15-316: Software Foundations of Security and Privacy

Lecture Notes on Proving Safety, Compositionally

Matt Fredrikson

Carnegie Mellon University
Lecture 4 & 5

1 Introduction

In the previous lecture, we introduced a relatively simple language that still has enough complexity to capture some interesting aspects of many imperative languages. This language describes programs that operate on states that map variable symbols to integers, and we defined its semantics in terms of the set of traces that its programs generate when they are run starting in a particular state. We then discussed *safety properties*, which allow us to specify whether a program will ever do something "bad" in the course of its execution. Safety properties are pervasive when considering security and correctness, capturing things like memory safety and access control that are indespensable for ensuring that the programs we write behave as intended without succumbing to the malign influence of an attacker. Finally, we saw how to specify safety properties formally using *dynamic logic*, a formal system that deals with the dynamics of state as programs make changes to variables.

In this lecture, we will use dynamic logic to develop reasoning principles that let us conclude decisively whether a program satisfies a given safety property. Just as we did with the propositional logic, we will start by giving dynamic logic a formal semantics to which these reasoning principles can refer in order to justify soundness. We will then see how to apply this reasoning to programs, writing sequent calculus proofs that incrementally break the program down into smaller and smaller pieces, ultimately proving statements about the safety of a program that apply to all possible inputs and traces it can possibly generate.

2 Review

Recall that in the previous lecture we introduced a programming language that exemplifies the core elements common to many imperative languages.

Definition 1 (Program)**.** *Deterministic while programs* are defined by the following grammar (α, β are programs, x is a variable, e is a term, and Q is a Boolean formula of arithmetic):

$$\alpha, \beta \ ::= \ x := e \mid \texttt{assert}(Q) \mid \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta \mid \alpha;\beta \mid \texttt{while}(Q)\,\alpha$$

We defined the semantics of this language in terms of its traces, which are finite or infinite sequences of program states mapping variables to integers.

Definition 2 (Trace semantics of programs)**.** The *trace semantics* $\tau(\alpha)$ *of a program* α is the set of all its possible traces and is defined inductively as follows:

1. $\tau(x := e) = \{(\omega, \nu) \ : \ \nu = \omega$ except that $\nu(x) = \omega[\![e]\!]$ for $\omega \in \mathcal{S}\}$
   The final state ν is identical to the initial state ω except in its interpretation of the variable x, which is changed to the value that e has in initial state ω.

2. $\tau(\texttt{assert}(Q)) = \{(\omega, \omega) \ : \ \omega \models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \not\models Q\}$
   The assert stays in its state ω if formula Q holds in ω, otherwise the final state is the error state Λ.

3. $\tau(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta) = \{\sigma \in \tau(\alpha) \ : \ \sigma_0 \models Q\} \cup \{\sigma \in \tau(\beta) \ : \ \sigma_0 \not\models Q\}$
   The $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$ program runs α if Q is true in the initial state and otherwise runs β.

4. $\tau(\alpha;\beta) = \{\sigma \circ \varsigma \ : \ \sigma \in \tau(\alpha)\,,\, \varsigma \in \tau(\beta)\}$;
   the composition of $\sigma = (\sigma_0, \sigma_1, \sigma_2, \dots)$ and $\varsigma = (\varsigma_0, \varsigma_1, \varsigma_2, \dots)$ is

   $$\sigma \circ \varsigma := \begin{cases} (\sigma_0, \dots, \sigma_n, \varsigma_1, \varsigma_2, \dots) & \text{if } \sigma \text{ terminates in } \sigma_n \text{ and } \sigma_n = \varsigma_0 \\ \sigma & \text{if } \sigma \text{ does not terminate} \\ \text{not defined} & \text{otherwise} \end{cases}$$

   The relation $\tau(\alpha;\beta)$ is the composition of traces from $\tau(\beta)$ after those from $\tau(\alpha)$ and can, thus, follow any transition of α through any intermediate state μ to a transition of β.

5. $\tau(\texttt{while}(Q)\,\alpha) = \{\sigma^{(0)} \circ \sigma^{(1)} \circ \dots \circ \sigma^{(n)} \ :$ for some $n \geq 0$ such that for all $0 \leq i < n$: ① the loop condition is true $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]$ and ③ $\sigma^{(n)}$ either does not terminate or it terminates in $\sigma_m^{(n)}$ and $\sigma_m^{(n)} \not\models Q$ in the end$\}$
   $\cup \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \dots \ :$ for all $i \in \mathbb{N}$: ① $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]\}$
   $\cup \{(\omega) \ : \ \omega \not\models Q\}$
   That is, the loop either runs a nonzero finite number of times with the last iteration either terminating or running forever, or the loop itself repeats infinitely often and never stops, or the loop does not even run a single time.

We then discussed safety properties, which are formalized as sets of traces in which for each trace in the property, "something bad" never happens. Safety properties include invariant properties like memory safety and access control, as well as assertions and contracts.

We then introduced first-order dynamic logic, whose formulas let us characterize program behavior.

