Sequential Nondeterminism

CS 536: Science of Programming, Fall 2021

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 avoid making decisions we don’t have to make.
• Example 1: A very simple example is a statement that sets \( \text{max} \) 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:
  \begin{align*}
  & \text{if } x \geq y \text{ then } \text{max} := x \text{ else } \text{max} := y \text{ fi} \\
  & \text{if } y \geq x \text{ then } \text{max} := y \text{ else } \text{max} := x \text{ fi}
  \end{align*}
• The difference is when \( x = y \), the first statement sets \( \text{max} := x \); the second sets \( \text{max} := y \). It doesn’t matter which one of these we choose, we just have to pick one.
• Our standard \textit{if}-\textit{else} statement is \textbf{deterministic}: It can only behave one way. A nondeterministic \textit{if-fi} will specify that one of \( \text{max} := x \) and \( \text{max} := y \) has to be run, but it won’t say how we choose which one.
  \begin{itemize}
    \item 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.
    \item When we make the code more concrete by rewriting it using everyday deterministic code, then we’ll decide which way to write it.
  \end{itemize}

D. Nondeterministic \textit{if-fi}

• \textbf{Syntax}: \( B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n \text{ fi} \)
  \begin{itemize}
    \item The box symbols separate the different arms, like commas in an ordered \( n \)-tuple.
    \item Don’t confuse these right arrows with ones in other contexts (implication operator and single-step execution).
  \end{itemize}
• **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 $B_1, B_2, \ldots, B_n$ 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:**

$$\text{if } x \geq y \rightarrow \text{max} := x \quad \Box \quad y \geq x \rightarrow \text{max} := y \text{ fi}$$

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

  $$\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}$$

  can be written

  $$\text{if } B \rightarrow S_1 \Box \neg B \rightarrow S_2 \text{ fi}.$$ 

**E. Nondeterministic Choices are Unpredictable**

- For us, “nondeterministic” means “unpredictable”.

- Let $\text{flip} = \text{if } T \rightarrow x := 0 \quad \Box \quad T \rightarrow x := 1 \text{ fi}$, 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:** \(\text{do } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n \text{ od}\)

• **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 deterministic test and a nondeterministic if-fi for its body. Let \(BB = (B_1 \lor B_2 \ldots \lor B_n)\) be the disjunction of the guards, then \(\text{do } B_1 \rightarrow S_1 \square \ldots \square B_n \rightarrow S_n \text{ od}\) 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 \(IF = \text{if } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n\) fi and let \(BB = B_1 \lor B_2 \lor \ldots B_n\).

  - 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 \(B_k \rightarrow S_k\) 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 \(DO = \text{do } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n\) od and let \(BB = B_1 \lor B_2 \lor \ldots B_n\).

  - Evaluation of \(DO\) is 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 \(B_k \rightarrow S_k\) 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:**
  - \(\Sigma\) is the set of all states (that proper for whatever we happen to be discussing at that time).
  - \(\Sigma_\perp = \Sigma \cup \{\perp\} = \Sigma \cup \{\perp_d, \perp_e\}\) right now; other versions can be added later.
  - If we just write \(\perp\), then we mean one of \(\perp_d\) or \(\perp_e\); if both are possible, then we should be explicit, as in \(\{\perp_d, \perp_e\}\).

© James Sasaki, 2021
• For a nondeterministic program, to get its denotational semantics, we have to collect all the possible final states (or pseudo-states if we get ⊥): \( M(S, \sigma) = \{ \tau \in \Sigma_\tau \mid \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \}. \)

• 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 \( \tau \in \Sigma_\tau \).

**Example 4**: Let \( S = \text{if } T \rightarrow x := 0 \sqcap T \rightarrow x := 1 \text{ fi} \). 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 \( \sigma \), 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 \( M(\text{skip}, \sigma) = \{ \varnothing \} \) to \( M(\text{skip}, \sigma) = \varnothing \), since it might look like we're claiming that \( M(\text{skip}, \sigma) \) has no final state — it does, the empty state.

• A nondeterministic program can have only one final state.

  **Example 5**: The \( \text{max} \) program from Example 2 has only one final state. Let \( \text{Max} = \text{if } x \geq y \rightarrow \text{max} := x \sqcap y \geq x \rightarrow \text{max} := y \text{ fi} \), in the nondeterministic case, where \( x = y \), both possible execution paths take us to the same state: \( M(\text{Max}, \{x = a, y = a\}) = \{(x = a, y = a, \text{max} = a)\}. \)

• **Note**: To keep from confusing the grader, avoid writing things that look like multisets, such as "\( \{\tau, \tau\} \) where \( \tau = (x = a, y = a, \text{max} = a)\)." For arbitrary \( S \), if \( M(S, \sigma) \) has > 1 member, then \( S \) is nondeterministic. The \( \text{m} \) program shows us that the converse doesn't hold: If \( M(S, \sigma) \) has just 1 member, \( S \) still could be nondeterministic.

