Sequential Nondeterministic Programs

CS 536: Science of Programming, Fall 2018

9/16: [Still have to post quicksort discussion]

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 of nondeterministic if-fi and do-od.
• How to find the operational and denotational semantics of nondeterministic if-fi and do-od.

C. Avoiding Unnecessary Design Choices

• 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 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:
  • 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 clauses, 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: In if \( B_1 \rightarrow S_1 \, \square \, B_2 \rightarrow S_2 \, \square \, \ldots \, \square \, B_n \rightarrow S_n \) fi, 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 exactly one of the guard tests \( B_1, B_2, \ldots, B_n \) is true, then execute its corresponding statement.
• If more than one test is true, then nondeterministically select a corresponding statement and execute it.
• If no guard is true, abort with a runtime error.

**Example 2:** The max-setting example can be written using \( \text{if } x \geq y \rightarrow \text{max} := x \ □ y \geq x \rightarrow \text{max} := y \) fi

• If only one of \( x \geq y \) and \( y \geq x \) is true, we execute its corresponding assignment.
• If both are true, then we choose one of the tests and execute its assignment.

In the \( \text{max} \) example, we *really* don’t care which arm is executed because they set \( \text{max} \) to the same value.

• 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 \( z := 1 \).
  This is okay if after the if-fi all we need to ensure is, say, \( z \geq 0 \). (If we needed, e.g., even(\( z \)), then we’d have a bug.)

• Of course, not all code written with nondeterministic if-fi has to behave nondeterministically.

**Example 3:** The statement \( \text{if } B \rightarrow S_1 \ □ \neg B \rightarrow S_2 \) fi behaves like our usual deterministic if-else.

### E. Nondeterministic Choices are Unpredictable

• For us, “nondeterministic” means “unpredictable”.
• Let \( \text{flip}() \) be a function that returns 0 or 1, so the assignment \( x := \text{flip}() \) behaves like a coin flip.
• If \( \text{flip}() \) models a random coin flip with a probability attached to the result, then we can talk about distributions and fairness — after a thousand coin flips, we’d expect roughly the same number of 0’s and 1’s.
• If \( \text{flip}() \) is nondeterministic, then its behavior is completely unpredictable. A thousand coin flips might give us anything: Random results, all 0’s, all 1’s, some pattern, etc.

**Example 4:** Using \( \text{if } T \rightarrow x := 0 \ □ T \rightarrow x := 1 \) fi models an nondeterministic \( x := \text{flip}() \)

**Unpredictability shouldn’t matter for purposes of correctness:** The idea with nondeterministic code is that it makes choices where we don’t care about which outcome is chosen. So, no matter how we later rewrite the code deterministically, the result should still be correct.

• If we need a program with fair random choices, we’ll have to code that in at some point, but before then we can concentrate on getting the right results once the choices are made. (E.g., make sure our code works whether we get heads or tails, and then later worry about how to toss the coin.)
• Other programs (like the \( \text{max} \) program) are written nondeterministically because they make overlapping choices. Here, converting from nondeterministic code to deterministic code is when we’d have to decide (e.g.) in what order to do a list of if-else if tests.

### F. Nondeterministic Loop

• In addition to conditionals with nondeterministic choices, we’ll have loops with nondeterministic choices

**Syntax:** \( \text{do } B_1 \rightarrow S_1 \ □ B_2 \rightarrow S_2 \ □ \ldots □ B_n \rightarrow S_n \ \text{od} \)

**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, 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 $B_1 \rightarrow S_1 \sqcap B_2 \rightarrow S_2 \sqcap \ldots \sqcap B_n \rightarrow S_n$ od behaves like
  
  - while $(B_1 \lor B_2 \ldots \lor B_n)$ do if $B_1 \rightarrow S_1 \sqcap B_2 \rightarrow S_2 \sqcap \ldots \sqcap B_n \rightarrow S_n$ fi od

### G. Semantics of Nondeterministic if-fi and do-od

#### Operational Semantics of Nondeterministic if

- Let $IF \equiv$ if $B_1 \rightarrow S_1 \sqcap B_2 \rightarrow S_2 \sqcap \ldots \sqcap B_n \rightarrow S_n$ fi.
- For the semantics of IF in state $\sigma$, first let $BB \equiv B_1 \lor B_2 \lor \ldots \lor B_n$ (the disjunction of the guards).
- If evaluation of any of the guards causes an error, then IF causes an error
  
  - If $\sigma(BB) = \bot$ (equivalently, if $\sigma \not\models BB$ and $\sigma \not\models \neg BB$) then $(IF, \sigma) \rightarrow (E, \bot)$.
- If none of the guards are satisfied, then IF causes an error.
  