Definition 3 (DL formula)**.** The *formulas of dynamic logic* (DL) are defined by the grammar (where P, Q are DL formulas, $e, \tilde{e}$ terms, x is a variable, α a program):

$$P, Q \ ::= \ e = \tilde{e} \mid e \leq \tilde{e} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \forall x\, P \mid \exists x\, P \mid [\alpha]P \mid \langle\alpha\rangle P$$

For example, we can use dynamic logic formulas to make the meaning of contracts precise. If we have a contract consisting of `@requires P` and `@ensures Q`, then the corresponding dynamic logic formula is $P \rightarrow [\alpha]Q$ for program α. If this formula is valid, then it means that in any state ω whenever P is true, then all terminating runs of α end with a state in which Q is true.

However, we can't actually reason about the validity of this formula because we have not yet defined the semantics of first-order dynamic logic. We will begin our discussion today by doing so. We will then introduce a set of axioms for dynamic logic that relate the box modality, which refers to programs, to simpler formulas that oftentimes make no reference to programs. We can use these axioms in sequent calculus proofs to reduce statements about program behavior, and in particular whether or not a program satisfies a safety property given as a contract, to a set of simpler statements involving only arithmetic.

3 Dynamic Logic Semantics

One thing that dynamic logic does is to make the meaning of contracts, and later on more general safety properties, completely precise. Of course, we need a semantics to accomplish this, which is our next task.

Like in the semantics for arithmetic formulas, the truth value of a dynamic logic formula depends on a state ω that maps variables to values that we will assume are integers. The semantics for terms in Dynamic Logic is the same as before, and so are the semantics for the predicate $\leq, =$ and logical connectives $\wedge, \vee, \ldots$.

Definition 4 (Semantics of dynamic logic)**.** The DL formula P is true in state ω, written $\omega \models P$, as inductively defined by distinguishing the shape of formula P:

1. $\omega \models e = \tilde{e}$ iff $\omega[\![e]\!] = \omega[\![\tilde{e}]\!]$
2. $\omega \models e \leq \tilde{e}$ iff $\omega[\![e]\!] \leq \omega[\![\tilde{e}]\!]$
3. $\omega \models P \wedge Q$ iff $\omega \models P$ and $\omega \models Q$.
4. $\omega \models P \vee Q$ iff $\omega \models P$ or $\omega \models Q$.

5. $\omega \models \neg P$ iff $\omega \not\models P$, i.e. it is not the case that $\omega \models P$.
6. $\omega \models P \to Q$ iff $\omega \not\models P$ or $\omega \models Q$.
7. $\omega \models P \leftrightarrow Q$ iff both are true or both false, i.e., it is either the case that both $\omega \models P$ and $\omega \models Q$ or it is the case that $\omega \not\models P$ and $\omega \not\models Q$.
8. $\omega \models \forall x\, P$ iff $\nu \models P$ for all states ν that only differ from ω in the value of variable x.
9. $\omega \models \exists x\, P$ iff $\nu \models P$ for at least one state ν that only differs from ω in the value of variable x.
10. $\omega \models [\alpha]P$ iff $\sigma_n \models P$ for all final states σ_n reachable on traces of α starting in ω, i.e. for all finite traces $(\sigma_0, \ldots, \sigma_n) \in [\![\alpha]\!]$ where $\sigma_0 = \omega$, it is true that $\sigma_n \models P$.
11. $\omega \models \langle\alpha\rangle P$ iff there is at least one finite trace σ of α starting in ω where the final state $\sigma_n \models P$, i.e. there exists $(\sigma_0, \ldots, \sigma_n) \in [\![\alpha]\!]$ where $\sigma_n \models P$ and $\sigma_0 = \omega$.

Lemma 5 (Determinism)**.** *The programs α from Def. 1 are* deterministic*, that is, for every initial state ω there is at most one trace σ such that $\sigma \in [\![\alpha]\!]$ and $\sigma_0 = \omega$.*

Proof. The proof is by induction on the structure of the program α and a good exercise. □

Because of determinacy, dynamic logic for the deterministic programs from Def. 1 also satisfy another particularly close relationship of the box and the diamond modality:

Lemma 6 (Deterministic program modality relation)**.** *Because the programs α from Def. 1 are* deterministic*, they make the following formula valid for all formulas P:*

$$\langle\alpha\rangle P \to [\alpha]P$$

Proof. Suppose that $\omega \models \langle\alpha\rangle P$. Then by the semantics of the diamond modality, there is at least one trace $\sigma \in [\![\alpha]\!]$, and moreover that trace is finite and the final state $\sigma_n \models P$. Because of Lemma 5, there is at most one final state for any given initial state, which means that σ is the *only* trace with initial state $\sigma_0 = \omega$, so by the semantics of the box modality $\omega \models [\alpha]P$. □

Colloquially, we also refer to this lemma as the "one for all" principle. We will occasionally have reason to work with a more general notion of programs that is no longer deterministic, so we should carefully mark all uses of this determinism principle to avoid getting confused about which results depend on determinism.

