Sequential Nondeterminism, pt 2

CS 536: Science of Programming, Fall 2019

9/16, 9/19 pp.5-8

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 lecture you should know

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

C. Nondeterministic Loop

- In part 1, we covered nondeterministic conditional statements. Now we'll look at loops with nondeterministic choices. Happily, it turns out they're very similar to nondeterministic conditionals.
- **Syntax**: **do** $B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n$ **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, nondeterministically select one of the corresponding statements and execute it. Then jump to the top of the loop.
- Relationship between nondeterministic **if-fi** and **do-od**: Our nondeterministic do loop is equivalent to a regular **while** loop with a nondeterministic if body.
  - **do** $B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n$ **od** behaves like
  - **while** BB **do if** $B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n$ **fi od** where $BB \equiv (B_1 \vee B_2 \ldots \vee B_n)$

D. Operational Semantics of Nondeterministic if-fi

- Let $IF \equiv$ **if** $B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n$ **fi**.
- For the semantics of IF in state σ, first let $BB \equiv B_1 \vee B_2 \vee \ldots B_n$ (the disjunction of the guards).
- If evaluation of any of the guards causes an error, then IF causes an error
  - If $\sigma(BB) = \perp_e$ (equivalently, if $\sigma \not\models BB$ and $\sigma \not\models \neg BB$) then $\langle IF, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$.
- If none of the guards are satisfied, then IF causes an error.
  - If $\sigma \models \neg BB$ (equivalently, if $\sigma(\neg BB) = \mathrm{T}$) then $\langle IF, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$.

 © James Sasaki, 2019

- If one or more guards are satisfied, then one of them is chosen nondeterministically and we jump to the command it guards.
  - If $\sigma \vDash BB$, then let $G = \{i \in \{1, \ldots, n\} \mid \sigma \vDash B_i\}$, the set of the indexes of the satisfied guards.
  - Since $\sigma \vDash BB$, we know $G \neq \varnothing$. Then $\langle IF, \sigma\rangle \rightarrow \langle S_j, \sigma\rangle$ for some $j \in G$ (where how we choose j is unspecified).

E. 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 $BB \equiv B_1 \vee B_2 \vee \ldots B_n$ (the disjunction of the guards). Then DO behaves like **while** BB **do if** $B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n$ **fi od**.
- If evaluation of BB fails, then we fail.
  - If $\sigma(BB) = \bot_e$ (i.e., $\sigma \nvDash BB$ and $\sigma \nvDash \neg BB$) then $\langle DO, \sigma\rangle \rightarrow \langle E, \bot_e\rangle$.
- If all of the guards are false, then the loop halts.
  - If $\sigma(BB) = \text{F}$ (i.e., $\sigma \vDash \neg BB$), then $\langle DO, \sigma\rangle \rightarrow \langle E, \sigma\rangle$.
- If at least one guard is true, we nondeterministically choose one of them and jump to the command it guards. We will execute it and then jump back to the top of the loop.
  - If $\sigma(BB) = \text{T}$ (i.e., $\sigma \vDash BB$), then let $G = \{i \in \{1, \ldots, n\} \mid \sigma \vDash B_i\}$, let $j \in G$ (chosen in some unspecified way) and then $\langle DO, \sigma\rangle \rightarrow \langle S_j$ **;** $DO, \sigma\rangle$.

