15-316: Software Foundations of Security and Privacy

Lecture Notes on Semantics, Safety, & Dynamic Logic

Matt Fredrikson

Carnegie Mellon University
Lecture 3,4

1 Introduction

This lecture will advance to reasoning about program behavior, using what we learned about the sequent calculus and formal proof as our logical foundation for doing so. We will focus on a class of behaviors called safety properties, which can be broadly understood as specifying behaviors in which "something bad" never happens. To formalize particular instances of "something bad", we will build on what we already know about contracts. In particular, the bad thing characterized by our safety properties will be contract violations.

To make our study of safety properties concrete, we will fix a simple imperative core language with such familiar constructs as assignments, conditionals (i.e., if-then-else), sequential composition (i.e., semicolon), and while loops. This will be sufficient to illustrate the key ideas, and in future lectures we will add features to the language as needed when we discuss various kinds of security policies.

Our understanding of this language and its safety properties will be grounded in first-order dynamic logic [?, ?], which will provide a set of axioms to use in sequent calculus proofs of program behavior. Dynamic logic has been used for many programming languages [?, ?, ?], and is the basis for a number of automated program verification tools [?, ?, ?, ?]. In the following lectures, we will see how to use it to ensure properties like memory safety, access control, and other security-related properties that can be formalized in terms of safety.

2 Review

In the last lecture, we introduced propositional logic by giving its syntax and semantics. Recall that all of the "variables" (i.e., atomic propositions denoted by lower-case letters) represent true/false statements, and the semantics is given in terms of an interpretation I that maps atomic propositions to either *true* (true) or *false* (false) values.

Definition 1 (Syntax of propositional logic). The formulas F, G of propositional logic are defined by the following grammar (where p is an atomic proposition):

$$F ::= \bot \mid \top \mid p \mid \neg F \mid F \wedge G \mid F \vee G \mid F \to G \mid F \leftrightarrow G$$

Definition 2 (Semantics of propositional logic). The propositional formula F is true in interpretation I, written $I \models F$, as inductively defined by distinguishing the shape of formula F:

1. $I \not\models \bot$, i.e., $\bot$ is true in no interpretations
2. $I \models \top$, i.e., $\top$ is true in all interpretations
3. $I \models p$ iff $I(p) = \top$ for atomic propositions p
4. $I \models F \wedge G$ iff $I \models F$ and $I \models G$.
5. $I \models F \vee G$ iff $I \models F$ or $I \models G$.
6. $I \models \neg F$ iff $I \not\models F$, i.e. it is not the case that $I \models F$.
7. $I \models F \to G$ iff $I \not\models F$ or $I \models G$.
8. $I \models F \leftrightarrow G$ iff both are either true or both false.

We then learned about the propositional sequent calculus, a deductive system for proving the validity of propositional formulas. The sequent calculus is comprised of compositional proof rules, and is sound and complete. In other words, by combining sequent calculus proof rules, it is possible to construct a proof for any valid formula (completeness), and it is only possible to construct such a proof for valid formulas (soundness). This property follows from the soundness of the individual proof rules, which holds if and only if the validity of all premises in the rule implies the validity of its conclusion.

Definition 3 (Soundness of a proof rule). A sequent calculus proof rule

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

is sound iff the validity of all premises implies the validity of the conclusion:

$$\text{if } \vDash (\Gamma_1 \vdash \Delta_1) \text{ and } \ldots \text{ and } \vDash (\Gamma_n \vdash \Delta_n) \text{ then } \vDash (\Gamma \vdash \Delta)$$

3 Programs

The first thing we do for the sake of concreteness is to fix the programming language as an imperative core while-programming language with assignments, conditional execution, and while loops.

Definition 4 (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):

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

Of course, imperative programming languages have other control structures, too, but in many cases they are not essential because they can be defined out of these. For example, a repeat-until loop can easily be defined in terms of the while-loop, and a switch statement can be defined in terms of nested conditionals. Likewise, real imperative languages that you have used in the past have more variation on the data types that are supported. Today we start very easily just with a single data type, and assume that all variables hold integer values.

