Sequential Nondeterminism

Stefan Muller, based on material by Jim Sasaki

CS 536: Science of Programming, Fall 2023
Lecture 21

1 What and Why?

1.1 What?

So far, the programs we have been dealing with have been *deterministic*, meaning that they behave in exactly one way at runtime, although we may not know which way until we run the program. For instance, consider the following program:

$$\mathsf{if}\ (x < \overline{0})\ \mathsf{then}\ z := \overline{0}\ \mathsf{else}\ z := \overline{1}\ \mathsf{fi}$$

We don't know the value of x before executing the program, i.e., when writing, analyzing, annotating, compiling, etc. the program. However, at runtime, x has a definite value, and therefore, the program will have a definite behavior. This is why, until now, $M(s, \sigma)$ has been a set containing exactly one state.

Nondeterminism occurs when a program has different and unpredictable behaviors at runtime. A major source of nondeterminism in programs is *concurrency*, where multiple threads run simultaneously, and the computer schedules them non-deterministically. We'll study this in the coming lectures. In this lecture, we'll look at nondeterminism in *sequential* (not parallel or concurrent) programs.

1.2 Why?

Sequential programs generally aren't *actually* nondeterministic, which is why we haven't discussed nondeterminism until now. However, non-determinism is useful for modeling anything that we can't reason about when we're writing, analyzing, or compiling a program such as randomness[1], user input, or undefined behavior. It is also useful when writing a program when we don't want to have to worry about overlapping cases. For instance, consider two programs that calculate the max of x and y:

$$\mathsf{if}\ (x \geq y)\ \mathsf{then}\ m := x\ \mathsf{else}\ m := y\ \mathsf{fi}$$

$$\mathsf{if}\ (y \geq x)\ \mathsf{then}\ m := y\ \mathsf{else}\ m := x\ \mathsf{fi}$$

These two programs are essentially the same, but they differ when $x = y$: in this case, the first program will assign m to x and the second will assign m to y. Although this is not true nondeterminism, as the result is the same in either case, it is possible that we might prefer one choice over the other for efficiency or other reasons, and we might want to delay that choice or let the compiler choose the most efficient version. A nondeterministic version of this if statement would let us choose nondeterministically between $m := x$ and $m := y$ when x is equal to y.

2 Syntax and Informal Semantics

In order to introduce nondeterminism into our language, we are extending its semantics by adding a nondeterministic branching statement.

[1] You may think that randomness should be the very definition of nondeterministic behavior, but most randomness used in programming (e.g., pseudorandom number generators or PRNGs) is, in fact, deterministic if one considers as inputs to the program anything from the environment that might be used to seed a PRNG, like the time or `/dev/rand`.

2.1 Nondeterministic Branch (if-fi)

The extended grammar of the statements with nondeterministic branching or conditional statements is as follows:

$$S ::= \ldots \mid \mathsf{if}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{fi}$$

The branching statement consists of multiple statements $s_1, \ldots, s_n$, each *guarded* by a Boolean expression $e_1, \ldots, e_n$ [2]. Each e_i tells us whether it's OK to run s_i (if e_i is true, it's OK). This is like a `switch` statement in C/Java or a pattern match in some higher-level languages, but with a key difference: it's nondeterministic, while most of those language constructs are deterministic.

The informal semantics of the branching statement can be summarized as:

- If none of the e_i are true, abort with a runtime error.
- Otherwise, if multiple guards are true, run one of the corresponding statements nondeterministically (in the special case that exactly one is true, that one will be run deterministically).

Example 1. We can write our max program from above as

$$\mathsf{if}\ x \geq y \to m := x \,\square\, y \geq x \to m := y\ \mathsf{fi}$$

It has two branches, guarded with the boolean expressions $x \geq y$ and $y \geq x$, respectively. In particular, if $x = y$, either branch can run. In this case, since both set m to the same value $x = y$, we don't care which. However, in general, the branches might do different things.

Example 2. The program

$$\mathsf{if}\ x\%2 = 0 \to \mathit{result} := x/2 \,\square\, x\%3 = 0 \to \mathit{result} := x/3\ \mathsf{fi}$$

has two branches guarded by $x\%2 = 0$ and $x\%3 = 0$. When the value of x is divisible by 2, the first branch is taken, and the result is x divided by 2. Similarly, when x is divisible by 3, the second branch is taken, and the result is x divided by 3. If both conditions are true, either of the branches can be taken nondeterministically. or instance, for $x = 2$ and $x = 3$, the result would be 1. However, in the case of $x = 6$, the result is non-deterministic and could be either 2 or 3.

