes.
  - We'll use an alternate approach by having sentinels: We'll assume $b0[n], b1[n],$ and $b2[n]$ all equal +∞ (positive infinity). This lets us write tests like $b0[k0] < b1[k1]$ without having to test for *k0* or $k1 = n$.
- How does the program work? If we set $k0 = k1 = k2 = 0$ initially, then we have to increment *k0* or *k1* or *k2* until we find a match.
- Let's study one pair of indexes, say *k0* and *k1*. There are three cases:
  1. $b0[k0] < b1[k1]$. If this happens, we should increment *k0*. Since the arrays are sorted by ≤, incrementing *k1* can't possibly result in $b0[k0] = b1[k1]$, whereas incrementing *k0* might.
  2. $b0[k0] > b1[k1]$. Symmetrically, if this happens, we should increment *k1*.

 

3. $b0[k0] = b1[k1]$. If this happens, we don't want to do anything, since we have a possible match. (Of course, we still need $b1[k1] = b2[k2]$ or $b0[k0] = b2[k2]$ — they're equivalent in this case.)

- If we write this up as a nondeterministic ***if-fi***, we get

  ***if*** $b0[k0] < b1[k1] \rightarrow k0 := k0 + 1 \;\square\; b0[k0] > b1[k1] \rightarrow k1 := k1 + 1$ ***fi***

- Repeating for the other two pairs of indexes, we get

  ***if*** $b1[k1] < b2[k2] \rightarrow k1 := k1 + 1 \;\square\; b1[k1] > b2[k2] \rightarrow k2 := k2 + 1$ ***fi***

  ***if*** $b2[k2] < b0[k0] \rightarrow k2 := k2 + 1 \;\square\; b2[k2] > b0[k0] \rightarrow k0 := k0 + 1$ ***fi***

- If we repeat these three ***if-fi*** statements until none of the < or > cases apply, then we're guaranteed that = holds between each pair. We can combine the six cases above into:

  // Program 10(a)
  //
  ***do*** $b0[k0] < b1[k1] \rightarrow k0 := k0 + 1$
  $\square\;\; b0[k0] > b1[k1] \rightarrow k1 := k1 + 1$
  $\square\;\; b1[k1] < b2[k2] \rightarrow k1 := k1 + 1$
  $\square\;\; b1[k1] > b2[k2] \rightarrow k2 := k2 + 1$
  $\square\;\; b0[k0] < b2[k2] \rightarrow k0 := k0 + 1$
  $\square\;\; b0[k0] > b2[k2] \rightarrow k2 := k2 + 1$
  ***od***

- If none of the loop guards apply, the ≤ and ≥ combine and ensure $b0[k0] = b1[k1] = b2[k2]$.
- The code can be cleaned up a couple of ways. The obvious one is to combine guards that guard the same command:

  // Program 10(b)
  //
  ***do*** $b0[k0] < b1[k1] \vee b0[k0] < b2[k2] \rightarrow k0 := k0 + 1$
  $\square\;\; b1[k1] < b2[k2] \vee b1[k1] < b0[k0] \rightarrow k1 := k1 + 1$
  $\square\;\; b2[k2] < b0[k0] \vee b2[k2] < b1[k1] \rightarrow k2 := k2 + 1$
  ***od***

- Another way is to note that if we don't have $b0[k0] = b1[k1] = b2[k2]$, then there must be a < relation between two of the three values. This gives us

  // Program 11(c)
  //
  ***do*** $b0[k0] < b1[k1] \rightarrow k0 := k0 + 1$
  $\square\;\; b1[k1] < b2[k2] \rightarrow k1 := k1 + 1$
  $\square\;\; b2[k2] < b0[k0] \rightarrow k2 := k2 + 1$
  ***od***

- For all three of the guards to be false, we need $b0[k0] \geq b1[k1] \geq b2[k2] \geq b0[k0]$, which only happens when $b0[k0] = b1[k1] = b2[k2]$.

 