F. Denotational Semantics of Nondeterministic Programs

- Recall we've defined $M(S, \sigma) = \{\tau\}$ if $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$, where σ is the starting memory state and τ is the ending memory state or $\bot$. With nondeterministic programs, τ might not be unique.
- **Example 5**: Let's reuse the program from Example 4: Let $S \equiv$ **if** $\text{T} \rightarrow$ **x :=** $0 \square \text{T} \rightarrow$ **x :=** 1 **fi**. Then $\langle S, \varnothing\rangle \rightarrow^* \langle E, \{\text{x} = 0\}\rangle$ and $\langle S, \varnothing\rangle \rightarrow^* \langle E, \{\text{x} = 1\}\rangle$ are both possible.
  - Since $M(S, \sigma)$ is supposed to hold all possible final states, we have $M(S, \sigma) = \{\{\text{x} = 0\}, \{\text{x} = 1\}\}$. (Don't write this as $\{\{\text{x} = 0, \text{x} = 1\}$ or $\{\text{x} = 0, \text{x} = 1\}$, since these involve improper states.
  - For any single execution of a nondeterministic program, we'll get only one final state: Here, we'll terminate in *one* of $\{\text{x} = 0\}$ or $\{\text{x} = 1\}$, not both of them.
- **Notation**:
  - σ is the set of all states (that proper for whatever we happen to be discussing at that time).
  - $\sigma_\bot = \sigma \cup \{\text{all flavors of } \bot\} = \sigma \cup \{\bot_d, \bot_e\}$ right now; other versions can be added later.
  - Again, for convenience, most times we can write $M(S, \sigma) = \tau$ as a shorthand for $M(S, \sigma) = \{\tau\}$, but there's one ambiguous case: we shouldn't write $M($**skip**$, \varnothing) = \varnothing$ as shorthand for $M($**skip**$, \varnothing) = \{\varnothing\}$.

 © James Sasaki, 2019

- **Definition**: (A restatement) $M(S, \sigma) = \{\sigma \in \sigma_\perp \mid \langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle\}$, the set of all possible final states for S in σ, possibly including $\perp$. Note $M(S, \sigma) \neq \varnothing$, because either S terminates in an actual memory state $\in \sigma$, or it yields $\perp$.
- Even with a nondeterministic program, it's possible to have only one final state.
  - **Example 6**: The `max` program from Example 1 only has one final state. If $S \equiv$ **`if`** `x` $\geq$ `y` $\rightarrow$ `max` := `x` $\square$ `y` $\geq$ `x` $\rightarrow$ `max` := `y` **`fi`**, in the nondeterministic case where `x` = `y`, it doesn't matter which guarded command we execute because both set `max` to the same value: $M(S, \{$`x` = α, `y` = $\alpha\}) = \{\{$`x` = α, `y` = α, `max` = $\alpha\}\}$. Note: We're using sets, not multisets, so don't write the same state twice.
- So if S is deterministic, $M(S, \sigma)$ has 1 member, and if for some σ, $M(S, \sigma)$ has > 1 member, then S is nondeterministic. But $M(S, \sigma)$ having only 1 member doesn't imply that S is deterministic.
- It's also possible to have programs that sometimes have a unique final state and sometimes don't.
  - **Example 7**: 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 $M(S, \sigma) = \{\tau\}$ and $\tau \in M(S, \sigma)$

- $M(S, \sigma) = \{\tau\}$ says that τ is the only possible final result
- $\tau \in M(S, \sigma)$ says that τ is a possible final result of S in σ. If $M(S, \sigma)$ has other members, then those are also possible results of S in σ.
- In particular, $M(S, \sigma) = \{\perp\}$ says S always causes an error; $\perp \in M(S, \sigma)$ says that S might cause an error.
- If we just write $\perp$, then we probably mean one of $\perp_d$ or $\perp_e$; we're either being lazy or vague and leaving off the subscript. If we mean both errors, we should probably write them out: $M(S, \sigma) = \{\perp_d, \perp_e\}$, e.g.

G. 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.

Nondeterminism Introduces Less Asymmetry in Tests

- With an n-way test (i.e., n-1 nested **`if-else-if`** statements), sometimes we realize that a different ordering of the tests makes for simpler tests. Since regular **`if-else`** is asymmetric, it's hard to figure out when branches of nested **`if-else-if`** code can be reordered.
- In nondeterministic **`if`**/**`do`**, the order of the guarded commands makes no difference, so we can shuffle them as we like; sometimes this makes similar or overlapping conditions more obvious.

