Lecture Notes on Proving Safety

15-316: Software Foundations of Security & Privacy
Matt Fredrikson

Lecture 5
January 26, 2026

1 Introduction

So far we have focused attention on proving general properties of programs. Such properties are certainly relevant to safety, but how does this now translate to safety? And how exactly do we then prove safety? Looking at our language up to now, all constructs are "safe". For example, we operate on integers, so there is no overflow. In the next lecture we will consider out-of-bounds memory access as a quintessential unsafe operation; in this lecture we consider division by 0. In a language like C, the behavior of division by 0 is *undefined*. For C, undefined behavior gives the compiler a lot of leeway. For example, it could raise an exception. But it could also optimize $(1/0) * 0$ to just 0 instead of raising an exception. Or it could exhibit some other unexpected behavior that could give an attacker access to your machine.

Because "*undefined*" has different meanings in different contexts, we avoid this term. Instead we use:

Unsafe: If an operation is *unsafe* we do not know what an implementation of a language might do. In particular, we consider *all* safety properties as being violated by an unsafe operation. In C, this would be called *undefined behavior*.

Indeterminate: If an operation is *indeterminate* it has a valid outcome, but the language specification does not say what precisely this outcome is. In C, the order of evaluation for many expressions is *unspecified* so that there may be many outcomes that are all correct.

Safe: The program performs no unsafe operation. This includes situations where the outcome may be indeterminate.

Basically, we fix a subset of all safety properties, namely those that arise from a single operation deemed *unsafe*.

This particular (even if restricted) concept of safety already raises the issue that dynamic logic only relates an initial state to a final state, but does not explicitly mention any intermediate states. So we have to start by extending dynamic logic so it can reflect the concept of an unsafe program. We do not have an explicit *predicate* inside the logic that expresses a program α is safe, but it will nevertheless be easy to write formulas that imply safety.

2 Soundess by Derivation

We ended the previous lecture by stating several axioms for Dynamic Logic.

$[:=]A$	$[x := e]Q(x) \leftrightarrow \forall x'.x' = e \rightarrow Q(x')$ (x' not in e or $Q(x)$)
$[;]A$	$[\alpha\ ;\ \beta]Q \leftrightarrow [\alpha]([\beta]Q)$
$[\mathbf{if}]A$	$[\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q \leftrightarrow (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q)$
$[\mathbf{unfold}]A$	$[\mathbf{while}\ P\ \alpha] \leftrightarrow (P \rightarrow [\alpha][\mathbf{while}\ P\ \alpha]Q) \wedge (\neg P \rightarrow Q)$

We saw how axioms that consist of biconditionals lead to proof rules that are sound an invertible. For conditional statements, we would directly obtain the left and right rules,

$$\frac{\Gamma \vdash (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q), \Delta}{\Gamma \vdash [\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q, \Delta}\ [\mathbf{if}]R \qquad \frac{\Gamma, (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q) \vdash \Delta}{\Gamma, [\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q \vdash \Delta}\ [\mathbf{if}]L$$

However, these rules introduce several connectives that hide the essential proof obligations that need to be discharged to reason about a conditional. This is fixed easily by using the rules that we already have for logical connectives. For example, we can derive as follows on the right rule.

$$\dfrac{\dfrac{\dfrac{\Gamma, P \vdash [\alpha]Q, \Delta}{\Gamma \vdash P \rightarrow [\alpha]Q, \Delta}\ \rightarrow R \quad \dfrac{\dfrac{\Gamma \vdash P, [\beta]Q, \Delta}{\Gamma, \neg P \vdash [\beta]Q, \Delta}\ \neg L}{\Gamma \vdash \neg P \rightarrow [\beta]Q, \Delta}\ \rightarrow R}{\Gamma \vdash (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q), \Delta}\ \wedge R}{\Gamma \vdash [\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q, \Delta}\ [\mathbf{if}]R$$

Of course we aren't able to close the proof entirely, and we're left with two premises that, were they giveen proofs, would prove the original goal. So this gives us a new rule.