2.2 Nondeterminism vs. Randomness

In this class, we'll treat "nondeterministic" as meaning something closer to "unpredictable" than "random". Consider the following program:

$$\mathit{flip} \triangleq \mathsf{if}\ T \to x := \overline{0} \,\square\, T \to x := \overline{1}\ \mathsf{fi}$$

Both guards always hold, so flip will always set x to 0 or 1 nondeterministically. As the name suggests, this models a coin flip, but it doesn't necessarily *act* like a coin flip. With a real coin flip, you expect a 50-50 chance of getting 0 or 1, but since the *flip* program is nondeterministic, its behavior is completely unpredictable. Running the program *flip* a thousand times might result in any outcome: all 0's, all 1's, some pattern, random 500 heads and 500 tails, etc.

Nondeterminism shouldn't affect correctness. We'll use nondeterminism when we don't want to worry about how choices are made. Maybe when we write the final version of the program, we'll replace *flip* with a call to a random number generator or a prompt for user input. The point is we want to make sure the rest of our program is correct, whether the coin comes up heads or tails.

This also applies to our max example above. If we were actually writing a max function, we'd probably go back and replace the nondeterministic code with deterministic code that makes one of the choices, but it's good to know that the correctness of the program won't change when we make that choice. Or maybe

[2] That $\square$ symbol between the branches isn't a missing font in your PDF reader or printer, it's actually the symbol we'll use. If you do have a missing font in your PDF reader or printer and you're seeing squares in other places, you may start getting very confused, unfortunately.

our language actually has a nondeterministic branching operator and the compiler will actually make the most efficient choice when multiple guards are true: in this case, we may want to leave it nondeterministic, but we definitely want to know that the correctness of the program doesn't depend on the compiler's choice (which may change in a future version of the compiler)!

This also exposes the difference between nondeterminism and randomness: in the *max* function, the compiler will almost certainly choose a branch deterministically (rather than randomly), but this choice is part of the *compiler*, not the *program*, and we don't want to reason about the whole compiler when we're proving our program correct.

Undefined behavior in languages is another good example of this. For example, in C, accessing an out-of-bounds array index is "undefined." The compiler is free to do anything it wants, including return a random number. Now, for any given compiler, what it does is deterministic. Generally, the compiler chooses the branch that allows it to generate the fastest code (probably return whatever's in memory next to the array or segfault). But if we wanted our program to be correct in the face of undefined behavior, we'd treat this as nondeterminism because it's *unpredictable* (to us).

2.3 Havoc

Extending the *flip* example, we might want code that sets x to *anything*, like so:

$$\mathsf{if}\ \cdots \square\, T \to x := \overline{-1} \square\, T \to x := \overline{0} \square\, T \to x := \overline{1} \square \ldots\ \mathsf{fi}$$

Of course, we can't write such a program in a finite amount of space[3]. Instead, we'll explicitly make it a statement:

$$s ::= \ldots \mid \mathsf{havoc}\ x$$

$\mathsf{havoc}\ x$ assigns any integer nondeterministically to x. We can think of it as a call to a random number generator, taking an unpredictable input from the user, or any other sort of unpredictable behavior; the point is it's some unpredictable process we can't or don't want to predict the exact behavior of.

Example 3. The program

$$\mathsf{havoc}\ x; y := x * x$$

assigns a value to x nondeterministically and then assigns the square of x to y.

2.4 Nondeterministic Loop

Finally, we add a nondeterministic while loop to the grammar:

$$s ::= \ldots \mid \mathsf{do}\ e_1 \to s_1 \square \cdots \square\, e_n \to s_n\ \mathsf{od}$$

The informal semantics of the nondeterministic loop is:

- 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.

Example 4. The program $W = \mathsf{do}\ T \to x := x + \overline{1} \square\, T \to x := x - \overline{1}\ \mathsf{od}$ models a nondeterministic (random) walk. The loop continues infinitely, incrementing or decrementing x on each iteration.

3 Small-step semantics

Nondeterminism is fairly simple to express in small-step semantics. We'll just have a rule for each possibility. To step a program, we can apply *any rule* whose premise holds. This means that now, there may be multiple rules with which we can take a step, and this there may be multiple σ' and s' such that $\langle s, \sigma\rangle \to \langle s', \sigma'\rangle$.

[3]Technically, if we only consider, e.g., 64-bit integers, we could, but please don't try.

