Lecture Notes on
Program Analysis

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

Lecture 9
September 24, 2024

1 Introduction

A major theme of this course is how rules of inference and other formal objects like the axioms of dynamic logic bridge the gap between mathematical definitions and implementation. One can also think of them as an interface: on one side we can rigorously prove mathematical properties (like: soundness of the rules) and on the other side we can translate them into programs and run them. The following table illustrates this point of view

semantic property	**formal system**	**algorithm**
validity	axioms $[\alpha]Q \leftrightarrow P$	weakest precondition $\mathsf{wlp}\ \alpha\ Q$
soundness	rules $\Gamma \vdash \Delta$	symbolic evaluation $\Gamma \Vdash [\alpha]S$

From a pragmatic perspective, it remains to translate the algorithm into code. We have already formulated weakest liberal precondition as a function, but at a fairly high level of abstraction. Concrete decision need to be made and depend to an extent on the implementation language you choose. Lab 1 will give you the opportunity to explore that. We recommend a statically type functional language because the abstractions it provides are most suitable for such implementation tasks.

The specific implementations we ask in Lab 1 are an *interpreter* and a *verification condition generator*. The first executes programs, while the second calculates the weakest liberal precondition and passes it to a theorem prover to verify soundness (if possible). We provide you with the most "boring" parts, mainly a parser from the *concrete syntax* of a program (a string of characters) into the *abstract syntax* (representing the mathematical structure of programs).

We have already talked extensively about the weakest liberal precondition, so today we transition from symbolic evaluation to actual evaluation of a program. This will raise an issue we will then address by an additional program analysis algorithm.

2 Evaluation

In Lecture 8 we formalized symbolic evaluation as a collection of rules of inference for $\Gamma \Vdash [\alpha]S$ where Γ is a collection of antecedents in pure arithmetic, α is a program where all conditions are also in pure arithmetic, and S represents a stack of programs followed by a postcondition Q in pure arithmetic. These rules can be read as an algorithm, applying them in a bottom-up fashion.

For evaluation, we have a stronger assumption, namely that every variable has a value so we can just look it up in the state that maps variable to values. But then how do we evaluate a program $y := x + 1$? If this program is all we have, then there is a problem. Namely: we don't have an initial value for x! While it is mathematically convenient to just assume that a state ω is a total map from variables to integer values, when actually running programs this may not be the case. For this lecture, therefore, when describing the algorithm for evaluation, we assume that the state is a *partial map* from variables to integers. We still use the same letters, like ω, μ, and ν. This partial map must be *defined* on all variables that a program may *use*, leading to the so-called *def/use* analysis of programs. We postpone that to Section 3, first showing evaluation.

We write

$$\mathsf{eval}\ \omega\ \alpha = \nu$$

where ω is the initial state, α is the program, and ν is the final state if it exists. We also need to evaluate expressions to return an integer, and conditions to return a Boolean.

$$\begin{array}{lcl} \mathsf{eval}_{\mathbb{Z}}\ \omega\ e & = & c \in \mathbb{Z} \\ \mathsf{eval}_{\mathbb{B}}\ \omega\ P & = & b \in \mathbb{B} \end{array}$$

Except for **while** loops, the rules for evaluation are faithful transcriptions of the semantic definition of programs as denoting a relation.

We start with expressions. We need to ensure that $\omega(x)$ is defined by our def/use analysis; here we just assume that it is.

$$\begin{array}{lcll} \mathsf{eval}_{\mathbb{Z}}\ \omega\ c & = & c & \\ \mathsf{eval}_{\mathbb{Z}}\ \omega\ x & = & \omega(x) & \text{(must be defined)} \\ \mathsf{eval}_{\mathbb{Z}}\ \omega\ (e_1 + e_2) & = & \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_1 + \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_2 & \end{array}$$

Note that for addition (and other operations we have elided), the "plus" on the left-hand side is a program construct and the "plus" on the right-hand side is the mathematical operation of addition on integers.

Boolean conditions (as they appear in if-then-else, loops, assertions, and tests) are evaluated in a similar manner.

$$\begin{array}{lcl} \mathsf{eval}_{\mathbb{B}}\ \omega\ (e_1 \leq e_2) & = & \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_1 \leq \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_2 \\ \mathsf{eval}_{\mathbb{B}}\ \omega\ (\top) & = & \top \\ \mathsf{eval}_{\mathbb{B}}\ \omega\ (\bot) & = & \bot \\ \mathsf{eval}_{\mathbb{B}}\ \omega\ (P \wedge Q) & = & \mathsf{eval}_{\mathbb{B}}\ \omega\ P \wedge \mathsf{eval}_{\mathbb{B}}\ \omega\ Q \end{array}$$

In the case of conjunction, the conjunction on right-hand side implements the truth table for conjunction. As we have emphasized multiple times, expressions and Boolean conditions are safe and always terminate. Because of that, the following definition suggested in lecture is semantically equivalent but more efficient.

$$\begin{array}{rcll} \mathsf{eval}_{\mathbb{B}}\ \omega\ (P \wedge Q) & = & \mathsf{eval}_{\mathbb{B}}\ \omega\ Q & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \top \\ & = & \bot & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \bot \end{array}$$

We elide the straightforward remaining cases. On to programs!

