Lecture Notes on Proving Safety

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Lecture 5
September 10, 2024

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 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]). 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

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

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. Starting with our

new command, we have the following two clauses:

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

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

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.

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

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.

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

3 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).

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

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 in Section 6. As an axiom, by the way, the property of the **divide** program would be written as

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

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.

4 A Sample Proof of Safety

Proofs of safety can often be significantly simpler than proof of correctness. On the other hand, sometimes safety depends critically on some other correctness property.

We reconsider the example from Lecture 4.

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

To prove this, we required a loop invariant, and $x \geq 0$ was sufficient.

We can modify this to introduce a division, and just prove safety (so the postcondition is $\top$.

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

After one step ($\rightarrow R$) it remains to prove

$$x \geq 6 \vdash [\textbf{while } (x > 1)\ x := \textbf{divide } x\ 2]\, \top$$

Here, we pick the weakest loop invariant we can think of, namely $J = \top$. Then we have to prove:

True Initially: $x \geq 6 \vdash \top$, which is manifestly valid.

Preserved: We lose the antecedent $x \geq 6$, but we add the loop invariant and the loop guard. So we have to show

$$\top, x > 1 \vdash [x := \textbf{divide } x\ 2]\top$$

Using the rule $[\textbf{divide}]R$, this reduces to showing

$$\top, x > 1 \vdash \neg(2 = 0)$$

and

$$\top, x > 1, x' = x/2 \vdash \top$$

Both of these are manifestly valid.

Implies Postcondition: Again, without the antecedent, but this time with the negated loop guard, we have to prove the postcondition $\top$. So:

$$\top, \neg(x > 1) \vdash \top$$

Again, this is easily seen to be true.

So to prove safety, we only need the very weakest loop invariant in this example (which corresponds to having no significant loop invariant at all).

This would still be true for the following modified program:

$$x \geq 6 \vdash [\textbf{while } (x > 1) \ \{y := \textbf{divide } y \ x \ ; \ x := x - 2\}] \top$$

By the loop guard we see that $x > 1$ inside the loop body, so the division is safe. However, if we had written **divide** $x \ y$ then there is an immediate counterexample with $y = 0$.

5 A Generic Unsafe Command

In the example of division, unsafe behavior comes down to a particular operation. In general, though, it may be the combination of some operations that makes a program unsafe. For example, a program should not be able to write to an output stream after it has been closed. So it is not the output operation *per se*, but a condition associated with it. Or we may not be able to read from an input stream if we are not authorized to do so. We can capture such conditions more generically with the command

$$\textbf{assert } P$$

where P is a formula that may depend on variables. **assert** P is *unsafe* if P is false; otherwise it is safe but does not change the state.

$$\omega[\![\textbf{assert } P]\!]\nu \quad \text{iff} \quad \omega \models P \text{ and } \nu = \omega$$

$$\omega[\![\textbf{assert } P]\!]\lightning \quad \text{iff} \quad \omega \not\models P$$

It should be clear that we preserve the property that unsafe programs have no poststate.

Among other things, we could rewrite our programs using **assert** commands. For example, if we replaced $x := \textbf{divide } e_1 \ e_2$ by **assert** $\neg(e_2 = 0) \ ; \ x := e_1/e_2$ the two programs would have the same meaning in every state (either both unsafe, or both safe and determinate).

How do we reason about $[\textbf{assert } P]Q$? If P is true, then the postcondition Q must be true. If P is false, then Q is irrelevant: the formula $[\textbf{assert } P]Q$ is always false. These two conditions are neatly captured by the axiom

$$[\textbf{assert } P]Q \leftrightarrow P \wedge Q$$

Here are the corresponding right and left rules of the sequent calculus.

$$\frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash Q, \Delta}{\Gamma \vdash [\mathbf{assert}\ P]Q, \Delta}\ [\mathbf{assert}]R \qquad \frac{\Gamma, P, Q \vdash \Delta}{\Gamma, [\mathbf{assert}\ P]Q \vdash \Delta}\ [\mathbf{assert}]L$$

Let's prove that the axiom is actually valid. From that, the soundness of the rules as previously explained for the axiom $[;]A$ for sequential program composition.

Theorem 1 *The axiom*

$$[\mathbf{assert}\ P]Q \leftrightarrow P \wedge Q$$

is valid.

Proof: We start with the proof from right to left. We set up, for an arbitrary state ω:

$\omega \models P \wedge Q$ (assumption)
...
$\omega \models [\mathbf{assert}\ P]Q$ (to show)

In order to show the conclusion, we have to show two properties: (a) if $\omega[\![\mathbf{assert}\ P]\!]\nu$ then $\nu \models Q$, and (b) **not** $\omega[\![\mathbf{assert}\ P]\!]\lightning$.

(a) follows, since $\nu = \omega$ by definition of $[\![-]\!]$ and $\omega \models Q$ from our assumption.

(b) follows by definition of $\lightning$ since $\omega \models P$ from from our assumption.

For the proof from left to right, we set up

$\omega \models [\mathbf{assert}\ P]Q$ (assumption)
...
$\omega \models P \wedge Q$

By the definition of $\models$, the assumption gives us (a) for every ν with $\omega[\![\mathbf{assert}\ P]\!]\nu$ we have $\nu \models Q$, and (b) **not** $\omega[\![\mathbf{assert}\ P]\!]\lightning$.

From (b) we know $\omega \models P$ (otherwise $\mathbf{assert}\ P$ would be unsafe). Therefore, by definition of $[\![-]\!]$ and our assumption we have $\omega = \nu$, so $\omega \models Q$.

Taking these two together we have $\omega \models P \wedge Q$. □

6 A Theorem about Safety[1]

Theorem 2 (Soundness of Dynamic Logic with Unsafe Programs) *All the rules of the sequent calculus are sound, and all the axioms we stated are valid.*

Proof: By considering each case and reasoning along similar lines as in the sample proofs of such properties in lecture. □

[1] mentioned, but not explicitly stated in lecture

We can rigorously state that if we can prove *some* postcondition for α then α is safe. The theorem assumes that we have proved the soundness of all the sequent calculus rules (or axioms) we use in the formal proof (as claimed in the preceding theorem).

Theorem 3 (Safety) *If* $\cdot \vdash P \to [\alpha]Q$ *then there is no* ω *with* $\omega \models P$ *such that* $\omega[\![\alpha]\!]\lightning$.

Proof: Assume $\cdot \vdash P \to [\alpha]Q$ and for some ω we have $\omega \models P$ and $\omega[\![\alpha]\!]\lightning$. We have to show a contradiction.

By soundness of the sequent calculus we have $\omega \models P \to [\alpha]Q$. Since $\omega \models P$ we obtain $\omega \models [\alpha]Q$. By definition, this implies that **not** $\omega[\![\alpha]\!]\lightning$, which is a contradiction. □

References

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