  - If $\sigma \models \neg BB$ (equivalently, if $\sigma(\neg BB) = T$) then $(IF, \sigma) \rightarrow (E, \bot)$.
- If one or more guards are satisfied, then one of them is chosen nondeterministically and we jump to the command it guards.
  
  - If $\sigma \models BB$, then let $G = \{i \in \{1, \ldots, n\} \mid \sigma \models B_i\}$, the set of the indexes of the satisfied guards.
  
  - Since $\sigma \models BB$, we know $G \neq \emptyset$. Then $(IF, \sigma) \rightarrow (S_j, \sigma)$ for some $j \in G$ (where how we choose $j$ is unspecified).

#### Operational Semantics of Nondeterministic do-od

- Let $DO \equiv$ do $B_1 \rightarrow S_1 \sqcap B_2 \rightarrow S_2 \sqcap \ldots \sqcap B_n \rightarrow S_n$ od and let $BB \equiv B_1 \lor B_2 \lor \ldots \lor B_n$ (the disjunction of the guards). Then $DO$ behaves like while $BB$ do if $B_1 \rightarrow S_1 \sqcap B_2 \rightarrow S_2 \sqcap \ldots \sqcap B_n \rightarrow S_n$ fi od.
- If evaluation of $BB$ fails, then we fail.
  
  - If $\sigma(BB) = \bot$ (i.e., $\sigma \not\models BB$ and $\sigma \not\models \neg BB$) then $(DO, \sigma) \rightarrow (E, \bot)$.
- If all of the guards are false, then the loop halts.
  
  - If $\sigma(BB) = F$ (i.e., $\sigma \models \neg BB$), then $(DO, \sigma) \rightarrow (E, \sigma)$.
- 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 $\sigma(BB) = T$ (i.e., $\sigma \models BB$), then let $G = \{i \in \{1, \ldots, n\} \mid \sigma \models B_i\}$, let $j \in G$ (chosen in some unspecified way) and then $(DO, \sigma) \rightarrow (S_j ; DO, \sigma)$.

#### Denotational Semantics of Nondeterministic Programs

- Recall we’ve defined $M(S, \sigma) = \{\tau\}$ if $(S, \sigma) \rightarrow^* (E, \tau)$, where $\sigma$ is the starting memory state and $\tau$ is the ending memory state or $\bot$. With nondeterministic programs, $\tau$ might not be unique.
• **Example 5:** Let’s reuse the program from Example 4: Let \( S \equiv \text{if } T \rightarrow x := 0 \text{ and } T \rightarrow x := 1 \text{ fi.} \) Then \( \langle S, \emptyset \rangle \rightarrow^* \langle E, \{x = 0\} \rangle \) and \( \langle S, \emptyset \rangle \rightarrow^* \langle E, \{x = 1\} \rangle \) are both possible.

• Since \( M(S, \sigma) \) is supposed to hold all possible final states, we have \( M(S, \sigma) = \{\{x = 0\}, \{x = 1\}\}. \)

• (Don’t write this as \( \{\{x = 0, x = 1\}\} \) or \( \{x = 0, x = 1\} \), 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 \( \{x = 0\} \) or \( \{x = 1\} \), not both of them.

• **Notation:**
  - \( \Sigma \) is the set of all states (that proper for whatever we happen to be discussing at that time).
  - \( \Sigma_\perp = \Sigma \cup \{\text{all flavors of } \perp\} \) 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 \( M(\text{skip}, \emptyset) = \emptyset \) as shorthand for \( M(\text{skip}, \emptyset) = \{\emptyset\} \).