As terms e we use addition and multiplication (but subtraction would be fine to add).

Definition 5 (Terms). Terms are defined by the following grammar ($e, \tilde{e}$ are terms, x is a variable, c is a number literal such as 7):

$$e, \tilde{e} \ ::= \ x \mid c \mid e + \tilde{e} \mid e \cdot \tilde{e}$$

Some applications need further arithmetic operators on terms such as subtraction $e - \tilde{e}$, integer division $e \div \tilde{e}$ provided $\tilde{e} \neq 0$, and integer remainder $e \mod \tilde{e}$ provided $\tilde{e} \neq 0$. Subtraction $e - \tilde{e}$ for example is already expressible as $e + (-1) \cdot \tilde{e}$.

Definition 6 (Arithmetic Formulas). Arithmetic formulas P, Q are defined by the following grammar ($e, \tilde{e}$ are terms, *true*, *false* are literals corresponding to true and false, respectively):

$$P, Q \ ::= \ \mathit{true} \mid \mathit{false} \mid e = \tilde{e} \mid e \leq \tilde{e} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q$$

We now know everything that we need to write programs in our simple imperative core language. As we did in the last lecture with propositional logic, we'll continue by defining semantics that tell us what the syntax means.

3.1 Semantics of terms and formulas

Recall from the previous lecture that we defined the semantics of propositional logic to assign values of either true or false to formulas, given an interpretation I that maps all the atomic propositions (i.e., variables) to true/false values. Our intuition about programs tells us that they do not map to true/false values, but rather execute to change the state of a machine until they (hopefully) terminate. For example, a program consisting of

the assignment $x := x + 1$ will move from an initial state ω to a new state ν that has a different value for the variable x, namely exactly such that $\nu(x) = \omega + 1$ while no other variables change their value.

We formalize a program state ω as a function assigning an integer value in $\mathbb{Z}$ to every variable. The set of all states is denoted $\mathcal{S}$. In addition, we will assume that $\mathcal{S}$ contains a special error state Λ. For example, a program might end up in the error state after attempting to divide by zero (in case our language supports division), or after executing a failed assertion command. The special state Λ does not map values for any variables, so no terms can be evaluated in it and no program can continue executing from Λ.

The value that a term e has in a state ω is written $\omega[\![e]\!]$ and defined by simply evaluating when using the concrete (integer) values that the state ω provides for all the variables in term e.

Definition 7 (Semantics of terms). The semantics of a term e in a state $\omega \in \mathcal{S}$ is its value $\omega[\![e]\!]$. It is defined inductively by distinguishing the shape of term e as follows:

- $\omega[\![x]\!] = \omega(x)$ for variable x
- $\omega[\![c]\!] = c$ for number literals c
- $\omega[\![e \odot \tilde{e}]\!] = \omega[\![e]\!] \odot \omega[\![\tilde{e}]\!]$, where $\odot \in \{+, \times\}$

Note that if $\omega = \Lambda$ then the state does not map any variables to values, and there is no way to evaluate a term. So if $\omega = \Lambda$ then $\omega[\![e]\!]$ is undefined.

The arithmetic formulas used in our programming language are distinct from propositional logic in an important way. Namely, rather than containing propositional variables their variables correspond to integer values. Otherwise we define the semantics of arithmetic formulas just as we did for propositional formulas, by enumerating rules for the relation $\omega \models P$ that evaluate formulas to either true or false.

Definition 8 (Semantics of arithmetic formulas). The DL formula P is true in state ω, written $\omega \models P$, as inductively defined by distinguishing the shape of formula P:

1. $\omega \not\models \bot$, i.e., $\bot$ is true in no states
2. $\omega \models \top$, i.e., $\top$ is true in all states
3. $\omega \models e = \tilde{e}$ iff $\omega[\![e]\!] = \omega[\![\tilde{e}]\!]$
4. $\omega \models e \leq \tilde{e}$ iff $\omega[\![e]\!] \leq \omega[\![\tilde{e}]\!]$
5. $\omega \models P \wedge Q$ iff $\omega \models P$ and $\omega \models Q$.
6. $\omega \models P \vee Q$ iff $\omega \models P$ or $\omega \models Q$.
7. $\omega \models \neg P$ iff $\omega \not\models P$, i.e. it is not the case that $\omega \models P$.
8. $\omega \models P \rightarrow Q$ iff $\omega \not\models P$ or $\omega \models Q$.