 © James Sasaki, 2019

Nondeterminism Makes It Easy to Combine Partial Solutions

- It's often easier to solve a problem by solving parts of it and then combining the solutions.
- It's easy to combine solutions if they both are nondeterministic **`if`** statements.
- **Example 8**: Here's one way we might develop the `max` program. (I know, I know, it's much more detailed than what you would do in practice, I'm just using it as an illustration.)
  - We might start by saying, well, I know the `max` of `x` and `y` is either `x` or `y`
  - When is it `x`? Ah, `max` should be `x` if `x` ≥ `y`. I can write that as **`if`** `x` ≥ `y` → `max` := `x` **`fi`**.
    - This is only a partial solution, (it works if `x` ≥ `y`; if `x` < `y`, I get an error) so I'm not done yet.
  - When is `max` = `y`? When `y` ≥ `x`. I can write that as **`if`** `y` ≥ `x` → `max` := `y` **`fi`**.
    - Again, this is only a partial solution: works if `y` ≥ `x`, fails if `y` < `x`.
  - But I can merge the two solutions and get one that works when (`x` ≥ `y` ∨ `y` ≥ `x`) is true

    `if` `x` ≥ `y` → `max` := `x` □ `y` ≥ `x` → `max` := `y` **`fi`**
  - Since the disjunction of the guards (`x` ≥ `y` ∨ `y` ≥ `x`) covers all possible situations, there are no other cases to consider, so we're done!
  - Also note that it doesn't matter if we thought of the `max` := `y` case before the `max` := `x` case; we'd end up with the same program but with the guarded commands swapped:

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

Nondeterminism Lets Us Put off Handling Overlapping Situations

- In Example 8, we might have started with
  - Well, I know the `max` of `x` and `y` is either `x` or `y`
  - Hey, if `x` = `y` then `max` = `x` = `y` . I can write that is as **`if`** `x` = `y` → `max` := `x` **`fi`**.
  - Actually, I could write it as **`if`** `x` = `y` → `max` := `y` **`fi`**, so this works too:

    `if` `x` = `y` → `max` := `x` □ `x` = `y` → `max` := `y` **`fi`**.
  - But the second `x` = `y` doesn't cover any more cases tests than the first `x` = `y` test, so let's just go with

    `if` `x` = `y` → `max` := `x` **`fi`**
- Then if we continue as in Example 8, we end up with a program with three guards:

  `if` `x` = `y` → `max` := `x` □ `x` ≥ `y` → `max` := `x` □ `y` ≥ `x` → `max` := `y` **`fi`**
- At some point, we'll realize we have some overlapping solutions.
- We might decide to let the `x` = `y` case be swallowed up by another case; since `x` = `y` → `x` ≥ `y`, the `x` = `y` case is redundant. We get

  `if` `x` ≥ `y` → `max` := `x` □ `y` ≥ `x` → `max` := `y` **`fi`**
- Or we might realize that `x` = `y` → `y` ≥ `x`, let `y` ≥ `x` subsume `x` = `y`, and get the same program.
- Or we might decide to *subtract* the `x` = `y` case from the other cases to remove the overlaps

 © James Sasaki, 2019

if x = y → max := x □ x > y → max := x □ y > x → max := y **fi**

- Or with the two-test program (x ≥ y or y ≥ x), we might subtract the x ≥ y case from the y ≥ x case and remove that overlap:

  if x ≥ y → max := x □ y > x → max := y **fi**

- This last program doesn't require a nondeterministic choice, so it can be rewritten using deterministic **if**:

  if x ≥ y **then** max := x **else** max := y **fi**

- Going back to the x =, <, or > y program, when you see that two of the guarded command bodies are the same, you can just combine their guards:

  if x = y → max := x □ x > y → max := x □ y > x → max := y **fi**

  becomes

  if x = y ∨ x > y → max := x □ y > x → max := y **fi**

  which of course gets optimized to

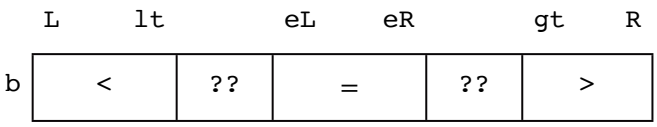
  if x ≥ y → max := x □ y > x → max := y **fi**

  and once again we rewrite this deterministically:

