# *Sequential Nondeterminism*

## *CS 536: Science of Programming, Fall 2022*

## *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 have to make at any given time, so it's helpful to put off decisions we don't have to make.
- Our standard ***if-else*** statement is ***deterministic***: It can only behave one way.
  - When we write an ***if-else***, we have to make choices that sometimes don't really matter.
- ***Example 1***: A very simple example is a statement that sets *max* to the max of *x* and *y*. It doesn't really matter which of the following two we use. They're written differently but behave the same:
  - $S1 \equiv$ ***if*** $x \geq y$ ***then*** $max := x$ ***else*** $max := y$ ***fi***
  - $S2 \equiv$ ***if*** $y \geq x$ ***then*** $max := y$ ***else*** $max := x$ ***fi***
- The difference is when $x = y$, the *S1* always sets $max := x$ whereas *S2* always sets $max := y$.
  - Since $x = y$, it doesn't matter which statement we use, we just have to pick one.
- 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 actually execute our programs nondeterministically; we design programs using nondeterminism. When we get to writing out deterministic program code, then we'll decide which ways to write things.

## *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 $B_k \rightarrow S_k$ clause is a ***guarded command***. The ***guard*** $B_k$ tells us when it's okay to run $S_k$.

- ***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 $B_k$ is true then execute the statement $S_k$ that it guards.
  - If more than one guard is true, select one of the guarded statements 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. For example, say we have predicates $p$ and $q$, where if $p$ is true we need to set $x := 1;$ if $q$ is true we need to set $x$ to $2$. But what if both $p$ and $q$ are true? If the answer is it doesn't matter whether we set $x$ to $1$ or $2$ in this case, then the statement that fully models this (and only this) is
  - "If only $p$ is true set $x := 1$; if only $q$ is true set $x := 2$; if both are true, set $x$ to $1$ or $2$"
- If later during a situation comes up where we realize, say, $x = 1$ results in faster execution, we can revisit our nondeterministic conditional statement and modify it.
- The guards don't have to logically overlap. If they don't, we have a deterministic ***if-else***.
- ***Example 3***: Our usual deterministic ***if*** $B$ ***then*** $S_1$ ***else*** $S_2$ ***fi*** can be written ***if*** $B \rightarrow S_1 \;\square\; \neg B \rightarrow S_2$ ***fi***. Since $B$ and $\neg B$ are never true simultaneously, there's no choice involved.

## *E. Nondeterministic Choices are Unpredictable*

- For us, ***nondeterministic choice*** means ***unpredictable choice***. There's no notion of ***probable choice*** and no notion of ***random fair choice***.
- 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 the same. 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 need to worry about how choices are made, we only need 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 whether to model a fair coin flip or not.

## *F. Nondeterministic Loop*

- Nondeterministic loops are very similar to nondeterministic conditionals, both in syntax and semantics.
- ***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.
- We can derive nondeterministic loops using a ***while*** loop with an nondeterministic ***if-fi*** body. The while loop test is deterministic, so all the nondeterminism is in the ***if-fi***.
  - Take ***do*** $B_1 \rightarrow S_1 \square \ldots \square B_n \rightarrow S_n$ ***od***. Let $BB \equiv (B_1 \vee B_2 \ldots \vee B_n)$ be the disjunction of the guards, then or ***do-od*** loop can be implemented as ***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 ($\sigma(BB) = \bot_e$), then *IF* causes an error: $\langle IF, \sigma\rangle \rightarrow \langle E, \bot_e\rangle$.
    - It doesn't matter if any non-failing guard is true: E.g., runtime error is always what we get on running ***if*** $T \rightarrow S_1 \square sqrt(-1) \neq 0 \rightarrow S_2$ ***fi***.
  - If all the guards evaluate successfully to true or false, then
    - If none of the guards are satisfied ($\sigma(BB) = F$), then *IF* causes an error: $\langle IF, \sigma\rangle \rightarrow \langle E, \bot_e\rangle$.
    - If one or more guarded commands $B_k \rightarrow S_k$ have $\sigma(B_k) = T$, then one such $k$ is chosen nondeterministically and we jump to the beginning of $S_k$: $\langle IF, \sigma\rangle \rightarrow \langle S_k, \sigma\rangle$.