• **Definition:** (A restatement) \( M(S, \sigma) = \{\tau \in \Sigma_\perp \mid \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle\} \), the set of all possible final states for \( S \) in \( \sigma \), possibly including \( \perp \). Note \( M(S, \sigma) \neq \emptyset \), 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 \( \text{max} \) program from Example 1 only has one final state. If \( S \equiv \text{if } x \geq y \rightarrow \text{max} := x \text{ and } y \geq x \rightarrow \text{max} := y \text{ fi.} \) in the nondeterministic case where \( x = y \), it doesn’t matter which guarded command we execute because both set \( \text{max} \) to the same value: \( M(S, \{x = \alpha, y = \alpha\}) = \{\{x = \alpha, y = \alpha, z = \alpha\}\} \). 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 \( \sigma \), \( M(S, \sigma) \) 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 \( S \equiv \text{if } x \geq 0 \rightarrow x := x \times x \text{ and } x \leq 8 \rightarrow x := -x \text{ fi.} \) 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 \( \tau \) is the only possible final result.

• \( \tau \in M(S, \sigma) \) says that \( \tau \) is a possible final result of \( S \) in \( \sigma \). If \( M(S, \sigma) \) has other members, then those are also possible results of \( S \) in \( \sigma \).

• 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.

**II. 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.

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
    
    \[
    \text{if } x \geq y \rightarrow \text{max := } x \quad \text{if } y \geq x \rightarrow \text{max := } y
    \]
    
    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:
    
    \[
    \text{if } y \geq x \rightarrow \text{max := } y \quad \text{if } x \geq y \rightarrow \text{max := } x
    \]

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:
      
      \[
      \text{if } x = y \rightarrow \text{max := } x \quad \text{if } x = y \rightarrow \text{max := } y
      \]
    - But the second x = y doesn’t cover any more cases tests than the first x = y test, so let’s just go with
      \[
      \text{if } x = y \rightarrow \text{max := } x
      \]
  - Then if we continue as in Example 8, we end up with a program with three guards:
    
    \[
    \text{if } x = y \rightarrow \text{max := } x \quad \text{if } x \geq y \rightarrow \text{max := } x \quad \text{if } y \geq x \rightarrow \text{max := } y
    \]
  - 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
    \[
    \text{if } x \geq y \rightarrow \text{max := } x \quad \text{if } y \geq x \rightarrow \text{max := } y
    \]
  - 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

\[
\text{if } x = y \rightarrow \text{max} := x \quad \text{if } x > y \rightarrow \text{max} := x \quad \text{if } y > x \rightarrow \text{max} := y \text{ fi}
\]

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

\[
\text{if } x \geq y \rightarrow \text{max} := x \\
\text{if } y > x \rightarrow \text{max} := y \text{ fi}
\]

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

\[
\text{if } x \geq y \text{ then max := } x \text{ else max := } y \text{ fi}
\]

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

\[
\text{if } x = y \rightarrow \text{max} := x \quad \text{if } x > y \rightarrow \text{max} := x \quad \text{if } y > x \rightarrow \text{max} := y \text{ fi}
\]

becomes

\[
\text{if } x = y \lor x > y \rightarrow \text{max} := x \quad \text{if } y > x \rightarrow \text{max} := y \text{ fi}
\]

which of course gets optimized to

\[
\text{if } x \geq y \rightarrow \text{max} := x \quad \text{if } y > x \rightarrow \text{max} := y \text{ fi}
\]

and once again we rewrite this deterministically:

\[
\text{if } x \geq y \text{ then max := } x \text{ else max := } y \text{ fi}
\]
Nondeterministic Sequential Programs
CS 536: Science of Programming, Fall 2018

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 \( IF \equiv \text{if} B_1 \rightarrow S_1 \sqcup B_2 \rightarrow S_2 \sqcup \ldots \sqcup B_n \rightarrow S_n \text{ fi} \) and \( BB \equiv B_1 \lor B_2 \lor \ldots \lor B_n \).
   a. What property does \( BB \) have to have for us to avoid a runtime error when executing \( IF \)?
   b. Does it matter if we reorder the guarded commands? (E.g., if we swap \( B_1 \rightarrow S_1 \) and \( B_2 \rightarrow S_2 \).)