  if x ≥ y **then** max := x **else** max := y **fi**

H. Quicksort Partitioning Example [added 9/19]

- As a more detailed example of how to use nondeterminism, we'll look at the partitioning problem that's part of quicksort.
- Recall that in quicksort, we take an array segment and **partition** it: Reorder its values into a block of all values < a "pivot" value, then the pivot value, and then a block of all values > the pivot. (The < or > blocks might be empty; if both are empty then the array segment is only one element long so is trivially sorted.) After partitioning, quicksort recursively sorts the two < and > blocks.
- The diagram below shows the general case: All the values in the <, =, and > sections are less than, equal to, or greater than the pivot value. The sections marked ?? contain unknown values, so we'll call them the left and right unknown sections.



- **Notation**: `b[L..lt]` < `b[eL..eR]` means all the values in `b[L..lt]` are < all the values in `b[eL..eR]`. Similarly, `b[eL..eR]` < `b[gt..R]` means all the values in `b[gt..R]` are > all the values in `b[eL..eR]`
- The `b[eL..eR]` section will never be empty, but the others might:
  - The < section is empty: Define `LTe` ≡ `lt` = `L-1`
  - The left unknown section is empty: `LUe` ≡ `eL` = `lt+1`

 © James Sasaki, 2019

  - The right unknown is empty: RUe ≡ gt = eR+1
  - The > section is empty: GTe ≡ gt = R+1
- If you're familiar with the problem*, then you may have noticed that this presentation of the problem is much more symmetric than some, where we might have only one unknown area and combine, say, the < and = areas into one ≤ area. Since we're working nondeterministically, we can look at the problem very symmetrically. Once the algorithm is complete, we decide how to break the symmetries as we translate it into a deterministic algorithm.
- Back to the problem: Initially, LTe ∧ GTe holds; when LUe ∧ RUe holds, we're done partitioning. So a very high-level algorithm to solve the problem is

  do ¬LUe → *process a L.U. entry* □ ¬RUe → *process a R.U. entry* **od**
- Here's a diagram detailing the borders around the unknown areas. To process a left unknown entry, we'll look at b[lt+1] to see if it's <, =, or > the pivot value. The goal is to increase the size of the <, =, or > sections by one, at lt+1, eL−1, or gt−1. However, we have to account for the left or right unknown areas being small or empty. This will complicate the > case.

	L	lt	lt+1	lt+2	eL−2	eL−1	eL	eR	eR+1	gt−2	gt−1	gt	R
b	<		?	??		?	=		??		?	>	

b[lt+1] < pivots: If b[lt+1] < the pivots, then since lt+1 is adjacent to the < section, we just increment lt.

if b[lt+1] < b[eL] → lt := lt+1 **fi**

b[lt+1] = pivots: If b[lt+1] = b[eL] then we'll move it to be adjacent to the left end of the = area by swapping b[lt+1] with the slot left of the = area; this will increase the = section. Note this still works if b[lt+1] is already just left of the = area (i.e., if lt+1 = eL−1).

if b[lt+1] = b[eL] → *Swap*(lt+1, eL−1); eL := eL−1 **fi**

b[lt+1] > pivots: If b[lt+1] > b[eL] there are two cases:

eR < gt−1: This is the simple case: The value at gt−1 is a right unknown value, so we can swap b[lt+1] and b[gt−1] (which increases the > section) and move gt left.

if b[lt+1] > b[eL] ∧ eR < gt−1 → *Swap*(lt+1, gt−1); gt := gt−1 **fi**

eR = gt−1 If the value at gt−1 is not unknown, it must be part of the section = the pivots. We can swap b[eL−1] and b[eR] to move the = area leftward and decrement eL and eR to compensate.

if b[lt+1] > b[eL] ∧ eR = gt−1

→ *Swap*(eL−1, eR); eL := eL−1; eR := eR−1 **fi**

* And you should be: It's part of any Data Structures and Algorithms course.

 © James Sasaki, 2019

There is a subtlety here: If the left unknown area consisted of just one value, then it was the one at `eL-1` (i.e., `eL-1` = `lt+1`). In that case, we've swapped the > value into its correct position. On the other hand, if the left unknown area had more than one value, then we've just moved an unknown to `gt-1` and formed a new right unknown area. We could add code to check for these two cases, but they can be detected and handled at the next iteration, so for simplicity, I'll omit them.