4 Proving statements about program behavior

Consider the following program, which swaps the values between two variables without the need for a superfluous third variable.

$$x := x + y; y := x - y; x := x - y \tag{1}$$

It may not be immediately obvious that this program does what is claimed of it, but luckily we can write a precise specification that describes its behavior using dynamic logic.

$$x = a \wedge y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \wedge y = a) \tag{2}$$

Recalling our discussion of dynamic logic and contracts from before, we could state this in the familiar notation of C0 as:

```
\\@requires x = a && y = b
\\@ensures x = b && y = a
```

The meaning of this dynamic logic formula, and thus the meaning of the program contract that it corresponds to, are now mathematically defined precisely. What can we do with its mathematical semantics? Well, we could, for example, follow the definitions of the semantics to find out how a specific initial state ω changes as the program is executing. Consider the initial state ω with $\omega(x) = 5$ and $\omega(y) = 7$. For this state to satisfy the preconditions, it also needs to have the following values $\omega(a) = 5$ and $\omega(b) = 7$ for variables a and b. Thus,

$$\omega \models x = a \wedge y = b$$

Since the swap program only changes the variables x and y, we only need to track their values, since everything else stays unchanged. After running the first assignment $x := x + y$, the program reaches state a μ_1 with $\mu_1(x) = 12, \mu_1(y) = 7$. After running the second assignment $y := x - y$; from state μ_1 the program reaches a state μ_2 with $\mu_2(x) = 12, \mu_2(y) = 5$. After running the third assignment $x := x - y$; from state μ_2 the program reaches a state ν with $\nu(x) = 7, \nu(y) = 5$. Let's write the respective program statements in the first row and the states in between these in the next rows:

$x := x + y;$	$y := x - y;$	$x := x - y$	
$\omega(x) = 5$	$\mu_1(x) = 12$	$\mu_2(x) = 12$	$\nu(x) = 7$
$\omega(y) = 7$	$\mu_1(y) = 7$	$\mu_2(y) = 5$	$\nu(y) = 5$

All those states agree that a has the value 5 and b the value 7. So indeed, the (only) final state ν satisfies the postcondition:

$$\omega \models x = b \wedge y = a$$

Well that's nice. We followed the semantics of program execution from the particular initial state ω with $\omega(x) = 5$ and $\omega(y) = 7$ and found out that all its final states (well ν

is the only one) satisfy the postcondition that formula (2) claims. This justifies that (2) is true in state ω:

$$\omega \models x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a)$$

Now all we need to do to justify that DL formula (2) is not just true in this particular initial state ω but is valid in all states, is to consider one state at a time and follow the semantics to show the same.

The only downside of that approach of following the semantics through concrete states is that it will keep us busy till the end of the universe because there are infinitely many different states. Even among those initial states that satisfy the precondition $x = a \land y = b$ (otherwise there is nothing to show for (2) since implications are true if their left hand sides are false), there are still infinitely many such states. That's not very practical for such a simple program nor, in fact, for any other interesting program with input.

4.1 Axioms for programs

Our approach to understanding programs with logic is to design one reasoning principle for each program operator that describes its effect in logic with simpler logical operators. If we succeed doing that for every operator that a program can have, then we will understand even the most complicated programs just by repeatedly making use of the respective logical reasoning principles.

Assertions. We'll start with perhaps the simplest program form, at least in terms of what is required for compositional reasoning. The assert statement $\texttt{assert}(Q)$ checks a condition on the current state, and stays in that state if the condition holds and otherwise enters the error state Λ. How can we express $[\texttt{assert}(Q)]P$ in logic in structurally simpler ways?

The formula $[\texttt{assert}(Q)]P$ is true iff formula P holds always after running the assert $\texttt{assert}(Q)$. But if the condition Q is true when the assert is executed, then the state after running $\texttt{assert}(Q)$ is exactly the same as before. If the condition Q is false prior to executing the assert, then the final state of the trace will be Λ, and there is no way that P holds in Λ.

Consequently P holds after all runs of the program $\texttt{assert}(Q)$ iff both the assertion condition Q and the postcondition P are true. This is captured in the assert axiom [assert]:

$$([\text{assert}]) \quad [\texttt{assert}(Q)]P \leftrightarrow (Q \land P)$$

From now on, every time we want to make use of this equivalence, we just refer to it by name: [assert]. And, indeed, this axiom tells us everything we need to know about assert statements. When using the equivalence [assert] from left to right, we can use it to simplify every question about an assert statement of the form $[\texttt{assert}(Q)]P$ by a corresponding structurally simpler formula $Q \land P$. that does not use the assert

statement any more but is logically equivalent. The axiom will enable us, for example to conclude this equivalence:

$$[\texttt{assert}(x < 0)]x \geq 0 \leftrightarrow x < 0 \wedge x \geq 0$$

Also, since axiom [assert] justifies this equivalence, we will be able to reduce a question about whether its left hand side is valid with axiom [assert] to the question whether its corresponding right hand side is valid. In sequent calculus proofs, we will, thus, mark the use of such an axiom by giving its name [assert].

So now we have an axiom that will be useful in proofs. Before we even consider using it, we must convince ourselves that it is sound. The [assert] axiom is an equivalence of first-order dynamic logic, so we must prove that it is a valid one. Having done so, we can use it freely in proofs to replace $\texttt{assert}(P)$ commands with their simplier equivalent form, and vice versa if we have a good reason to do so.