9. $\omega \models P \leftrightarrow Q$ 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 $\omega \not\models Q$.

Note that if $\omega = \Lambda$ then the state does not map any variables to values, and there is no way to evaluate terms to integer values, and thus no way to evaluate the predicates $\leq$ and $=$. So as in the case of terms, if $\omega = \Lambda$ then $\omega \models P$ is undefined.

3.2 Program semantics

The semantics of a program is comprised of the set of traces generated by running the program starting in any initial state ω. To fully appreciate what this means, we first need some background on what a trace is.

A trace σ is either a finite sequence of states $(\sigma_0, \sigma_1, \ldots, \sigma_n)$ of some length $n \in \mathbb{N}$, or an infinite sequence of states, one for each natural number: $(\sigma_0, \sigma_1, \sigma_2, \ldots)$. We say that a trace terminates if and only if it is finite, and its last state is not the error state Λ. If a trace is finite and its last state is Λ, we say that it aborts. Otherwise, we say that the trace diverges if it is infinite.

Definition 9 (Trace semantics of programs). The trace semantics $[\![\alpha]\!]$ of a program α is the set of all its possible traces and is defined inductively as follows:

1. $[\![x := e]\!] = \{(\omega, \nu) \ : \ \nu = \omega \text{ except that } \nu(x) = \omega[\![e]\!] \text{ for } \omega \in \mathcal{S}\}$
   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. $[\![\texttt{assert}(Q)]\!] = \{(\omega, \omega) \ : \ \omega \models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \not\models Q\}$
   The assert stays in its state ω if formula Q holds in ω, otherwise the final state is the error state Λ.

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

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

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

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

5. $[\![\texttt{while}(Q)\,\alpha]\!] = \{\sigma^{(0)} \circ \sigma^{(1)} \circ \cdots \circ \sigma^{(n)} \ : \ \text{for some } n \geq 0 \text{ such that for all } 0 \leq i \leq n:$
   ① the loop condition is true $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]$ and ③ $\sigma^{(n)}$ is either infinite or, if finite, ends with $\sigma_m^{(n)}$ and $\sigma_m^{(n)} \not\models Q\}$

$\cup \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \ldots \ : \ \text{for all } i \in \mathbb{N}\text{: } ① \ \sigma_0^{(i)} \models Q \text{ and } ② \ \sigma^{(i)} \in [\![\alpha]\!]\}$
$\cup \{(\omega, \omega) \ : \ \omega \not\models Q\}$
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.

4 Safety properties

Now that we have defined the semantics of programs in terms of their traces, we are in a better position to understand safety properties. Intuitively, safety properties characterize programs in which "something bad" never happens. Which "bad thing" we care about is precisely what a policy is meant to express. A few examples of safety properties that you may already be familiar with are:

- Memory safety refers to a class of safety properties that characterize appropriate use of memory resources by software. One aspect of memory safety in C programs has to do with making sure that each time an array is accessed, the index is within the bounds of 0 and the allocated length of the array. In this case the "bad thing" that must never happen is the program accessing an array outside of its allocated bounds.
- Mutual exclusion has to do with using a shared resource, and characterizes behaviors in which two processes never access the shared resource at the same time. In other words the "bad thing" that defines safety property is the event in which two processes simultaneously access their shared resource.
- Access control policies correspond to safety properties. A typical access control policy defines a set of principals who make use of the system, a set of resources that need protection, and a list of rules that define which principals may use which resources. The corresponding safety property is based on the "bad thing" which is that someone accesses a resource not allowed by the policy rules.

Each of the properties listed above is an example of an invariant, which is an important class of safety properties. Invariants require that some condition P holds in all reachable states of the program.

