Lecture Notes on
Formal Proof

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

Lecture 2
January 20, 2026

1 Introduction

We begin by talking about safety, and in particular one important class of safety properties: memory safety. Mistakes that programmers make having to do with memory safety have been responsible for a lot of security vulnerabilities over the years, and we will discuss some of the different types of memory safety errors that lead to vulnerabilities in common programs.

Our goal is to write code that does not have these vulnerabilities. There are many ways that have been proposed to avoid memory safety errors: type systems that ensure that memory is always used correctly, tools that check for memory safety errors at compile time, runtime systems that detect and prevent errors, and many others at various parts of the software and hardware stack. To understand how these work, and what their limitations and tradeoffs are, we need to understand how memory safety materializes in programs at the level of precise semantics.

In the second part of this lecture, we will go into detail to understand the semantics of programs and how they relate to proving things about how programs will behave when run. We won't yet define exactly what memory safety means at this level, but we will have laid the essential groundwork to do this, as well as to start reasoning about the safety of programs in more systematic ways.

2 Review: Program Semantics

In the previous lecture we described the smantics of a small imperative programming language. For the sake of simplicity we assumed that variables range of the integers $\mathbb{Z}$. Arithmetic expressions, denoted by placeholder e, were given by a simple grammar.

$$\text{Arithmetic Expressions} \quad e \ ::= \ c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \dots$$

Their meaning is determined with respect to a *state* (ω, μ, ν) that assigns integers to variables. We write $\omega(x) = c$ if ω maps x to c. Then we saw how to define the value of expression e in state ω, written as $\omega[\![e]\!] = c$, in a straightforward fashion.

$$\begin{array}{lcl}
\omega[\![c]\!] & = & c \\
\omega[\![x]\!] & = & \omega(x) \\
\omega[\![e_1 + e_2]\!] & = & \omega[\![e_1]\!] + \omega[\![e_2]\!] \\
\omega[\![e_1 - e_2]\!] & = & \omega[\![e_1]\!] - \omega[\![e_2]\!] \\
\ldots & &
\end{array}$$

The meaning of a program is a *relation* between the *prestate* and *poststate* of its execution, and we write

$$\omega[\![\alpha]\!]\nu$$

if the meaning of the program α relates prestate ω to poststate ν. With these conventions in place, defining the semantics of assignment, sequential composition, and conditional statements is a matter being unambiguous about which pre and post states are related.

$$\begin{array}{lll}
\omega[\![x := e]\!]\nu & \text{iff} & \omega[x \mapsto c] = \nu \text{ where } \omega[\![e]\!] = c \\
\omega[\![\alpha \,;\, \beta]\!]\nu & \text{iff} & \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\beta]\!]\nu \text{ for some state } \mu \\
\omega[\![\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]\!]\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\nu \quad \text{or} \\
 & & \omega \not\models P \text{ and } \omega[\![\beta]\!]\nu
\end{array}$$

Loops take a bit more care. There are several ways we might approach loops, because there are several possibilities about what can happen when going to execute a loop: the guard might be immediately false; or the body of the loop itself might not terminate; or the guard might never become false. Note that in all of these cases but the first, the loop doesn't terminate, so there is no post-state to relate to the pre state in question.

A concise way to navigate all this is to define a second relation between pre and post states, which is defined only for loops, and which is indexed on the number of iterations that the loop executes for.

$$\begin{array}{lll}
\omega[\![\textbf{while } P\ \alpha]\!]\nu & \text{iff} & \omega[\![\textbf{while } P\ \alpha]\!]^n\nu \quad \text{for some } n \in \mathbb{N} \\[1ex]
\omega[\![\textbf{while } P\ \alpha]\!]^{n+1}\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\textbf{while } P\ \alpha]\!]^n\nu \\
\omega[\![\textbf{while } P\ \alpha]\!]^0\nu & \text{iff} & \omega \not\models P \text{ and } \omega = \nu
\end{array}$$

Although we might not ourselves know for a given program what precise value of n will satisfy this definition, or even whether such an an n exists, it does not matter for our purposes here. What matters is that the definition matches our understanding of how an imperative loop should behave: if a loop terminates in some final state ν, then there must be some number of iterations that can reach ν from the initial state ω, with $n - 1$ intermediate states related by the body α.