Theorem 7. *The assert axiom [assert] is sound, i.e. all its instances are valid:*

$$[\texttt{assert}(Q)]P \leftrightarrow Q \wedge P$$

Proof. Recall the semantics for assert commands:

$$[\![\texttt{assert}(Q)]\!] = \{(\omega, \omega) \ : \ \omega \models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \not\models Q\}$$

We must show that $\models [\texttt{assert}(Q)]P \leftrightarrow Q \wedge P$, so consider any state ω and show that $\omega \models [\texttt{assert}(Q)]P \leftrightarrow Q \wedge P$. We prove the biimplication by proving each implication seperately.

"$\leftarrow$" Assume the right side $\omega \models Q \wedge P$, and show that $\omega \models [\texttt{assert}(Q)]P$. From this assumption, we know that $\omega \models Q$, and so by the semantics of assert the only trace in $[\![\texttt{assert}(Q)]\!]$ whose initial state is ω is (ω, ω). But our assumption also gives us that $\omega \models P$, so P is also true in the final state. Thus, $\omega \models [\texttt{assert}(Q)]P$.

"$\leftarrow$" Conversely, assume that the left side $\omega \models [\texttt{assert}(Q)]P$ holds, and show $\omega \models Q \wedge P$. By the semantics of the box modality in dynamic logic, all finite traces $\sigma \in [\![\texttt{assert}(Q)]\!]$ have final states in which P is true. By the semantics of assert commands, all traces in which $\omega \not\models Q$ have Λ as the final state. So we know that $\omega \models Q$, in which case $\sigma = (\omega, \omega)$. We can thus conclude $\omega \models Q \wedge P$. □

Now that we know [assert] is sound, we move on to other commands.

Conditionals. We continue on to the formula $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$, which expresses that formula P always holds after running the if-then-else conditional $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$ that runs program α if formula Q is true and runs β otherwise. In order to understand it from a logical perspective, how could we express $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$ in easier ways?

If Q holds then $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$ says that P always holds after running α. If Q does not hold then the same formula $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$ says that P always holds after running β. It is easy to say with a logical formula that P always holds after running α,

which is precisely what $[\alpha]P$ is good for. Likewise $[\beta]P$ directly expresses in logic that P always holds after running β. Both of those formulas $[\alpha]P$ as well as $[\beta]P$ are simpler than the original formula $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$. But, of course, they express something else, because the program $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$ only runs the respective programs conditionally depending on the truth-value of Q.

Yet, there still is a way of expressing $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$ in logic in easier ways with the help of other logical operators. Implications are perfect at expressing the conditions that an if-then statement states in a program. Indeed, if Q holds then $[\alpha]P$ needs to be true and if Q does not hold then $[\beta]P$ for $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$ to hold. Indeed, $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P$ is true if and only if $(Q \to [\alpha]P) \wedge (\neg Q \to [\beta]P)$ is true. We capture this argument once and for all in the if-then-else axiom [if]:

$$([\text{if}])\quad [\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P \leftrightarrow (Q \to [\alpha]P) \wedge (\neg Q \to [\beta]P)$$

Again, before using [if] in proofs we must know that it is sound. Theorem 8 tells us this.

Theorem 8. *The [if] axiom is sound, i.e. all its instances are valid:*

$$[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P \leftrightarrow (Q \to [\alpha]P) \wedge (\neg Q \to [\beta]P)$$

Proof. Having seen the proof of Theorem 7, you should be able to complete this proof as an exercise. □

Sequential composition. The axioms we investigated so far already handle some programs, but sequential compositions are missing quite noticeably and we won't get very far in programs without them. So how can we equivalently express $[\alpha;\beta]P$ in simpler logic without sequential compositions? This formula expresses that P holds after all runs of $\alpha;\beta$, which first runs α and then runs β. How can this be expressed in an easier way in logic, again using just the subprograms α as well as β of $\alpha;\beta$ then?

In order to express $[\alpha;\beta]P$ what we need to say is that after all runs of α it is the case that P holds after all runs of β. It is comparably easy to say that P holds after all runs of β just with the formula $[\beta]P$. But where does this formula need to hold? After all runs of α. In particular, all we need to say is that $[\beta]P$ holds after all runs of α, which is exactly what the formula $[\alpha][\beta]P$ says. We capture these insights in the sequential composition axiom [;]:

$$([;])\quad [\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$$

Indeed, after all runs of $\alpha;\beta$ does P hold if and only if after all runs of α it is the case that after all runs of β does P hold.

Theorem 9. *The sequential composition axiom [;] is sound, i.e. all its instances are valid:*

$$([;])\quad [\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$$

Proof. The proof of this axiom is left as an exercise. □

Assignments. The next case to look into is what we need to prove in order to show the formula $[x := e]p(x)$, which expresses that the formula $p(x)$ holds after the assignment $x := e$ that assigns the value of term e to variable x. How could we reduce this to another logical formula that is simpler?

If we want to show that the formula $p(x)$ holds after assigning the new value e to variable x then we might as well show $p(e)$ right away. And, in fact, p is true of x after assigning e to x if and only if p is true of its new value e. That is, the formula $[x := e]p(x)$ is equivalent to the formula $p(e)$. We capture this argument once and for all in the assignment axiom [:=]:

$$([:=]) \quad [x := e]p(x) \leftrightarrow p(e)$$

Note that the notation $p(x)$ refers to a formula with some *free* occurences of x, and $p(e)$ refers to the same formula with all such free occurences replaced with the term e. Intuitively, a variable is free in a formula if it can be replaced with another variable—a *fresh* variable not appearing anywhere else—and not change the meaning of the formula. For example, consider the formula:

$$[x := y](x < 10)$$

In the sub-formula $x < 10$, x is not free because it is *bound* by the assignment that occurs in the box. If we were to attempt replacing x with a fresh variable z in $x < 10$, then the meaning of the formula completely changes:

$$[x := y](z < 10)$$

The other type of construct that is able to bind variables are the quantifiers $\exists x.p(x)$ and $\forall x.p(x)$. Replace the assignment box with a quantifier in the example above to confirm that this is really the case!