$$\frac{\Gamma, P \vdash [\alpha]Q, \Delta \quad \Gamma \vdash P, [\beta]Q, \Delta}{\Gamma \vdash [\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q, \Delta}\ [\mathbf{if}]R$$

Note that this rule immediately surfaces the proof obligations, short circuiting past the connectives from the axiom. We can do the same thing for the left rule as well,

which would lead to the following rule.

$$\frac{\Gamma, P, [\alpha]Q \vdash \Delta \quad \Gamma, [\beta]Q \vdash P, \Delta}{\Gamma, [\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]Q \vdash \Delta}\ [\textbf{if}]L$$

Assignments. In order to do this for the assignment axiom, we'll need rules for reasoning about the universal quantifier.

$$\frac{\Gamma \vdash P(x'), \Delta}{\Gamma \vdash \forall x.\, P(x), \Delta}\ \forall R^{x'} \qquad \frac{\Gamma, \forall x.\, P(x), x' = e, P(x') \vdash \Delta}{\Gamma, \forall x.\, P(x) \vdash \Delta}\ \forall L^{x'}$$

We write $\forall R^{x'}$ to denote that x' is a fresh new variable that doesn't appear anywhere in the context of Γ, Δ, or P. Reading the rules $\forall L^{x'}$ bottom-up, we can freely choose the expression e to instantiate the quantifier with as long as it doesn't mention x'. Now we show the derivation of the right rule for assignment.

$$\cfrac{\cfrac{\cfrac{\Gamma, x' = e \vdash Q(x')}{\Gamma \vdash x' = e \to Q(x')}\ {\to}R}{\Gamma \vdash \forall x'.x' = e \to Q(x'), \Delta}\ \forall R^{x'}}{\Gamma \vdash [x := e]Q(x), \Delta}\ [:=]R^{x'}$$

Doing the same for the left rule gives us the following direct rules for assignment.

$$\frac{\Gamma, x' = e \vdash Q(x'), \Delta}{\Gamma \vdash [x := e]Q(x), \Delta}\ [:=]R^{x'} \qquad \frac{\Gamma, x' = e, Q(x') \vdash \Delta}{\Gamma, [x := e]Q(x) \vdash \Delta}\ [:=]L^{x'}$$

3 Loops

How does a loop **while** P α execute? If P is true then we execute the loop body α once, followed again by **while** P α. If P is false we just exit the loop. The following rule suggests itself:

$$\frac{\Gamma \vdash [\textbf{if } P \textbf{ then } (\alpha \,;\, \textbf{while } P\ \alpha) \textbf{ else skip}]Q, \Delta}{\Gamma \vdash [\textbf{while } P\ \alpha]Q, \Delta}\ [\textbf{unwind}]R$$

Here we made up a new program **skip** that doesn't do anything. It behaves like the unit of parallel composition in that **skip** ; α is equivalent to α. We could use $x := x$ instead, but that seems more complicated because it mentions a variable.

The problem with our first rule is that it replaces a program with a larger one, so it is not reductive. We can use the rules we already have to simplify it a bit to

$$\frac{\Gamma, P \vdash [\alpha]([\textbf{while } P\ \alpha])Q, \Delta \quad \Gamma, \neg P \vdash Q, \Delta}{\Gamma \vdash [\textbf{while } P\ \alpha]Q, \Delta}\ [\textbf{unfold}]R$$

This is better, but in the first premise we still have to reason about exactly the same program. So while these two rules are sound, their application is somewhat limited as we will see in the next lecture.

If you think back to 15-122 *Principles of Imperative Computation* you may remember how we reasoned about loops: we used *loop invariants*. In that course, loop invariants (like pre- and post-conditions for functions) where themselves *executable*. Here they are formulas and subject to logical reasoning. How do loop invariants work? Let's look at a trivial program:

$$\textbf{while}\ (x > 1)\ x := x - 2$$

under the precondition that (say) $x \geq 6$. After the loop we know that if the initial x was even, then in the poststate x must be 0, and if the initial x was odd, then in the poststate x must be 1. For safety properties that may a bit specific, so here we only want to ascertain that $0 \leq x \leq 1$ in the poststate.