• Also, the size of \( M(S, \sigma) \) can vary depending on \( \sigma \).

  **Example 6**: If \( S = \text{if } x \geq 0 \rightarrow x := x \cdot x \sqcap x \leq 8 \rightarrow x := -x \text{ fi} \), then \( M(S, \{x = 0\}) = \{(x = 0)\}. \) However, \( M(S, \{x = 3\}) = \{(x = 9), (x = -3)\} \).

**Difference between \( M(S, \sigma) = \{ \tau \} \) and \( \tau \in M(S, \sigma) \)**

• 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, \( M(S, \sigma) = \{ \bot \} \) says \( S \) always causes an error whereas \( \bot \in M(S, \sigma) \) says that \( S \) might cause an error. Remember that when we write \( \bot \), we're being ambiguous as to whether we mean \( \bot_d \) or \( \bot_e \). If both kinds of failure are possible, we should be explicit: \( M(S, \sigma) = \{ \bot_d, \bot_e \} \) or \( \{ \bot_d, \bot_e \} \subseteq M(S, \sigma) \).

**J. Why Use Nondeterministic programs?**

• Without having defined program correctness yet, it's hard to motivate having nondeterministic programs, so I'll just make some general comments and we'll have to come back to this question at later times.
**Reason 1: 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.

**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, we can approach the problem by asking "When does max := x work?" and "When does max := y work?".
- Since max := x is correct exactly when x ≥ y, the program if x ≥ y → max := x fi is correct.
- Similarly, since max := y is correct exactly when x ≤ y, the program if x ≤ y → 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
  ```

- This program works when x ≥ y or x ≤ y, and since that covers all possibilities, our program is done.

**Reason 2: Nondeterminism Makes it Easy to Handle 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, not worry about there being overlapping cases, and wait to introduce asymmetry when it's unavoidable — i.e., when we rewrite the code deterministically.

**Example 8:** Let's take the Max program yet again.

- Both programs below are correct:
  
  ```
  if x ≥ y → max := x
  □ x ≤ y → max := y
  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 ≥ y → max := x
  □ x < y → max := y
  ```

- Introducing the asymmetry makes the code correspond to the deterministic statements
  
  ```
  if x ≥ y then max := x else max := y
  ```

**Example 9:** Another example of introducing asymmetry is how
• if \( x \geq 0 \) \( \rightarrow y := \sqrt{x} \) \( \square x \leq 0 \rightarrow y := 0 \)
  turns into if \( x \geq 0 \) then \( y := \sqrt{x} \) else \( y := 0 \), while
• if \( x \leq 0 \) \( \rightarrow y := 0 \) \( \square x \geq 0 \rightarrow y := \sqrt{x} \)
  turns into if \( x \leq 0 \) then \( y := 0 \) else \( y := \sqrt{x} \)

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, \( b_0 \), \( b_1 \), and \( b_2 \), all of length \( n \) and all sorted in non-descending order. The goal is to find indexes \( k_0 \), \( k_1 \), and \( k_2 \) such that \( b_0[k_0] = b_1[k_1] = b_2[k_2] \).

But what if no such \( k_0 \), \( k_1 \), and \( k_2 \) exist? An easy solution is to use sentinel values: Let's assume that \( b_0[k_0] \), \( b_1[k_1] \), and \( b_2[k_2] \) all equal \( +\infty \) (i.e., positive infinity). Then tests like \( b_0[k_0] < b_1[k_1] \) behave well even if one or both of \( k_0 \) and \( k_1 \) are “out of range”.

How does the program work? Clearly we want to set \( k_0 = k_1 = k_2 = 0 \) initially, and we have to increment \( k_0 \) or \( k_1 \) or \( k_2 \) until we find a match.

Let's study one pair of indexes, say \( k_0 \) and \( k_1 \). There are three cases:

1. \( b_0[k_0] < b_1[k_1] \). If this happens, we should increment \( k_0 \). Since the arrays are sorted by \( \leq \), incrementing \( k_1 \) can't possibly result in \( b_0[k_0] = b_1[k_1] \), whereas incrementing \( k_0 \) might.
2. \( b_0[k_0] > b_1[k_1] \). Symmetrically, if this happens, we should increment \( k_1 \).
3. \( b_0[k_0] = b_1[k_1] \). If this happens, we don't want to do anything, since we have a possible match. (Of course, we still need \( b_1[k_1] = b_2[k_2] \) or \( b_0[k_0] = b_2[k_2] \) — they're equivalent in this case.)