More about variable binding. When substituting free variables in formulas as the assignment axiom tells us to do, we never replace the variables bound by other assignments and quantifiers. In the assignment axiom [:=], the formula $p(e)$ has the term e everywhere in place of where the formula $p(x)$ has the variable x. Of course, it is important for this substitution of e for x to avoid capture of variables and not make any replacements under the scope of a quantifier or modality binding an affected variable. For example, the following formula is an instance of [:=]:

$$[x := y](x \geq 0 \land \forall z\,(x \geq z)) \leftrightarrow (y \geq 0 \land \forall z\,(y \geq z))$$

But the following is not because it would capture the replacement y that is used for x:

$$[x := y](x \geq 0 \land \forall y\,(x \geq y)) \leftrightarrow (y \geq 0 \land \forall y\,(y \geq y))$$

First, observe that replacing x with y in the formula $\forall y(x \geq y)$ completely trivializes the meaning of the quantified formula. Before, the formula placed a meaningful condition

on the value of x, but replacing x with y to obtain $\forall y(y \geq y)$ gives us a formula that is vacuously true. So what happened?

Within the scope of a quantifier that binds a variable y (i.e., $\forall y, \exists y$), the symbol y refers to a different object than outside the quantifier. For example, the following formula is true of the integers, despite giving conflicting constraints on the value of x:

$$x = 0 \land \exists x(x > 0)$$

The reason is that the x referred to by the constraint $x = 0$ is not the same x referred to by the existential quantifier $\exists x(x > 0)$. The quantifier *binds* all occurrences of the variable x in the formula that it quantifies over (in this case, $x > 0$), removing any associations that the symbol x has in the formula containing it. So, we can write the following, which is a valid formula of dynamic logic:

$$[x := 0]x = 0 \land \exists x(x > 0) \leftrightarrow 0 = 0 \land \exists x(x > 0)$$

This is valid because we did not replace the x bound by the existential with 0, which would have resulted in the invalid formula:

$$[x := 0]x = 0 \land \exists x(x > 0) \leftrightarrow 0 = 0 \land \exists x(0 > 0)$$

Quantifiers are not the only constructs that bind variables. Box and diamond modalities of assignment statements do as well! Consider the following:

$$x = a \land y = b \rightarrow [x := x + y][y := x - y][x := x - y]x = b \land y = a \tag{3}$$

Suppose that we proceed by applying [:=] first to the leftmost assignment:

$$x = a \land y = b \rightarrow [y := (x + y) - y][x := (x + y) - y]x + y = b \land y = a \tag{4}$$

Simplifying terms a bit, we have:

$$x = a \land y = b \rightarrow [y := x][x := x]x + y = b \land y = a \tag{5}$$

And now accounting for the new leftmost assignment:

$$x = a \land y = b \rightarrow [x := x]x + x = b \land x = a \tag{6}$$

And because $x := x$ changes nothing, we are left with

$$x = a \land y = b \rightarrow x + x = b \land x = a \tag{7}$$

This is certainly not a valid formula. The problem is that the x assigned by the first assignment statement is different from the x that appears in the postcondition! By the time the program finishes, x has been updated twice, and the last assignment to x binds its occurrence in the formula $x = b \land y = a$. More confusingly, when we changed $[x := x - y]$ in (3) to $[x := (x + y) - y]$ in (4), something very subtle and bad happened. The $(x + y)$ in (4) refer to the **values of x and y when the first assignment took place**,

Aside: closing out proofs with arithmetic. It is quite common for nontrivial arithmetic to be needed during program verification. For example, the proof below ends with the application of the rule $\mathbb{Z}$, which symbolically references something about the integers:

$$\mathbb{Z}\frac{*}{x=a, y=b \vdash x+y-x=b \land x=a}$$

Looking at this step, it seems reasonable to close out the proof, because we can internally simplify $x+y-x$ to y, and then use the identity rule after eliminating the conjunction.
This course is not about proving arithmetic facts, and we will not introduce formal proof rules for reasoning about integer arithmetic. Whenever our proof goal is purely one of arithmetic, and contains no mention of program commands or logical connectives, we will simply avail ourselves of the $\mathbb{Z}$ rule and close out the proof. In fact, this is also what automated program verification and analysis tools do as well: generate a series of arithmetic formulas whose validity implies program correctness, and invoke a *decision procedure* to solve the arithmetic. We will see more of this style of reasoning as we proceed through the course.

and the **second y that is subtracted away in $(x+y)-y$ refers to the value of y after the *second* assignment**. Accordingly, we cannot simplify this instance of $(x+y)-y$ to just x, as this last assignment is not actually a no-op! See the first example we covered in this lecture to convince yourself of this if you are having trouble seeing why.