- The code for handling the right unknown value at `gt-1` is symmetric to the left unknown code. Combining all the cases and substituting the code for detecting whether the left/right unknown areas are nonempty gives

  do `eL` > `lt+1` → *process a L.U. entry* □ `eR+1` < `gt` → *process a R.U. entry* **od**

  - *Process a L.U. entry* ≡

    if `b[lt+1]` < `b[eL]` → `lt` := `lt+1` **fi**

    □ `b[lt+1]` = `b[eL]` → *Swap*(`lt+1`, `eL-1`); `eL` := `eL-1` **fi**

    □ `b[lt+1]` > `b[eL]` ∧ `eR` < `gt-1` → *Swap*(`lt+1`, `gt-1`); `gt` := `gt-1` **fi**

    □ `b[lt+1]` > `b[eL]` ∧ `eR` = `gt-1`

    → *Swap*(`eL-1`, `eR`); `eL` := `eL-1`; `eR` := `eR-1`

    fi

  - *Process a R.U. entry* ≡

    if `b[gt-1]` > `b[eR]` → `gt` := `gt-1` **fi**

    □ `b[gt-1]` = `b[eR]` → *Swap*(`gt-1`, `eR+1`); `eR` := `eR+1` **fi**

    □ `b[gt-1]` < `b[eR]` ∧ `eL` > `lt+1` → *Swap*(`gt-1`, `lt+1`); `lt` := `lt+1` **fi**

    □ `b[gt-1]` < `b[eR]` ∧ `eL` = `lt+1`

    → *Swap*(`eR+1`, `eL`); `eL` := `eL+1`; `eR` := `eR+1`

    fi

 © James Sasaki, 2019

Nondeterministic Sequential Programs

CS 536: Science of Programming, Fall 2019

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 activity assignment you should

- Be able to evaluate a nondeterministic **`if-fi`** and **`do-od`**.

C. Questions

1. 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 $BB \equiv B_1 \vee B_2 \vee \ldots B_n$. What property does BB have to have for us to avoid an infinite loop when executing DO?
2. Consider the loop `i := 0`; **do** `i < 1000` $\rightarrow S_1$`; i := i+1` $\square$ `i < 1000` $\rightarrow S_2$`; i := i+1` **od** (where neither S_1 nor S_2 modifies `i`). Do we know anything about how many times or in what pattern we will execute S_1 vs S_2?
3. Consider the loop `x := 1; do x ≥ 1 → x := x+1 □ x ≥ 2 → x := x−2 od`. Can running it lead to an infinite loop?
4. What are the three reasons mentioned for why using nondeterminism might be helpful?

5. What is $M(S, \{\texttt{x} = 1\})$ where $S \equiv$ **do** `x ≤ 20 → x := x*2 □ x ≤ 20 → x := x*3` **od** ? [added 9/19]

 © James Sasaki, 2019

Solution to Activity 7 (Nondeterministic Sequential Programs)

1. The nondeterministic **`do–od`** loop halts if *BB* is false at the top of the loop; an infinite loop occurs when *BB* is always true at the top of the loop.

2. Say S_1 is run m times and S_2 is run n times. We know $0 \le m, n \le 1000$ and $m+n = 1000$, but that's all. At each iteration, the choice is nondeterministic (i.e., unpredictable). The choice does not have to be random (like with a coin flip), and the sequence of choices don't have to follow an pattern or distribution or be fair, etc. We can't even assign a probability to any particular sequence of choices (like "always choose S_1").

3. It's possible that the loop could run forever. There's no guaranteed fairness in nondeterministic choice, so we could increment **x** by 1 many more times than we decrement it by 2.

4. Nondeterminism introduces less asymmetry in tests, it makes combining partial solutions easier, and it lets us put off decisions regarding overlapping situations.

5. {{**x** = 12}, {**x** = 16}, {**x** = 18}, {**x** = 24}, {**x** = 27}}

 © James Sasaki, 2019