In dynamic logic we express this as the proposition

$$x \geq 6 \rightarrow [\textbf{while}\ (x > 1)\ x := x - 2]\, 0 \leq x \leq 1$$

But how do we prove it? What is the loop invariant? Recall:

- The loop invariant must be true initially.
- The loop invariant must be preserved by the loop body, under the assumption that the loop guard is true.
- The postcondition of the loop must be implied by the loop invariant together with the negated loop guard.

If we can prove all three of these then we can conclude the postcondition of the loop. In this example, we pick the loop invariant J to be $x \geq 0$. Then we have to prove:

True Initially $x \geq 6 \vdash x \geq 0$

Preserved $x \geq 0, x > 1 \vdash [x := x - 2]\ x \geq 0$

Implies Postcondition $x \geq 0, \neg(x > 1) \vdash 0 \leq x \leq 1$

These are all easy to prove (with an oracle for arithmetic), reducing the one in the middle in one step (using rule $[:=]R^{x'}$) to

$$x \geq 0, x > 1, x' = x - 2 \vdash x' \geq 0$$

Summarizing all of this with a rule yields the following, for an arbitrary loop invariant J.

$$\frac{\Gamma \vdash J, \Delta \quad J, P \vdash [\alpha]J \quad J, \neg P \vdash Q}{\Gamma \vdash [\textbf{while}\ P\ \alpha]Q, \Delta}\ [\textbf{while}]R$$

Sadly, since J is an arbitrary formula, this rule is not reductive. It would be okay, however, if we forced the programmer to write J in the program (as we do in C0), because then each premise only refers to components of the conclusion. When we want to emphasize this point we may write

$$[\textbf{while}_J\ P\ \alpha]Q$$

where J is the loop invariant.

An important point about this rule is that we drop Γ and Δ in the second and third premise. This is because we don't know how often we may have to go around the loop. Preservation (the second premise) has to hold for any state we might reach during the iteration, but the antecedents in Γ are only guaranteed *before* the first iteration.

In our example, $x \geq 6$ is only known to be true before the loop starts, and not after each iteration. In fact, it may be false after the first iteration and therefore we cannot use it to prove for the second and third premise. Similarly, the additional succedents Δ also must be dropped. Otherwise we could reformulate the goal

$$\cdot \vdash [\textbf{while}\ (x > 1)\ x := x - 2]\, 0 \leq x \leq 1, \neg(x \geq 6)$$

and then use $\neg R$ rule to turn the succedent $\neg(x \geq 6)$ into the (unwarranted) antecedent $x \geq 6$.

Putting all of this together, we can prove our program as follows.

$$\dfrac{\dfrac{\text{(by arithmetic)}\ x \geq 6 \vdash x \geq 0 \qquad \dfrac{\overset{\text{(by arithmetic)}}{x \geq 0, x > 1, x' = x - 2 \vdash x' \geq 0}}{x \geq 0, x > 1 \vdash [x := x - 2]\, x \geq 0}\ [:=]R^{x'} \qquad \overset{\text{(by arithmetic)}}{x \geq 0, \neg(x > 1) \vdash 0 \leq x \leq 1}}{x \geq 6 \vdash [\textbf{while}\ (x > 1)\ x := x - 2]\, 0 \leq x \leq 1}\ [\textbf{while}]R^{*}}{\cdot \vdash x \geq 6 \to [\textbf{while}\ (x > 1)\ x := x - 2]\, 0 \leq x \leq 1}\ \to R$$

$(*)$ with loop invariant $J(x) = (x \geq 0)$

4 Unsafe Programs

We might deem expressions such as $1/0$ as inherently *unsafe*. But formulas include expressions, and it is unclear what "unsafe formulas" would be, or how we reason with them logically and correctly. It is actually possible to design logics where formulas may be true or false or undefined, that is, may not have a truth value (see, for

example, the article about *Free Logic* [Nolt, 2010]). This would be a rather drastic revision and further depart from what theorem provers and decision procedures offer. Another standard path is to consider such expressions as *indeterminate*. In that case, we simply won't be able to deduce much about indeterminate expressions. For example, an axiom about the quotient a/b and remainder $a \,\%\, b$ might state

$$0 \leq r < b \wedge a = q * b + r \rightarrow a/b = q \wedge a \,\%\, b = r$$

The antecedent of the implication cannot be satisfied if $b = 0$ so in that case we can't deduce anything about the nature of a/b or $a \,\%\, b$ except that they are integers (since all variables here are typed as integers).