To get a better handle on what really happens when a program executes a series of assignments to the same variables, consider the following modification of (3) that uses a different variable each time an assignment is made.

$$x=a \land y=b \rightarrow [x_1 := x+y][y_1 := x_1 - y][x_2 := x_1 - y_1]x_2 = b \land y_1 = a \tag{8}$$

This formula now talks about x, x_1, and x_2 to refer to the initial, first, and second updated values that x takes, and y, y_1 for the initial and first updated values that y takes. Here it is easy to see how to apply [:=] in any order, because the variables do not overlap with their future symbols after update. Below we begin after having applied →R and ∧L to move the antecedent of the implication into our assumptions, and eliminated its conjunction.

$$\begin{array}{l}
\mathbb{Z}\dfrac{*}{x=a, y=b \vdash x+y-x=b \land x=a} \\
\mathbb{Z}\dfrac{}{x=a, y=b \vdash x+y-(x+y-y)=b \land x+y-y=a} \\
[:=]\dfrac{}{x=a, y=b \vdash [x_2 := x+y-(x+y-y)]x_2=b \land x+y-y=a} \\
[:=]\dfrac{}{x=a, y=b \vdash [y_1 := x+y-y][x_2 := x+y-y_1]x_2=b \land y_1=a} \\
[:=]\dfrac{}{x=a, y=b \vdash [x_1 := x+y][y_1 := x_1-y][x_2 := x_1-y_1]x_2=b \land y_1=a}
\end{array}$$

Relabeling variables like this so that the same variable is never assigned twice is called putting the program into *static single assignment* (SSA) form. This is a fine thing

to do before conducting a proof, but it is sometimes not obvious how to do so when there are loops in the program. In general, a good way to avoid variable capture issues when working with the assignment axiom is to always start from the last, rightmost assignment, and work your way backwards to the first leftmost one. The following additional rules are also helpful in removing assignments that occur at the beginning of the program, such as through variable initialization.

$$\text{(=R)}\ \frac{\Gamma, x = e \vdash p(e), \Delta}{\Gamma, x = e \vdash p(x), \Delta}\quad \text{(=L)}\ \frac{\Gamma, x = e, p(e) \vdash \Delta}{\Gamma, x = e, p(x) \vdash \Delta}\quad ([:=]_=)\ \frac{\Gamma, y = e \vdash p(y), \Delta}{\Gamma \vdash [x := e]p(x), \Delta}\ (y \text{ fresh})$$

The =R and =L rules allow us to apply equalities that appear in the assumptions either to the proof goals (=R) or to the other assumptions (=L). For example, consider the following proof.

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{y = z, z = w, z < 0 \vdash z < 0}\,\text{id}}{y = z, z = w, w < 0 \vdash z < 0}\,\text{=L}}{y = z, z = w, w < 0 \vdash [x := z]x < 0}\,[:=]}{y = z, z = w, z < 0 \vdash [x := y]x < 0}\,\text{=R}$$

When we make substitutions with these or any rules, we need to be careful with capture just as we do with the assignment axioms. In general, **never replace a variable that is bound by an assignment**, and **never replace the right-hand side of an assignment or quantifier variable binding**.

The rule $[:=]_=$ says that if the first statement of a program is an assignment to x of term e, then we might as well just assume that $x = e$ in the statements that come after. The catch, which may not be surprising at this point, comes from variable capture. When we introduce the assumption that reflects the assignment, we rename x to some fresh, new variable y that is not free anywhere in our current assumptions Γ or anywhere else in p. To see why this is necessary, consider the following attempt where we do not rename x to something fresh.

$$\cfrac{x = 0, x = x + 1 \vdash x = 1}{x = 0 \vdash [x := x + 1]x = 1}\,[:=]_=$$

Right away, we have introduced an arithmetical contradiction into our assumptions with $x = x + 1$! If we had done this the right way, then the proof would have been a breeze:

$$\cfrac{\cfrac{*}{x = 0, z = x + 1 \vdash z = 1}\,\mathbb{Z}}{x = 0 \vdash [x := x + 1]x = 1}\,[:=]_=$$

To finish this important digression on variable by reiterating that most of the time, you should be fine working assignments from right to left in your proofs. The following example shows how to prove the swap program using only the basic assignment axiom, starting from the final rightmost assignment to the first leftmost. These axioms already

enable us to prove the correctness of the integer-based swapping function

$$x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a)$$

All we need to do is turn it into a sequent and start with this as the desired conclusion at the bottom of a sequent proof and successively apply axioms until the proof completes:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{=}a \land y{=}b \vdash y = b \land x = a}\,*}{x{=}a \land y{=}b \vdash x + y - (x + y - y) = b \land x + y - y = a}\,\mathbb{Z}}{x{=}a \land y{=}b \vdash [x := x + y](x - (x - y) = b \land x - y = a)}\,[:=]}{x{=}a \land y{=}b \vdash [x := x + y][y := x - y](x - y = b \land y = a)}\,[:=]}{x{=}a \land y{=}b \vdash [x := x + y][y := x - y][x := x - y](x = b \land y = a)}\,[:=]}{x{=}a \land y{=}b \vdash [x := x + y][y := x - y; x := x - y](x = b \land y = a)}\,[;]}{x{=}a \land y{=}b \vdash [x := x + y; y := x - y; x := x - y](x = b \land y = a)}\,[;]}{\vdash x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a)}\,\rightarrow\mathrm{R}$$

Remember how we mark the use of arithmetic reasoning as $\mathbb{Z}$.