Nondeterministic branching. The nondeterministic branching statement can step to any statement guarded by a guard that is satisfied by the state. If all guards are false, nondeterministic branching fails. We have the following two rules.

$$\frac{\sigma(e_1) = \cdots = \sigma(e_n) = F}{\langle \mathsf{if}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{fi}, \sigma\rangle \to \langle \mathsf{skip}, \bot\rangle} \qquad \frac{\sigma(e_i) = T \qquad \forall i \in [1,n].\sigma(e_i) \neq \bot}{\langle \mathsf{if}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{fi}, \sigma\rangle \to \langle s_i, \sigma\rangle}$$

Moreover, if any of the branches results in a run-time error, the nondeterministic branch fails.

$$\frac{\exists i \in [1,n].\sigma(e_i) = \bot}{\langle \mathsf{if}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{fi}, \sigma\rangle \to \langle \mathsf{skip}, \bot\rangle}$$

Example 5. Consider the program

$$S_1 \triangleq \mathsf{if}\ x/y > 0 \to \mathit{sign} := 1 \,\square\, x * y \leq 0 \to \mathit{sign} := 0\ \mathsf{fi}.$$

The program assigns 1 to *sign* if both x and y are both positive or both negative. It assigns 0 to *sign* if x is positive and y is negative or vice versa. Let's look at the behavior of the program when y is equal to 0 in the initial state. For example, consider the state $\sigma_1 \triangleq \{x = 1, y = 0\}$, we have

$$\langle S_1, \sigma_1\rangle \to \langle \mathsf{skip}, \bot\rangle$$

Havoc. The rule for havoc just picks a number and updates the state. There's only one rule, but we can choose any value for n, hence the nondeterminism.

$$\frac{n \in \mathbb{Z}}{\langle \mathsf{havoc}\ x, \sigma\rangle \to \langle \mathsf{skip}, \sigma[x \mapsto n]\rangle}$$

Nondeterministic loop. The rules for the nondeterministic loop are pretty similar to the nondeterministic branch (Unfortunately, we can't just use the same trick of stepping a "while" to an "if" because the nondeterministic loop has a different behavior if all the guards are false):

$$\frac{\sigma(e_1) = \cdots = \sigma(e_n) = F}{\langle \mathsf{do}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{od}, \sigma\rangle \to \langle \mathsf{skip}, \sigma\rangle}$$

$$\frac{\sigma(e_i) = T \qquad \forall i \in [1,n].\sigma(e_i) \neq \bot}{\langle \mathsf{do}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{od}, \sigma\rangle \to \langle s_i; \mathsf{do}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{od}, \sigma\rangle}$$

$$\frac{\exists i \in [1,n].\sigma(e_i) = \bot}{\langle \mathsf{do}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{od}, \sigma\rangle \to \langle \mathsf{skip}, \bot\rangle}$$

3.1 More Examples

Example 6. (Flip) There are two ways we could step the *flip* program:

$$\begin{array}{lll} & \langle \mathsf{if}\ T \to x := \overline{0} \,\square\, T \to x := \overline{1}\ \mathsf{fi}, \{\}\rangle & \\ \to & \langle x := \overline{0}, \{\}\rangle & \sigma(T) = T \\ \to & \langle \mathsf{skip}, \{x = 0\}\rangle & \end{array}$$

or:

$$\begin{array}{lll} & \langle \mathsf{if}\ T \to x := \overline{0} \,\square\, T \to x := \overline{1}\ \mathsf{fi}, \{\}\rangle & \\ \to & \langle x := \overline{1}, \{\}\rangle & \sigma(T) = T \\ \to & \langle \mathsf{skip}, \{x = 1\}\rangle & \end{array}$$

Example 7. (Infinite "Random" (Nondeterministic) Walk) Let $W = \mathsf{do}\ T \to x := x + \overline{1} \,\square\, T \to x := x - \overline{1}\ \mathsf{od}$. This loop continues infinitely, incrementing or decrementing x on each iteration.

$$\begin{array}{rl} & \langle W, \{x = 0\}\rangle \\ \to & \langle x := x + \overline{1}; W, \\ \rangle \to^2 & \langle W, \{x = 1\}\rangle \\ \to & \langle x := x + \overline{1}; W, \\ \rangle \to^2 & \langle W, \{x = 2\}\rangle \\ \to & \langle x := x - \overline{1}; W, \\ \rangle \to^2 & \langle W, \{x = 1\}\rangle \\ & \ldots \end{array}$$

4 Big-Step Semantics

Next, we provide the big-step semantics for each of the new nondeterministic statements. Recall that $M(s, \sigma)$ is the set of states, which we'll sometimes denote Σ, that you can get to from running s starting from the initial state σ. For deterministic programs, there is only one such state, so $M(s, \sigma) = \{\sigma'\}$ iff $\langle s, \sigma\rangle \to^* \langle \mathsf{skip}, \sigma'\rangle$. Nondeterministic programs may (or may not) have multiple final states.