2. Let \( T_1 \equiv \text{if} B_1 \rightarrow S_1 \sqcup B_2 \rightarrow S_2 \text{ fi} \) and \( T_2 \equiv \text{if} B_1 \text{ then } S_1 \text{ else if } B_2 \text{ then } S_2 \text{ fi fi} \).
   a. Fill in the table below to describe what happens for each combination of \( B_1 \) and \( B_2 \) being true or false.
   b. For what kinds of states \( \sigma \) can these two statements behave differently?

<table>
<thead>
<tr>
<th>If ( \sigma \models \ldots )</th>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_1 \land B_2 )</td>
<td>\text{Executes } S_1 \text{ or } S_2</td>
<td></td>
</tr>
<tr>
<td>( B_1 \land \neg B_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \neg B_1 \land B_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \neg B_1 \land \neg B_2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Let \( DO \equiv \text{do} B_1 \rightarrow S_1 \sqcup B_2 \rightarrow S_2 \sqcup \ldots \sqcup B_n \rightarrow S_n \text{ od} \) and \( BB \equiv B_1 \lor B_2 \lor \ldots \lor B_n \). What property does \( BB \) have to have for us to avoid an infinite loop when executing \( DO \)?

4. Consider the loop \( i := 0; \text{ do } i < 1000 \rightarrow S_1; i := i + 1 \sqcup i < 1000 \rightarrow S_2; i := i - 1 \text{ od} \) (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 \( S_2 \)?

5. Consider the loop \( x := 1; \text{ do } x \geq 1 \rightarrow x := x + 1 \sqcup x \geq 2 \rightarrow x := x - 2 \text{ od} \). Can running it lead to an infinite loop?

6. What are the three reasons mentioned for why using nondeterminism might be helpful?
Solution to Activity 7 (Nondeterministic Sequential Programs)

1. (Basic properties of nondeterministic if)
   a. We need $\sigma \models BB$, because if $\sigma \models \neg BB$, then $M(\text{IF }, \sigma) = \{ \bot \}$. (In English: At least one guard must be true; if none of them are true, we get a runtime error.)
   b. The order of the guarded commands doesn’t matter: If more than one guard is true, we nondeterministically choose one element from the set of corresponding statements, and in a set, the elements aren’t ordered.

2. (Deterministic vs nondeterministic conditionals) Recall $T_1 \equiv \text{if } B_1 \rightarrow S_1 \text{ else if } B_2 \rightarrow S_2 \text{ fi}$ and $T_2 \equiv \text{if } B_1 \text{ then } S_1 \text{ else if } B_2 \text{ then } S_2 \text{ fi}$.
   a. Execution of $T_1$ and $T_2$:
   
<table>
<thead>
<tr>
<th>If $\sigma \models \ldots$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1 \land B_2$</td>
<td>Executes $S_1$ or $S_2$</td>
<td>Executes $S_1$</td>
</tr>
<tr>
<td>$B_1 \land \neg B_2$</td>
<td>Executes $S_1$</td>
<td>Executes $S_1$</td>
</tr>
<tr>
<td>$\neg B_1 \land B_2$</td>
<td>Executes $S_2$</td>
<td>Executes $S_2$</td>
</tr>
<tr>
<td>$\neg B_1 \land \neg B_2$</td>
<td>Produces runtime error</td>
<td>Executes skip</td>
</tr>
</tbody>
</table>

   b. $T_1$ and $T_2$ behave the same when one of $B_1$ and $B_2$ is true and the other is false. When both are true, $T_2$ always executes $S_1$ but $T_1$ will execute $S_1$ or $S_2$. When both of $B_1$ and $B_2$ are false, $T_1$ yields a runtime error but $T_2$ does nothing.

3. The nondeterministic $\text{do-od}$ 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.

4. Say $S_1$ is run $m$ times and $S_2$ is run $n$ times. We know $0 \leq m, n \leq 1000$ 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 $S_1$”).

5. 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.

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