Continuing on. Now let's see how to combine the axioms of multiple different statements in a proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{\geq}0 \vdash x{=}|x|}\,*}{x{\geq}0 \vdash [y := x]\, y{=}|x|}\,\mathbb{Z}}{\vdash x{\geq}0 \rightarrow [y := x]\, y{=}|x|}\,[:=] \quad \dfrac{\dfrac{\dfrac{}{\neg x{\geq}0 \vdash -x{=}|x|}\,*}{\neg x{\geq}0 \vdash [y := -x]\, y{=}|x|}\,\mathbb{Z}}{\vdash \neg x{\geq}0 \rightarrow [y := -x]\, y{=}|x|}\,[:=]}{\vdash (x{\geq}0 \rightarrow [y := x]\, y{=}|x|) \land (\neg x{\geq}0 \rightarrow [y := -x]\, y{=}|x|)}\,\rightarrow\mathrm{R}\,\land\mathrm{R}$$
$$\dfrac{}{\vdash [\mathtt{if}(x{\geq}0)\, y := x\, \mathtt{else}\, y := -x]\, y{=}|x|}\,[\mathrm{if}]$$

This proof shows validity of the following formula, which says that the given program correctly implements the absolute value function $|\cdot|$ from mathematics:

$$[\mathtt{if}(x{\geq}0)\, y := x\, \mathtt{else}\, y := -x]\, y{=}|x|$$

The proof refers to propositional logic sequent calculus rules such as $\land$R and $\rightarrow$R as well as the dynamic logic axioms [if] and [:=]. The proof is developed starting with the desired conclusion at the bottom and working with proof rules to the top as usual in sequent calculus. The proof also makes use of integer arithmetic reasoning (marked by $\mathbb{Z}$) to show that, indeed, if x is nonnegative then x equals the absolute value of x (on the left branch). Likewise, integer arithmetic reasoning is needed to show that if x is negative then $-x$ equals the absolute value of x (on the right branch).

We must not forget to establish the soundness of [:=]. Theorem 10 covers the relevant claim.

Theorem 10. *The [:=] axiom is sound, i.e. all its instances are valid:*

$$[x := e]p(x) \leftrightarrow p(e)$$

Proof. Recall the semantics for assignments:

$$[\![x := e]\!] = \{(\omega, \nu) \ : \ \nu = \omega \text{ except that } \nu(x) = \omega[\![e]\!] \text{ for } \omega \in \mathcal{S}\}$$

We must show that $\models [x := e]p(x) \leftrightarrow p(e)$, so consider any state ω and show that $\omega \models [x := e]p(x) \leftrightarrow p(e)$. We prove the biimplication by proving each implication seperately.

"$\leftarrow$" Assume the right side $\omega \models p(e)$, and show that $\omega \models [x := e]p(x)$. By the semantics of assignment, we know that the only trace in $[\![x := e]\!]$ beginning with ω is (ω, ν) where $\nu = \omega$ everywhere except at x, and $\nu(x) = \omega[\![e]\!]$. But from our assumption we know that $\omega \models p(e)$, i.e. p is true in ω when e is substituted in place of x, and in ν, x is mapped to e and otherwise is the same as ω, so $\nu \models p(x)$. Therefore, $\omega \models [x := e]p(x)$.

"$\rightarrow$" Conversely, assume that the left side $\omega \models [x := e]p(x)$ holds, and show $\omega \models p(e)$. By the semantics of assignment, there is one trace $(\omega, \nu) \in [\![x := e]\!]$ beginning with ω and $\nu = \omega$ except that $\nu(x) = \omega[\![e]\!]$. Our assumption gives us that $\nu \models p(x)$, and because $\nu(x) = \omega[\![e]\!]$, it is also true that $\nu \models p(\omega[\![e]\!])$. Then $\omega \models p(e)$, because $\omega \models p(e)$ if and only if $\omega \models p(\omega[\![e]\!])$, and $\nu = \omega$ at all other variables except x. □

Loops. How can we prove $[\texttt{while}(Q)\,\alpha]P$ in another way by rephrasing it equivalently in logic? What the loop $\texttt{while}(Q)\,\alpha$ does is to test whether formula Q is true and, if so, run α, and then repeating that process until Q is false (if it ever is, otherwise the loop just keeps running α until the end of time).

Let's try to understand that by cases. If Q holds then $[\texttt{while}(Q)\,\alpha]P$ runs α and then runs the while loop afterwards yet again. If Q does not hold then the loop has no effect and just stops right away. That is why $\texttt{while}(Q)\,\alpha$ is equivalent to $\texttt{if}(Q)\,\{\alpha; \texttt{while}(Q)\,\alpha\}$, because both have no effect if Q is false but repeat α as long as Q is true. We can capture these thoughts in the following axiom:

$$([\text{unwind}]) \quad [\texttt{while}(Q)\,\alpha]P \leftrightarrow [\texttt{if}(Q)\,\{\alpha; \texttt{while}(Q)\,\alpha\}]P$$

The [unwind] axiom is sound, as shown in Figure 11.