$$\begin{array}{rcll} M(\mathsf{if}\ e_1 \to s_1 \,\square \cdots \square\, e_n \to s_n\ \mathsf{fi}, \sigma) & = & \{\bot\} & \sigma(e_i) = \bot \\ M(\mathsf{if}\ e_1 \to s_1 \,\square \cdots \square\, e_n \to s_n\ \mathsf{fi}, \sigma) & = & \{\bot\} & \forall i. \sigma(e_i) = F \\ M(\mathsf{if}\ e_1 \to s_1 \,\square \cdots \square\, e_n \to s_n\ \mathsf{fi}, \sigma) & = & \{\bigcup_{i \in [1,n], \sigma(e_i) = T} M(s_i, \sigma) & \\ M(\mathsf{havoc}\ x, \sigma) & = & \{\sigma[x \mapsto n] \mid n \in \mathbb{Z}\} & \end{array}$$

We won't formally define the big-step semantics for nondeterministic loops, but it involves the same technique we used for regular while loops. For example, for the infinite nondeterministic walk, we have $M(\mathsf{do}\ T \to x := x + \overline{1} \,\square\, T \to x := x - \overline{1}\ \mathsf{od}) = \{\}$, since the program never terminates.

Example 8: Flip.

$$\begin{array}{rl} & M(\mathsf{if}\ T \to x := \overline{0} \,\square\, T \to x := \overline{1}\ \mathsf{fi}, \{\}) \\ = & M(x := \overline{0}, \{\}) \cup M(x := \overline{1}, \{\}) \\ = & \{\{x = 0\}\} \cup \{\{x = 1\}\} \\ = & \{\{x = 0\}, \{x = 1\}\} \end{array}$$

Note: This is a set containing two states. DO NOT EVER write it as {x = 0, x = 1}, which is a single ill-formed state.

Example 9: Nested nondeterminism.

$$\begin{array}{rl} & M(\mathsf{if}\ T \to (\mathsf{if}\ T \to x := \overline{0} \,\square\, T \to x := \overline{2}\ \mathsf{fi}) \,\square\, T \to x := \overline{1}\ \mathsf{fi}, \{\}) \\ = & M((\mathsf{if}\ T \to x := \overline{0} \,\square\, T \to x := \overline{2}\ \mathsf{fi}), \{\}) \cup M(x := \overline{1}, \{\}) \\ = & M(x := 0, \{\}\}) \cup M(x := 2, \{\}) \cup \{\{x = 1\}\} \\ = & \{\{x = 0\}\} \cup \{\{x = 2\}\} \cup \{\{x = 1\}\} \\ = & \{\{x = 0\}, \{x = 1\}, \{x = 2\}\} \end{array}$$

The set of states is a set, so the order doesn't matter.

4.1 Important Observations

- We still have that if $\langle s, \sigma\rangle \to^* \langle \mathsf{skip}, \sigma'\rangle$, then $\sigma' \in M(s, \sigma)$. But now there may be more than one σ' for which that's true.
- If $|M(s, \sigma)| > 1$, then s is nondeterministic.

- The converse isn't true: it's possible for a nondeterministic program to have only one final state:

$$\begin{array}{ll} & M(\mathsf{if}\ x \geq y \to m := x \,\square\, y \geq x \to m := y\ \mathsf{fi}, \{x = 3, y = 3\}) \\ = & M(m := x, \{x = 3, y = 3\}) \cup M(m := y, \{x = 3, y = 3\}) \\ = & \{\{x = 3, y = 3, m = 3\}\} \cup \{\{x = 3, y = 3, m = 3\}\} \\ = & \{\{x = 3, y = 3, m = 3\}\} \end{array}$$

- It's also possible for $|M(s, \sigma)|$ to vary based on σ: Let $s = \mathsf{if}\ x \geq 0 \to x := x * x \,\square\, x \leq 8 \to x := -x\ \mathsf{fi}$. We have:
  - $M(s, \{x = 0\}) = \{\{x = 0\}\}$
  - $M(s, \{x = 3\}) = \{\{x = 9\}, \{x = -3\}\}$
  - $M(s, \{x = 10\}) = \{\{x = 100\}\}$
- There's a difference between saying $M(s, \sigma) = \{\sigma'\}$ and saying $\sigma' \in M(s, \sigma')$. Both say that σ' can be a final state, but $M(s, \sigma) = \{\sigma'\}$ says it is the only final state and $\sigma' \in M(s, \sigma')$ leaves open the possibility that there are others.
- In particular, $M(s, \sigma) = \{\bot\}$ says that s always causes an error while $\bot \in M(s, \sigma)$ says that s *might* cause an error.
- $M(s, \sigma) = \{\}$ says that s always diverges.