## *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 of *IF*:
  - If evaluation of any guard fails ($\sigma(BB) = \bot_e$), then *DO* causes an error: $\langle DO, \sigma\rangle \rightarrow \langle E, \bot_e\rangle$.
  - 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 $B_k \rightarrow S_k$ have $\sigma(B_k) = T$, then one such $k$ is chosen nondeterministically and we jump to the beginning of $S_k$; after it completes, we'll jump back to the top of the loop: $\langle DO, \sigma\rangle \rightarrow \langle S_k;\, DO, \sigma\rangle$.

## *I. Denotational Semantics of Nondeterministic Programs*

- ***Notation***:
  - Define $\Sigma$ to be the set of all states (that are proper) for whatever we happen to be discussing at that moment. Define $\Sigma_\bot = \Sigma \cup \{$all flavors of $\bot\} = \Sigma \cup \{\bot_d, \bot_e\}$; whenever a new flavor of $\bot$ is introduced, we add it to $\Sigma_\bot$
  - If we just write $\bot$, then we mean one or the other of $\bot_d$ or $\bot_e$, with the difference being unimportant. So $\{\bot\}$ means "one kind of error", either $\{\bot_d\}$ or $\{\bot_e\}$ as appropriate. To get "both kinds of error", we write $\{\bot_d, \bot_e\}$.
- For the denotational semantics of a nondeterministic program,, we collect all the possible final states (or pseudo-states if we get $\bot$): $M(S, \sigma) = \{\tau \in \Sigma_\bot \mid \langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle\}$.
- 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 $\tau\in\Sigma_\bot$.
- ***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.) Note $M(S, \sigma)$ describes the set of all possible final states. For any particular execution of $S$ in $\sigma$, we'll get one (and only one) of these final states. A deterministic program always has an $M(S, \sigma)$ of size one.
- ***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***$, \varnothing) = \varnothing$. $M($***skip***$, \varnothing) = \{\varnothing\}$ makes it clear that there is exactly one final state, but $M($***skip***$, \varnothing) = \varnothing$ might look like we're claiming ***skip*** has no final state, which is incorrect.
- With nondeterministic programs, if multiple execution paths lead to the same result (i.e., there are two distinct paths $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$), then the final state only appears once in $M(S, \sigma)$.
  - Another way to say this is that $M(S, \sigma)$ is a set, not a multiset.
- Though (program has > 1 final state) $\Rightarrow$ (program is nondeterministic), the converse is not true.
- ***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***. When $x = y$, both possible execution paths take us to the same state, but the final state only appears once in the results: $M(max, \{x = \beta, y = \beta\}) = \{\{x = \beta, y = \beta, max = \beta\}\}$.
- Another point: the size of $M(S, \sigma)$ can vary depending on $\sigma$.
  - ***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 big 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, $M(S, \sigma) = \{\bot\}$ says $S$ always causes an error whereas $\bot \in M(S, \sigma)$ says that $S$ might cause an error (and it's silent on whether non-$\bot$ can happen). Note $M(S, \sigma) = \{\bot\}$ says that $S$ always leads to the same error. If both kinds of failure are possible, we should be explicit: $M(S, \sigma) = \{\bot_d, \bot_e\}$ if only errors can occur and $\{\bot_d, \bot_e\} \subseteq M(S, \sigma)$ if we don't want to talk about other possible behaviors.

## *J. Why Use Nondeterministic programs?*

- The distinguishing feature of nondeterministic code is that it doesn't specify what happens when multiple cases overlap when making a decision.