Definition 10 (Invariant property). An invariant property Φ characterizes a set of traces where some formula P holds in every state of every trace.

$$\Phi = \{\sigma \in \mathit{Traces}(\mathcal{S}) \ : \ \text{there does not exist } i \text{ where } \sigma_i \not\models P\}$$

The formula P is called an invariant condition of Φ.

Invariants are not the only type of safety property, and are not the only useful kind of safety property for defining security policies. Consider a basic user login module, which requires that users enter the correct password before proceeding with any further

operations. This requirement is not an invariant, because it cannot be expressed in terms of a formula that must hold over every state in the module's execution traces. It is however a safety property because it can be characterized by the "bad event" of a user proceeding past login without entering the correct password.

In general, we can formalize any safety property by defining a set of prefixes such that if a trace ever starts out with such a prefix, then it can never satisfy the corresponding safety property regardless of what corrective steps it might attempt to make. Thinking of our examples from before, if a program ever lets someone execute a command without providing a correct password, then safety has been violated because it has a trace that starts out by allowing the bad thing to happen.

Definition 11 (Safety property). A set of traces Φ is a safety property if for all traces $\sigma \in \mathit{Traces}(\mathcal{S}) \setminus \Phi$, there exists a finite prefix $\hat{\sigma}$ of σ such that:

$$\Phi \cap \{\sigma' \in \mathit{Traces}(\mathcal{S}) \ : \ \hat{\sigma} \text{ is a prefix of } \sigma'\} = \emptyset$$

In other words, we can think of a safety property as an enumeration of every possible trace that does not begin with one of the bad prefixes that we wish to avoid; every trace not in Φ has some bad prefix $\hat{\sigma}$ that is not shared by any trace in Φ. This means that properties do not necessarily apply to any particular program, and a safety property may contain traces from many programs. Given an arbitrary program α and property Φ, it then makes sense to ask whether α satisfies Φ. We answer this question by determining whether the set of traces in the semantics of α is a subset of those that define Φ; if so, then the program satisfies the property, and otherwise it does not.

Definition 12 (Trace property satisfaction). A program α satisfies a trace property Φ if and only if $[\![\alpha]\!] \subseteq \Phi$. In other words, if all of the behaviors of α are explicitly allowed by the property, then α satisfies Φ.

We will now go into further detail on two types of safety properties that are especially useful when reasoning about programs written in our core language.

4.1 Assertions and aborted execution

One of the commands in our language is `assert`(P), where P is some arithmetic formula over the program variables. An assertion effects no change on the program state if the formula evaluates to true, and otherwise aborts the program by transitioning to Λ. In the latter case, the program is said to violate the assertion.

We can define a safety property Φ_Λ that characterizes all programs that never violate their assertions by the set of bad prefixes $\widehat{\Phi_\Lambda}$ shown in Equation 1.

$$\widehat{\Phi_\Lambda} = \{\hat{\sigma} \in \mathit{Traces}(\mathcal{S}) \ : \ \hat{\sigma} \text{ is finite and the final state } \hat{\sigma}_n = \Lambda\} \tag{1}$$

In words, the bad prefixes characterized by Equation 1 comprise all aborted traces, i.e., those that end in the error state Λ. Then the safety property Φ_Λ is the set of all traces that never abort after a finite number of steps.

$$\Phi_\Lambda = \{\sigma \in \mathit{Traces}(\mathcal{S}) \ : \ \sigma \text{ does not begin with any } \hat{\sigma} \in \widehat{\Phi_\Lambda}\} \tag{2}$$

Φ_Λ is a tremendously useful property because it captures unwanted behavior that may arise from a broad set of erroneous and insecure program behaviors. It also allows us as programmers to use the `assert`(P) syntax to specify what we intend for the policy to be, and then if we have a way of checking whether our program satisfies Φ_Λ, we can use it to check our policy.