Theorem 11. *The unwind axiom [unwind] is sound, i.e. all its instances are valid:*

$$([\textit{unwind}]) \quad [\texttt{while}(Q)\,\alpha]P \leftrightarrow [\texttt{if}(Q)\,\{\alpha; \texttt{while}(Q)\,\alpha\}]P$$

Proof. The simplest way to prove this axiom is to show that the semantics of $\texttt{while}(Q)\,\alpha$ are exactly the same as those of $\texttt{if}(Q)\,\{\alpha; \texttt{while}(Q)\,\alpha\}$, i.e.,

$$[\![\texttt{while}(Q)\,\alpha]\!] = [\![\texttt{if}(Q)\,\{\alpha; \texttt{while}(Q)\,\alpha\}]\!]$$

So begin by considering an arbitrary trace $\sigma \in [\![\texttt{while}(Q)\,\alpha]\!]$. According to the semantics for while loops, there are three cases to consider.

1. $\sigma = \sigma^{(0)} \circ \sigma^{(1)} \circ \cdots \circ \sigma^{(n)}$ for some $n \geq 0$ such that for all $0 \leq i < n$: ① the loop condition is true $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]$ and ③ $\sigma^{(n)}$ either does not terminate or it terminates in $\sigma_m^{(n)}$ and $\sigma_m^{(n)} \not\models Q$ in the end.
2. $\sigma = \sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \ldots$ for all $i \in \mathbb{N}$: ① $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]$.
3. $\sigma = (\omega)$ and $\omega \not\models Q$.

In each case, we must show that $\sigma \in [\![\texttt{if}(Q)\,\{\alpha;\texttt{while}(Q)\,\alpha\}]\!]$. The third case follows directly from the semantics of conditionals. The first two cases are best approached by an induction on the number of iterations i, which we leave as an exercise.

To finish the proof, consider a trace $\sigma \in [\![\texttt{if}(Q)\,\{\alpha;\texttt{while}(Q)\,\alpha\}]\!]$ and show that it is also in $[\![\texttt{while}(Q)\,\alpha]\!]$. This will follow similar reasoning as in the other direction, and is also left as an exercise.

□

By applying the [if] axiom and the composition axiom [;] on the right hand side of axiom [unwind], we obtain the following minor variation of axiom [unwind] which we call [unfold]. But on paper we might just as well accept either name, because both axioms follow essentially the same idea and one can easily tell which one we refer to:

$$([\text{unfold}])\quad [\texttt{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\texttt{while}(Q)\,\alpha]P) \wedge (\neg Q \to P)$$

Lemma 12. *The following axiom is a* derived axiom, *so can be proved from the other axioms in sequent calculus, and is, thus, sound:*

$$(*[\textit{unfold}]*)\quad [\texttt{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\texttt{while}(Q)\,\alpha]P) \wedge (\neg Q \to P)$$

Proof. The axiom [unfold] can be proved from the other axioms by using some of them in the backwards implication direction:

$$\dfrac{\dfrac{\dfrac{*}{\vdash [\texttt{while}(Q)\,\alpha]P \leftrightarrow [\texttt{if}(Q)\,\{\alpha;\texttt{while}(Q)\,\alpha\}]P}[\text{unwind}]}{\vdash [\texttt{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha;\texttt{while}(Q)\,\alpha]P) \wedge (\neg Q \to P)}[\text{if}]}{\vdash [\texttt{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\texttt{while}(Q)\,\alpha]P) \wedge (\neg Q \to P)}[;]$$

□

Every time we need the derived axiom [unfold], we could instead write down this sequent proof to prove it. It just won't be very efficient, so instead we will settle for deriving axiom [unfold] in the sequent calculus once and then just believing it from then on.

([:=]) $[x := e]p(x) \leftrightarrow p(e)$

([assert]) $[\texttt{assert}(Q)]P \leftrightarrow (Q \land P)$

([if]) $[\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]P \leftrightarrow (Q \to [\alpha]P) \land (\neg Q \to [\beta]P)$

([;]) $[\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$

([unwind]) $[\texttt{while}(Q)\,\alpha]P \leftrightarrow [\texttt{if}(Q)\,\{\alpha;\texttt{while}(Q)\,\alpha\}]P$

([unfold]) $[\texttt{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\texttt{while}(Q)\,\alpha]P) \land (\neg Q \to P)$

Figure 1: Axioms of the day

5 Summary and next steps

The axioms introduced in this lecture are summarize in Fig. 1.

Today we saw how the axioms of dynamic logic allow us to reason about contract safety properties by reducing such questions to simpler ones involving arithmetic. From a practical standpoint this is a win because there already exist automated tools for solving arithmetic formulas, and we can make good use of them to automatically determine whether a program is safe to run. But this isn't entirely true, as the [unfold] and [unwind] axioms don't actually make things simpler—after applying them, we are left with a formula that is actually more complicated because it contains a copy of the original program! In the next lecture, we will study a technique called *bounded model checking* that uses [unwind] to automatically verify safety properies up to a given execution depth. You will use bounded model checking on C code in the first lab to check for memory safety, but it is not an ideal technique in every setting so in subsequent lectures we will see how to address its shortcomings.