3 A Program Proof

Now that we have precise semantics to work with, let's apply them towards proving something about how a particular program behaves. We'll prove that the program

$$x := x + y \; ; \; y := x - y \; ; \; x := x - y$$

swaps the values of x and y. We'll approach this by working backwards, supposing that the program ends in a final state where x and y are mapped to arbitrary constants, assuming nothing else. We'll show that given the semantics from Lecture 2, any initial state related to this final state must map x and y to these constants, but swapped.

So suppose we have an arbitrary initial state ω and an arbitrary final state ν such that

$$\omega[\![x := x + y \; ; \; y := x - y \; ; \; x := x - y]\!]\nu$$

and in the final state $\nu(x) = a$ and $\nu(y) = b$ for some arbitrary constants a and b. We will show that necessarily $\omega(x) = b$ and $\omega(y) = a$.

We set up the proof:

$\omega[\![x := x + y \; ; \; y := x - y \; ; \; x := x - y]\!]\nu$ (1, assumption)
$\nu(x) = a$ and $\nu(y) = b$ (2, assumption)
$\ldots$
$\omega(x) = b$ and $\omega(y) = a$ (to show)

By definition of sequential composition, assumption 1 tells us that there must be intermediate states μ_1 and μ_2 such that the three assignments relate the states in sequence.

$\omega[\![x := x + y]\!]\mu_1$ and $\mu_1[\![y := x - y \; ; \; x := x - y]\!]\nu$ (3, from 1 by defn. of $[\![-]\!]$)
$\mu_1[\![y := x - y]\!]\mu_2$ and $\mu_2[\![x := x - y]\!]\nu$ for some μ_2 (4, from 3(b) by defn. of $[\![-]\!]$)

Now we start working backwards from the last assignment. By definition of assignment, $\mu_2[\![x := x - y]\!]\nu$ means that ν is obtained from μ_2 by updating x to the value of $x - y$ evaluated in μ_2.

$\nu = \mu_2[x \mapsto c_3]$ where $c_3 = \mu_2[\![x - y]\!]$ (5, from 4(b) by defn. of $[\![-]\!]$)
$\mu_2(y) = \nu(y) = b$ (6, from 2, 5 by defn. of update)
$\nu(x) = \mu_2[\![x - y]\!]$ (7, from 5 by defn. of state update)
$\mu_2(x) - \mu_2(y) = a$ (8, from 2 and 7 by defn. of $[\![-]\!]$)
$\mu_2(x) = a + b$ (9, from 6 and 8)

Next we unwind the second assignment $y := x - y$. This relates μ_1 to μ_2 by updating only y, so x stays the same from μ_1 to μ_2.

$\mu_2 = \mu_1[y \mapsto c_2]$ where $c_2 = \mu_1[\![x - y]\!]$ (10, from 4(a) by defn. of $[\![-]\!]$)
$\mu_1(x) = \mu_2(x) = a + b$ (12, from 10 by defn. of state update)
$\mu_2(y) = \mu_1[\![x - y]\!] = \mu_1(x) - \mu_1(y) = b$ (13, from 12 by defn. of state update)
$\mu_1(y) = a$ (14, from 12 and 13)

Finally we unwind the first assignment $x := x + y$ to relate ω to μ_1. This assignment updates only x, so y stays the same from ω to μ_1.