4.2 Program contracts

Going back to 15-122, program contracts given by `@requires` and `@ensures` clauses also define safety properties. In the previous lecture, we looked at a binary multiplication function annotated with the following C0 contract.

```
//@requires b >= 0;
//@ensures  \result = a*b
```

The meaning of this contract is that whenever the function begins executing with $b \geq 0$, then when it finishes the return value will hold the value $a \cdot b$. Our core language doesn't have functions or procedures yet, so from now on we will view contracts as applying to the execution of the entire program. In other words, `@requires` clauses hold at the very beginning of execution, and `@ensures` apply to the program state upon termination (if the program actually terminates). The language also doesn't have a `return` command, or any variable designated to hold results, so we'll state the postcondition in terms of the variable used to store the result prior to returning. The contract will be:

```
//@requires b >= 0;
//@ensures  z = a*b
```

Why is this a safety property? Think about the contract's meaning in terms of bad prefixes. A bad prefix for this contract would be a finite trace σ whose initial state $\sigma_0 \models b \geq 0$, and whose final state $\sigma_n \not\models z = a \cdot b$. Alternatively, a bad prefix could be one where $\sigma_0 \models b \geq 0$ and the final state $\sigma_n = \Lambda$.

In general, we can formalize the safety property corresponding to a contract given by `@requires P` and `@ensures Q` as shown in Equation 3.

$$\Phi = \{\sigma \in \mathit{Traces}(\mathcal{S}) \ : \ \sigma \text{ is finite of length } n, \sigma_0 \models P, \text{ and } \sigma_n \models Q\} \tag{3}$$

Notice that Equation 3 does not make any mention of the error state, although our description of the bad prefixes did. Consideration of Λ in this case is already taken care of by the way we defined the semantics of arithmetic formulas. If $\sigma_n = \Lambda$, then it is not the case that $\sigma_n \models Q$ because $\models$ is not defined on the error state.

5 Reasoning about safety properties and programs

We continue in our study of safety properties now by considering a way of determining whether a given program satisfies a property. We won't tackle the whole problem of verifying arbitrary safety properties yet, and will instead focus on an approach that

works well for contracts. Along the way, we will discover tools that will help us with different types of safety properties later on.

Given one particular value for each of the variables, the arithmetic formulas that define a contract are either true or false (much like, in an interpretation I, propositional logic formulas are either true or false). Looking at Equation 3, we see that the values used to evaluate the precondition P are taken from the initial state σ_0, and those used to evaluate the postcondition Q from the final state σ_n. In this sense the logic used to define the pre- and postconditions is static, or fixed according to the values in a single state or interpretation.

Why is this problematic? Consider a situation where the precondition evaluates to true in the initial state, but becomes false in the final state. If the logic that we use to reason about whether the states generated by a program gives us no way of referring to what was true at the beginning, and what will be true at the end of a trace, then how can we possibly figure out that this trace violates the contract? Contracts deal with the dynamics of state as programs make changes to variables, and we need a logical formalism that can do so as well.

Dynamic logic provides modalities that talk about what is true after a program runs. The modal formula $[\alpha]P$ expresses that the formula P is true after all terminating runs of program α. That is, the formula $[\alpha]P$ is true in a state ω if it is indeed the case that all states ν reached after running program α starting in ω satisfy the postcondition P. As we will see, we can use dynamic logic to rigorously express what contracts mean, as well as to reason about them by way of sequent calculus proofs. But let's first officially introduce the language of dynamic logic.

5.1 Dynamic Logic

Definition 13 (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):

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

The propositional connectives such as $\wedge, \vee, \ldots$ and predicates $\leq, =$ mean what they already mean in Definition 8, and terms $e, \tilde{e}$ are constructed exactly as in Definition 5. The universal quantifier in $\forall x\, P$ and the existential quantifier in $\exists x\, P$ quantify over all (in the case of $\forall$), or over some (in the case of $\exists$) value of the variable x. But it will be quite important to settle on the domain of values that both quantifiers range over. In most of our applications, this will be the set of integers $\mathbb{Z}$, but other domains are of interest, too.