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

\[
\text{if } b_0[k_0] < b_1[k_1] \rightarrow k_0 := k_0 + 1 \square b_0[k_0] > b_1[k_1] \rightarrow k_1 := k_1 + 1 \text{ fi}
\]

Repeating for the other two pairs of indexes, we get

\[
\text{if } b_1[k_1] < b_2[k_2] \rightarrow k_1 := k_1 + 1 \square b_1[k_1] > b_2[k_2] \rightarrow k_2 := k_2 + 1 \text{ fi}
\]
\[
\text{if } b_2[k_2] < b_0[k_0] \rightarrow k_2 := k_2 + 1 \square b_2[k_2] > b_0[k_0] \rightarrow k_0 := k_0 + 1 \text{ fi}
\]

We can combine these partial solutions into one large if-fi:

\[
\text{if } b_0[k_0] < b_1[k_1] \rightarrow k_0 := k_0 + 1 \square b_0[k_0] > b_1[k_1] \rightarrow k_1 := k_1 + 1 \square b_1[k_1] < b_2[k_2] \rightarrow k_1 := k_1 + 1 \square b_1[k_1] > b_2[k_2] \rightarrow k_2 := k_2 + 1 \square b_2[k_2] < b_0[k_0] \rightarrow k_2 := k_2 + 1 \square b_2[k_2] > b_0[k_0] \rightarrow k_0 := k_0 + 1 \text{ fi}
\]

This large if-fi generates a runtime error if we run it when all the guards are false, namely, when \((b_0[k_0] \geq b_1[k_1] \land b_0[k_0] \leq b_1[k_1] \land \ldots \land b_0[k_0] \leq b_2[k_2])\). Some logical analysis shows that this can only happen when \( b_0[k_0] = b_1[k_1] = b_2[k_2] \). If these three are less than \( +\infty \), then \( k_0, k_1, \) and \( k_2 \) solve our problem; if they are all \( +\infty \), then the problem has no solution.
It's not obvious, but we can halve the number of if cases. First, we'll reorder the cases, separating the < tests from the > tests:

\[
\begin{align*}
\text{if } & b_0[k_0] < b_1[k_1] \rightarrow k_0 := k_0 + 1 \\
\text{and } & b_1[k_1] < b_2[k_2] \rightarrow k_1 := k_1 + 1 \\
\text{and } & b_2[k_2] < b_0[k_0] \rightarrow k_2 := k_2 + 1 \\
\text{if } & b_0[k_0] > b_1[k_1] \rightarrow k_1 := k_1 + 1 \\
\text{and } & b_1[k_1] > b_2[k_2] \rightarrow k_2 := k_2 + 1 \\
\text{and } & b_2[k_2] > b_0[k_0] \rightarrow k_0 := k_0 + 1
\end{align*}
\]

One of the first three cases gets executed if \((b_0[k_0] < b_1[k_1] \lor b_1[k_1] < b_2[k_2] \lor b_2[k_2] < b_0[k_0])\), so none of them get executed if the negation holds. The negation \((b_0[k_0] \geq b_1[k_1] \land b_1[k_1] \geq b_2[k_2] \land b_2[k_2] \geq b_0[k_0])\) is equivalent to \((b_0[k_0] \geq b_1[k_1] \geq b_2[k_2] \geq b_0[k_0])\), which simplifies to \(b_0[k_0] = b_1[k_1] = b_2[k_2]\). Another way to say this is that one of the first three cases can execute unless all three of \(b_0[k_0], b_1[k_1],\) and \(b_2[k_2]\) are equal. This in turn implies that the last three cases are redundant. Dropping them yields

\[
\begin{align*}
\text{if } & b_0[k_0] < b_1[k_1] \rightarrow k_0 := k_0 + 1 \\
\text{and } & b_1[k_1] < b_2[k_2] \rightarrow k_1 := k_1 + 1 \\
\text{and } & b_2[k_2] < b_0[k_0] \rightarrow k_2 := k_2 + 1 \\
\text{fi}
\end{align*}
\]

(Symmetrically, we can argue that the last three cases make the first three redundant, and we get an if-fi that flips all the < tests to be > tests.

These if-fi statements carry out one iteration of a loop we can run to do the entire search. Simply replace the if and fi with do and od:

\[
\begin{align*}
\text{do } & b_0[k_0] < b_1[k_1] \rightarrow k_0 := k_0 + 1 \\
\text{and } & b_1[k_1] < b_2[k_2] \rightarrow k_1 := k_1 + 1 \\
\text{and } & b_2[k_2] < b_0[k_0] \rightarrow k_2 := k_2 + 1 \\
\text{od}
\end{align*}
\]

Note we got this program by asking how the three values might be not all equal and for each possibility, we figured out some code to get us closer to a solution. Another approach is to focus on what actions we might take as the program runs and figure out what guard makes that statement safe. For example, \(k_0 := k_0 + 1\) can be used when \(b_0[k_0] < b_1[k_1]\) or \(b_0[k_0] < b_2[k_2]\). This approach might result in an if-fi with three arms: if \(b_0[k_0] < b_1[k_1] \lor b_0[k_0] < b_2[k_2] \rightarrow k_0 := k_0 + 1\) \[ ... fi,\] but that code can also be optimized into the result above.

One note: This example illustrates a down side of allowing overlapping decisions — when you make the code deterministic, you want to find out which decisions overlap so that you don't include unnecessary code.