Since expressions are shared between formulas and programs we therefore simply declare all expressions to be *safe*, although possibly indeterminate. An intuitively unsafe expression then is elevated to the level of commands. Here we use the terminology *command* for a primitive part of a program such as assignment ($x := e$) or **skip** (which has no effect). Commands are included in the grammar for programs. Our new form of command is $x := \mathbf{divide}\ e_1\ e_2$. This is *unsafe* if e_2 denotes 0; otherwise it assigns to x the result of e_1/e_2 (integer division).

We emphasize: $\mathbf{divide}\ e_1\ e_2$ is not a new *expression* (because all expressions should remain safe), but $x := \mathbf{divide}\ e_1\ e_2$ is a new command. This means an ordinary assignment such as $x := x/y + 1$ is not part of our language. It would instead have to be expressed as $t := \mathbf{divide}\ x\ y\ ;\ x := t + 1$ where t is a fresh variable (often called a *temporary variable* in a compiler).

In order to capture unsafe behavior, we semantically characterize unsafe programs using the form

$$\omega[\![\alpha]\!]\lightning$$

which means that α is unsafe when executed starting in state ω. We can think of the symbol $\lightning$ as denoting an unsafe state, different from the states we have been using so far (ω, μ, ν) from which the program can proceed as expected. Formally, we don't change our definition of state as a total map from variables to integers, which is why we instead introduce a new notation and new relation. Starting with our new command, we have the following two clauses:

$$\omega[\![x := \mathbf{divide}\ e_1\ e_2]\!]\nu \text{ iff } \omega[\![e_1]\!] = a, \omega[\![e_2]\!] = b, c = a/b \text{ and } \nu = \omega[x \mapsto c]$$
$$\text{provided } b \neq 0$$

$$\omega[\![x := \mathbf{divide}\ e_1\ e_2]\!]\lightning \text{ iff } \omega[\![e_2]\!] = 0$$

We need to be careful (and want to prove) that there is no program α such that $\omega[\![\alpha]\!]\nu$ for some ν and at the same time $\omega[\![\alpha]\!]\lightning$. That is, unsafe programs never have a final state, and programs with a final state are never unsafe. On the other hand, it is possible for a program to have no final state and yet be safe—these are programs that execute safely but never terminate.

We continue by defining when other programs besides division are unsafe. We do not need to change the previous definition for when a program relates a prestate to a poststate, because it does not change (see Lecture 4, Figure 3 on page L4.11 for reference).

Assignment. Since expressions are never unsafe, assignments are never unsafe:

$$\omega[\![x := e]\!]\lightning \quad \text{never}$$

To make our semantic definitions more uniform, we will often state this equivalently as

$$\omega[\![x := e]\!]\lightning \quad \text{iff} \quad \text{false}$$

Sequential composition. $\alpha\ ;\ \beta$ is unsafe if either α is unsafe, or β is unsafe. In the latter case, we have to specify that the poststate of α is the prestate of β.

$$\begin{array}{lll} \omega[\![\alpha\ ;\ \beta]\!]\lightning & \text{iff} & \text{either } \omega[\![\alpha]\!]\lightning \\ & & \text{or } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\beta]\!]\lightning \text{ for some } \mu \end{array}$$

Conditional. $\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta$ is unsafe if α or β are unsafe, depending on P. Fortunately, we don't have to worry about P being unsafe: formulas are always either true or false.

$$\begin{array}{lll} \omega[\![\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]\!]\lightning & \text{iff} & \text{either } \omega \models P \text{ and } \omega[\![\alpha]\!]\lightning \\ & & \text{or } \omega \not\models P \text{ and } \omega[\![\beta]\!]\lightning \end{array}$$