Most importantly, and indeed the defining characteristic of dynamic logic, are the box modality in $[\alpha]P$ and the diamond modality in $\langle\alpha\rangle P$. The modal formula $[\alpha]P$ is true in a state iff the final states of all runs of program α beginning in that final state satisfy the postcondition P. Likewise the modal formula $\langle\alpha\rangle P$ is true in a state iff there is a final state for at least one run of program α beginning in that final state that satisfies the postcondition P. So $[\alpha]P$ expresses that P is true after all terminating runs of α

whereas $\langle\alpha\rangle P$ expresses that P is true after at least one run of α. We will focus almost exclusively on the box modality.

5.2 Contracts in Dynamic Logic

Since the box modality in $[\alpha]P$ expresses that formula P holds after all runs of program α, we can use it directly to express `@ensures` postconditions. Let `bmult` be the binary multiplication program from the previous lecture, `pre` and `post` be the pre and post-conditions.

$$\texttt{bmult} \equiv x := a; y := b; z := 0; \texttt{while}(y > 0)\,\{\texttt{if}(y\%2 = 1)\,\{z := z + x\}x := 2 * x; y := y \div 2\}$$
$$\texttt{pre} \equiv b \geq 0$$
$$\texttt{post} \equiv z = a \cdot b$$

With these abbreviations and the box modalities of dynamic logic it suddenly is a piece of cake to express the `@ensures` postcondition holds after all program runs:

$$[\texttt{bmult}]\texttt{post}$$

Suppose that we had a second postcondition $\texttt{post2} \equiv y \leq b$. Well, if we want to say that both postconditions are true after running `bmult` and the logic is closed under all operators including conjunction, we can simply use the conjunction of both formulas for the job:

$$[\texttt{bmult}]\texttt{post} \wedge [\texttt{bmult}]\texttt{post2}$$

This formula means that `post` is true after all runs of `bmult` and that `post2` is also true after all runs of `bmult`. Maybe it would have been better to simultaneously state both postconditions at once? That is simply the formula

$$[\texttt{bmult}](\texttt{post} \wedge \texttt{post2})$$

which says that the conjunction of `post` and `post2` is true after all runs of `bmult`. Which formula is better now?

Well that depends. For one thing, both are perfectly equivalent, because that is what it means for a formula to be true after all runs of a program. That means the following biimplication in dynamic logic is valid so true in all states:

$$[\texttt{bmult}]\texttt{post} \wedge [\texttt{bmult}]\texttt{post2} \;\leftrightarrow\; [\texttt{bmult}](\texttt{post} \wedge \texttt{post2})$$

Now that we have worried so much about how to state the postcondition in a lot of different equivalent ways, the question is whether the following formula or any of its equivalent forms is actually always true?

$$[\texttt{bmult}](\texttt{post} \wedge \texttt{post2})$$

The answer is of course "no", because we forgot to take the program's precondition from the `@requires` clause into account, which the program assumes to hold in the initial

state. But that is really easy in logic because we can simply use implication for the job of expressing such an assumption:

$$\texttt{pre} \rightarrow [\texttt{bmult}](\texttt{post} \wedge \texttt{post2})$$

And, indeed, this formula will now turn out to be valid, so true in all states. In particular, in every initial state it is true that if that initial state satisfies the `@requires` preconditions $b \geq 0$, then all states reached after running the `gcd` program will satisfy the `@ensures` postconditions $\texttt{post} \wedge \texttt{post2}$. If the initial state does not satisfy the precondition, then the implication does not claim anything, because it makes an assumption about the initial state that apparently is not presently met.

6 Next lecture

In this lecture we formalized safety properties in terms of the semantics of programs in our core language. We then zoomed in on one particular kind of safety property corresponding to program contracts, and saw how to make their meaning precise using dynamic logic. This is nice, but we didn't go to all the trouble of introducing (another) new logic just to write contracts down in a different way. In the next lecture, we will study several useful axioms of dynamic logic, and see how to use them in sequent calculus proofs of program safety. This will lay the groundwork for understanding how two important automated safety verification techniques, called bounded model checking and symbolic execution, work to identify safety vulnerabilities in real code.