5 Hoare Triples, wp, sp (for loop-free nondeterministic programs)

Next, we design a Hoare-logic rule for nondeterministic branching and havoc.

Nondeterministic branching. For a nondeterministic branch, in each branch, we get the precondition that the branch's guard holds. We need all of the branches to guarantee the postcondition (as with if, we could have an alternate rule that lets s_i have postcondition q_i and then the postcondition for the whole nondeterministic branch is $q_1 \vee \cdots \vee q_n$, but this is more unwieldy with more branches.)

$$\frac{\forall i. \vdash \{p \wedge e_i\}\ s_i\ \{q\}}{\vdash \{p\}\ \mathsf{if}\ e_1 \to s_1 \,\square\, \cdots \,\square\, e_n \to s_n\ \mathsf{fi}\ \{q\}}\ (\textsc{NDBranch})$$

Havoc. Like with normal assignments, we have two rules for havoc, which you can see as a form of nondeterministic assignment. The backward rule requires that anything that is true of x in the postcondition needs to be true of *any* integer in the precondition.

$$\frac{}{\vdash \{\forall x_0 \in \mathbb{Z}.[x_0/x]q\}\ \mathsf{havoc}\ x\ \{q\}}\ (\textsc{NDHavoc-Backward})$$

The forward rule replaces x with a new ghost variable x_0, representing the old value of x: this is the first half of what the normal forward assignment rule does. Instead of adding what we now know about x, it stops there, because we now know nothing about x. This is basically equivalent to forgetting anything we knew about x, except things that are true of any integer.

$$\frac{x_0\ \mathsf{fresh}}{\vdash \{p\}\ \mathsf{havoc}\ x\ \{[x_0/x]p\}}\ (\textsc{NDHavoc-Forward})$$

It may seem like havoc x is equivalent to leaving x as an uninitialized variable (like a function argument), which we also know nothing about. In many cases, this is true, but we can also use havoc to forget things about x in the middle of a program, maybe in just one branch:

$$
\begin{array}{ll}
 & \{T\} \\
x := \overline{0}; & \{x = 0\} \\
\mathsf{if}\ (x < 0)\ \{ & \{x = 0 \wedge x < 0\} \\
\quad \mathsf{havoc}\ x & \{x_0 = 0 \wedge x_0 < 0\} \Rightarrow \{F\} \Rightarrow \{x = 0\} \\
\}\ \mathsf{else}\ \{ & \{x = 0 \wedge x \geq 0\} \Rightarrow \{x = 0\} \\
\quad \mathsf{skip} & \{x = 0\} \\
\} & \{x = 0\}
\end{array}
$$

The fact that we can prove a postcondition about x means that the branch with the havoc must never be reached!

5.1 Wlp, wp and sp

The extensions to the algorithm for computing wlp are pretty straightforward.

$$
\begin{array}{rcl}
wlp(\mathsf{if}\ e_1 \to s_1 \,\square \cdots \square\, e_n \to s_n\ \mathsf{fi}, q) & = & (e_1 \to wlp(s_1, q)) \wedge \cdots \wedge (e_n \to wlp(s_n, q)) \\
wlp(\mathsf{havoc}\ x, q) & = & \forall x_0.[x_0/x]q
\end{array}
$$

The extensions to the error-free predicates are also straightforward. Note that in the predicate for nondeterministic branches, we need $(e_1 \vee \cdots \vee e_n)$ to prevent the run-time error that occurs when all guards $e_1, \cdot e_n$ are false.

$$
\begin{array}{rcl}
D(\mathsf{if}\ e_1 \to s_1 \,\square \cdots \square\, e_n \to s_n\ \mathsf{fi}) & = & D(e_1) \wedge \cdots \wedge D(e_n) \wedge (e_1 \vee \cdots \vee e_n) \wedge (e_1 \to D(s_1)) \wedge \ldots (e_n \to D(s_n)) \\
D(\mathsf{havoc}\ x) & = & T
\end{array}
$$

The strongest postcondition says that for a nondeterministic branch, we know one branch was taken but not which. The sp for havoc uses the forward rule.

$$
\begin{array}{rcll}
sp(\mathsf{if}\ e_1 \to s_1\ \mathsf{fi} \,\square \cdots \square\, e_n \to s_n, p) & = & sp(s_1, p \wedge e_1) \vee \cdots \vee sp(s_n, p \wedge e_n) & \\
sp(\mathsf{havoc}\ x, p) & = & [x_0/x]p & x_0\ \textit{fresh}
\end{array}
$$