While loop. $\textbf{while } P\ \alpha$ is unsafe if α is unsafe after any number of iterations. So we proceed as in the prior semantic definition, using an auxiliary form.

$$\begin{array}{lll} \omega[\![\textbf{while } P\ \alpha]\!]\lightning & \text{iff} & \omega[\![\textbf{while } P\ \alpha]\!]^n\lightning \text{ for some } n \in \mathbb{N} \\ \omega[\![\textbf{while } P\ \alpha]\!]^{n+1}\lightning & \text{iff} & \text{either } \omega \models P \text{ and } \omega[\![\alpha]\!]\lightning \\ & & \text{or } \omega \models P \text{ and } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\textbf{while } P\ \alpha]\!]^n\lightning \\ & & \quad \text{for some } \mu \\ \omega[\![\textbf{while } P\ \alpha]\!]^0\lightning & \text{iff} & \text{false} \end{array}$$

5 Reasoning about Safety

Our previous definition for the truth of $[\alpha]Q$ was the following:

$$\omega \models [\alpha]Q \quad \text{iff} \quad \text{for every } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ we have } \nu \models Q$$

This is a statement about *partial correctness* of α because if α does not terminate, *there is no such* ν so the statement is vacuously true.

With unsafe behavior we have a similar situation: An unsafe program has no poststate. If we leave the definition as is, then an unsafe program would satisfy every postcondition, which is clearly not desirable. So we modify our definition to add the condition that the program be *safe* (that is, not unsafe).

$$\omega \models [\alpha]Q \quad \text{iff} \quad \begin{array}{l}\text{for every } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ we have } \nu \models Q \\ \text{and } \textbf{not } \omega[\![\alpha]\!]\lightning\end{array}$$

The second part of this definition is how we solve that problem that in dynamic logic we only reason about the prestate and the poststate of the program. If it is unsafe, we should be not be able to prove the anything about the poststate. This includes the universally true proposition $\top$.

This means if we want to prove that a program α is safe given a precondition P we "just" need to prove

$$P \to [\alpha]\top$$

Note how this is different from general correctness where we have a postcondition Q. Of course, during the (formal) proof the formula above we may encounter other kinds of postconditions. Consider, for example, the case where α is $\alpha_1 \,;\, \alpha_2$. Or the case where safety of a division requires a loop invariant.

But how do unsafe programs come into the meaning of $[\alpha]Q$? We have to prevent such formulas to be provable when α is unsafe. For the division command we do this as follows.

$$\frac{\Gamma \vdash \neg(e_2 = 0), \Delta \quad \Gamma, x' = e_1/e_2 \vdash Q(x'), \Delta}{\Gamma \vdash [x := \textbf{divides } e_1\ e_2]Q(x), \Delta}\ [\textbf{divides}]R^{x'}$$

where x' must be chosen fresh (that is, it does not appear in e_1, e_2, $Q(x)$, Γ, or Δ).

Two important points about this rule:

1. We cannot apply this rule unless we can *prove* that $e_2 \neq 0$, that is, the division is safe.

2. The expression e_1/e_2 (denoting integer division) that we add to Γ is *indeterminate* in the manner explained in the introduction. So while it is technically an expression, we do not allow it in programs, only in formulas like $x' = e_1/e_2$. If we did allow the program to use it directly, our language would then have indeterminate results because the result of a/b is an indeterminate integer. While this could be allowed as long as we carefully distinguish between unsafe and indeterminate behavior, we avoid this complication here.

We do not prove the soundness or invertibility of this rule, but we will prove a related result later. As an axiom, by the way, the property of the **divide** program would be written as

$$[x := \textbf{divides } e_1\ e_2]Q(x) \leftrightarrow \forall x'.\ \neg(e_2 = 0) \wedge x' = e_1/e_2 \to Q(x')$$

A pleasant part of this approach is that the axioms and rules we derived so far can remain unchanged, essentially because they only rely on safe behavior. A premise involving a program will simply not be provable if its behavior is unsafe.

References

John Nolt. Free logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2021 edition edition, 2010. URL https://plato.stanford.edu/entries/logic-free/.