$\mu_1 = \omega[x \mapsto c_1]$ where $c_1 = \omega[\![x + y]\!]$ (15, from 3(a) by defn. of $[\![-]\!]$)
$\omega(y) = \mu_1(y) = a$ (16, from 15 by defn. of state update)
$\mu_1(x) = \omega[\![x + y]\!] = \omega(x) + \omega(y) = a + b$ (17, from 15 by defn. of state update)
$\omega(x) = b$ (18, from 16 and 17)

We have now shown $\omega(x) = b$ and $\omega(y) = a$, as desired. Since a and b were arbitrary, we have shown that in every execution of this program the final value of x is the initial value of y and the final value of y is the initial value of x.

3.1 A Look Ahead

We've now seen how defining precise semantics for our programming language makes it possible to prove conclusive facts about what will happen when a program runs. Although the property that we proved was simple, following from the simplicity of the program, it's not difficult to see how we could apply a similar process to, for example, proving that the expression used to index into an allocated array is within its bounds; or that a pointer is not null prior to dereferencing it. The semantics of the language would be more complex, as would the program in question in nearly all cases, but the underlying approach would be very similar.

However, given the effort involved in the reasoning we just did, despite the simplicity of the language and program, it would quickly become daunting to imagine following through with this each and every time we needed to reason about safety. In the next lecture, we will develop a more streamlined and systematic approach for coming to conclusions about program safety. We will introduce a logic with primitives that let us express safety properties, and straightforward rules for manipulating formulas to make inferences. Eventually we'll see how the whole approach can in large part be automated, reducing the burden of demonstrating safety while at the same time removing the risk of human error in conducting the detailed, often tedious reasoning that is involved.

4 Summary

$$
\begin{array}{lcl}
\omega[\![c]\!] & = & c \\
\omega[\![x]\!] & = & \omega(x) \\
\omega[\![e_1 + e_2]\!] & = & \omega[\![e_1]\!] + \omega[\![e_2]\!] \\
\omega[\![e_1 * e_2]\!] & = & \omega[\![e_1]\!] \times \omega[\![e_2]\!]
\end{array}
$$

Figure 1: Semantics of Expressions

$$
\begin{array}{lll}
\omega[\![x := e]\!]\nu & \text{iff} & \omega[x \mapsto c] = \nu \text{ where } \omega[\![e]\!] = c \\
\omega[\![\alpha \,;\, \beta]\!]\nu & \text{iff} & \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\beta]\!]\nu \text{ for some state } \mu \\
\omega[\![\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]\!]\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\nu \quad \text{or} \\
 & & \omega \not\models P \text{ and } \omega[\![\beta]\!]\nu \\
\omega[\![\textbf{while } P\ \alpha]\!]\nu & \text{iff} & \omega[\![\textbf{while } P\ \alpha]\!]^n\nu \quad \text{for some } n \in \mathbb{N} \\
\omega[\![\textbf{while } P\ \alpha]\!]^{n+1}\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\textbf{while } P\ \alpha]\!]^n\nu \\
\omega[\![\textbf{while } P\ \alpha]\!]^0\nu & \text{iff} & \omega \not\models P \text{ and } \omega = \nu
\end{array}
$$

Figure 2: Semantics of Programs

$$
\begin{array}{lllll}
\omega \models e_1 \leq e_2 & \text{iff} & \omega[\![e_1]\!] \leq \omega[\![e_2]\!] & & \\
\omega \models e_1 = e_2 & \text{iff} & \omega[\![e_1]\!] = \omega[\![e_2]\!] & & \\
\omega \models P \wedge Q & \text{iff} & \omega \models P & \text{and} & \omega \models Q \\
\omega \models P \vee Q & \text{iff} & \omega \models P & \text{or} & \omega \models Q \\
\omega \models P \rightarrow Q & \text{iff} & \omega \models P & \text{implies} & \omega \models Q \\
\omega \models \neg P & \text{iff} & \omega \not\models P & & \\
\omega \models P \leftrightarrow Q & \text{iff} & \omega \models P & \text{iff} & \omega \models Q
\end{array}
$$

Figure 3: Semantics of Formulas