### *Reason 2: Nondeterminism Makes It Easy 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. You can come up with ***if*** $B_1 \rightarrow S_1 \,\square\, B_2 \rightarrow S_2$ ***fi*** by noticing that ***if*** $B_1 \rightarrow S_1$ ***fi*** and ***if*** $B_2 \rightarrow S_2$ ***fi*** are partial solutions to the problem.
- ***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 the problem is by asking "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 correct.
  - Similarly, since $max := y$ is correct exactly when $x \leq y$, the program ***if*** $x \leq y \rightarrow max := y$ ***fi*** is also correct.
  - We can combine the two partial solutions and get

```
if x ≥ y → max := x
□ x ≤ y → max := y
fi
```

  - We can stop combining partial cases when the disjunction of the guards covers all possible start states. With *max*, $(x \geq y \vee y \geq x) \Leftrightarrow T$, so there are no other cases to solve.

### *Reason 2: Nondeterminism Makes it Easy to Handle Overlapping Cases*

- The reason that we can combine partial solutions is that if two guards overlap (some state(s) satisfy both guards), we can combine the partial solutions without worrying about the overlap. If it's really the case that both ***if*** $B_1 \rightarrow S_1$ ***fi*** and ***if*** $B_2 \rightarrow S_2$ ***fi*** are partial solutions separately, then joining them as ***if*** $B_1 \rightarrow S_1 \,\square\, B_2 \rightarrow S_2$ ***fi*** gives solution too.
  - If $B_1 \wedge B_2$ causes (say) $S_1$ to not work correctly, then ***if*** $B_1 \rightarrow S_1$ ***fi*** wasn't really a correct partial solution.
- Note with nondeterministic ***if/do***, the order of the guarded commands makes no difference: Writing the $B_1$ case before the $B_2$ case has exactly the same semantics as writing the $B_2$ case before the $B_1$ case.

- ***Example 7:*** Let's take the *max* program yet again.
  - Both programs below are correct:

    Let $S_1 \equiv$ ***if*** $x \geq y \rightarrow max := x \ \square \ x \leq y \rightarrow max := y$ ***fi*** and

    Let $S_2 \equiv$ ***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

    Let $S_1{*} \equiv$ ***if*** $x \geq y \rightarrow max := x \ \square \ x < y \rightarrow max := y$ ***fi*** and

    Let $S_2{*} \equiv$ ***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

    Let $S_1{}' \equiv$ ***if*** $x \geq y$ ***then*** $max := x$ ***else*** $max := y$ ***fi*** and

    Let $S_2{}' \equiv$ ***if*** $x \leq y$ ***then*** $max := y$ ***else*** $max := x$ ***fi***
- ***Example 9:*** Another example of introducing asymmetry is how
  - We can turn ***if*** $x \geq 0 \rightarrow y := sqrt(x) \ \square \ x \leq 0 \rightarrow y := 0$ ***fi*** into ***if*** $x \geq 0$ ***then*** $y := sqrt(x)$ ***else*** $y := 0$ ***fi***
  - We can turn ***if*** $x \leq 0 \rightarrow y := 0 \ \square \ x \geq 0 \rightarrow y := sqrt(x)$ ***fi*** into ***if*** $x \leq 0$ ***then*** $y := 0$ ***else*** $y := sqrt(x)$ ***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:
    - We can't just test $b0[k0] < b1[k1]$, we must test $k0 < n \wedge k1 < n \wedge b0[k0] < b1[k1]$ (and assume that "$\wedge$" is short-circuiting).
  - Using sentinels is an alternative: Assume $k0[n] = k1[n] = k2[n] = +\infty$ (positive infinity). This lets us write tests like $b0[k0] < b1[k1]$ without having to test for *k0 or k1* equalling *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.
- But say we don't have a match. E.g., say $b0[k0] < b1[k1]$. To make progress, we need to change one (or both?) of the indexes *k0* and *k1*.
  - Since $b1[k1] \leq b1[k1+1]$, it doesn't make sense to increment *k1*.
  - We know $b0[k0] \leq b0[k0+1]$, but we don't know the relationship between $b0[k0+1]$ and $b1[k1]$, so incrementing *k0* might make $b0[k0] = b1[k1]$, which could be part of the answer. So one partial solution to the problem is:

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

- Symmetrically, two other partial solutions are

```
if b1[k1] < b2[k2] → k1 := k1+1 fi
if b2[k2] < b0[k0] → k2 := k2+1 fi
```

- We can join the three partial solutions together to get

***Example 10a:***

```
if  b0[k0] < b1[k1] → k0 := k0+1
□ b1[k1] < b2[k2] → k1 := k1+1
□ b2[k2] < b0[k0] → k2 := k2+1
fi
```

- This code is definitely nondeterministic; e.g., if $b0[k0] < b1[k1] < b2[k2]$ then we'll increment *k0* or *k1* but the choice is nondeterministic.

## *What Cases Remain to be Handled?*

- In Example 10a, the disjunction of the three guards is

$$BB \equiv b0[k0] < b1[k1] \vee b1[k1] < b2[k2] \vee b2[k2] < b0[k0]$$

so $\neg BB \Leftrightarrow b0[k0] \geq b1[k1] \wedge b1[k1] \geq b2[k2] \wedge b2[k2] \geq b0[k0]$ describes the set of states that the program doesn't handle: The semantics of nondeterministic ***if-fi*** tell us that starting the program in a $\sigma \vDash \neg BB$ causes a runtime error. So we should figure out when that happens and how to fix it. For example 10a, a little manipulation tells us

$$\neg BB \Leftrightarrow b0[k0] \geq b1[k1] \geq b2[k2] \geq b0[k0]$$
$$\Leftrightarrow b0[k0] = b1[k1] = b2[k2] = b0[k0] .$$

- But that state is the one that tells us that our search has completed successfully; it's not an error case. So all our program has to do is repeat the ***if … fi*** until all three conditions are false. The easiest way to do that is to rewrite the ***if … fi*** as a ***do … od***:

***Example 10b:***

```
do b0[k0] < b1[k1] → k0 := k0+1
□  b1[k1] < b2[k2] → k1 := k1+1
□  b2[k2] < b0[k0] → k2 := k2+1
od
```

- This is a nondeterministic solution to our problem; to actually run it, we need a deterministic version. Here's one:

***Example 10c:***

```
while  b0[k0] ≠ b1[k1] ∨ b1[k1] ≠ b2[k2])
do          if  b0[k0] < b1[k1]  then k0 := k0+1
      else if  b1[k1] < b2[k2]  then k1 := k1+1
      else if  b2[k2] < b0[k0]  then k2 := k2+1 fi fi fi
od
```

- ***Other approaches:*** This certainly isn't the only way to analyze the cases. For example, if we test *b0[k0]* vs *b1[k1]* for <, =, or >, then we can identify the > case and handle it using and get

```
        if b0[k0] > b1[k1] → k1 := k1+1 fi.
```

Continuing with the the *b1[k1]* vs *b2[k2]* and *b2[k2]* vs *b0[k0]* tests gives us an overall ***if … fi*** with 6 cases:

***Example 10d:***

```
if  b0[k0] < b1[k1] → k0 := k0+1
□ b0[k0] > b1[k1] → k1 := k1+1
□ b1[k1] < b2[k2] → k1 := k1+1
□ b1[k1] > b2[k2] → k2 := k2+1
□ b2[k2] < b0[k0] → k2 := k2+1
□ b2[k2] > b0[k0] → k0 := k0+1
fi
```

- An easy optimization to make is to combine the cases according to which indexes they increment:

***Example 10e:***

```
if  b0[k0] < b1[k1] ∨ b0[k0] < b1[k2] → k0 := k0+1
□ b1[k1] < b0[k0] ∨ b1[k1] < b2[k2] → k1 := k1+1
□ b2[k2] < b0[k0] ∨ b2[k2] < b1[k1] → k2 := k2+1
fi
```

- We can get the code in Example 10b looking for collections of disjuncts that make the remaining disjuncts redundant.  We already have seen *b0[k0] < b1[k1]* ∨ *b1[k1] < b2[k2]* ∨ *b2[k2] < b0[k0]*. Another collection is *b0[k0] < b1[k2]* ∨ *b2[k2] < b1[k1]* ∨ *b1[k1] < b0[k0]*.