Sequential Composition. In order to evaluate $\alpha\ ;\ \beta$ we evaluate β in the state resulting from the evaluation of α.

$$\mathsf{eval}\ \omega\ (\alpha\ ;\ \beta) \quad = \quad \mathsf{eval}\ (\mathsf{eval}\ \omega\ \alpha)\ \beta$$

There is the possibility that $\mathsf{eval}\ \omega\ \alpha$ does not have a poststate (e.g., a loop is infinite or a test fails). In your programming language you will have to account for this somehow, for example by raising an exception if a test fails, or returning either some token indicating failure or that poststate and then distingushing the cases. Since this depends on your programming language and your intended interface to the implementation, we won't specify this here.

Assignment. For assignment, we update the state ω.

$$\mathsf{eval}\ \omega\ (x := e) \quad = \quad \omega[x \mapsto c] \quad \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e = c$$

Conditionals.

$$\begin{array}{rcll} \mathsf{eval}\ \omega\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) & = & \mathsf{eval}\ \omega\ \alpha & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \top \\ & = & \mathsf{eval}\ \omega\ \beta & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \bot \end{array}$$

Tests and Assertions. Assertions in programs are there to prove safety statically, so we don't execute them. Test are there to guarantee safety dynamically, so we perform them and "abort" if necessary.

$$\begin{array}{lll} \mathsf{eval}\ \omega\ (\mathbf{assert}\ P) & = \ \ \omega & \\ \mathsf{eval}\ \omega\ (\mathbf{test}\ P) & = \ \ \omega & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \top \\ \mathsf{eval}\ \omega\ (\mathbf{test}\ P) & \text{has no poststate} & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \bot \end{array}$$

The action of "abort" is not explicitly represented—it depends on your implementation language and environment.

Loops. For loops, the semantic specification and the implementation diverge. Instead of "guessing" how many times we go around the loop, but we just proceed recursively.

$$\begin{array}{lcll} \mathsf{eval}\ \omega\ (\mathbf{while}\ P\ \alpha) & = & \omega & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \bot \\ & = & \mathsf{eval}\ (\mathsf{eval}\ \omega\ \alpha)\ (\mathbf{while}\ P\ \alpha) & \text{provided } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \top \end{array}$$

The interpreter may itself not terminate if the loop does not terminate, or the implementation may carry a bound on the number of evaluation steps before giving up.

3 Def/Use Analysis

Before starting the program, we'd like to make sure that all variables the program may use are actually defined by the time their value is needed. We could probably translate this kind of problem into a proposition of dynamic logic and reason about it logically. But the "defined-before-use" property is fundamental to evaluation, so we want to check it before we ever attempt to execute program. This check should not require a theorem prover, but reflect a simple algorithm that is easy for the programmer to understand.

We define two functions returning finite sets on programs. Since expressions and formulas use variables but do not define them, we only have **use** for them.

$\mathsf{use}\ \alpha$	the set of variables *used* by α
$\mathsf{def}\ \alpha$	the set of variables *defined* by α
$\mathsf{use}_{\mathbb{Z}}\ e$	the set of variables *used* by e
$\mathbf{use}_{\mathbb{B}}\ P$	the set of variables *used* by P

We don't give a full definition (leaving this to you in Lab 1), but we show a few cases.

Expressions and Formulas.

$$\begin{array}{lcl} \mathsf{use}_{\mathbb{Z}}\ c & = & \{\,\} \\ \mathsf{use}_{\mathbb{Z}}\ x & = & \{x\} \\ \mathsf{use}_{\mathbb{Z}}\ (e_1 + e_2) & = & \mathsf{use}_{\mathbb{Z}}\ e_1 \cup \mathsf{use}_{\mathbb{Z}}\ e_2 \\ \mathbf{use}_{\mathbb{B}}\ (\top) & = & \{\,\} \\ \mathbf{use}_{\mathbb{B}}\ (P \wedge Q) & = & \mathbf{use}_{\mathbb{B}}\ P \cup \mathbf{use}_{\mathbb{B}}\ Q \end{array}$$

Assignment. Assignment is the base case for definition.

$$\begin{array}{lcl} \mathsf{use}\ (x := e) & = & \mathsf{use}_{\mathbb{Z}}\ e \\ \mathsf{def}\ (x := e) & = & \{x\} \end{array}$$

Conditionals. Here, we are reminded that "use" has to be interpreted as "*may use*" while "def" means "*must define*". Keeping this in mind:

$$\begin{aligned}\mathsf{use}\ (\textbf{if}\ P\ \textbf{then}\ \alpha\ \textbf{else}\ \beta) &= \textbf{use}_{\mathbb{B}}\ P \cup \mathsf{use}\ \alpha \cup \mathsf{use}\ \beta \\ \mathsf{def}\ (\textbf{if}\ P\ \textbf{then}\ \alpha\ \textbf{else}\ \beta) &= \mathsf{def}\ \alpha \cap \mathsf{def}\ \beta\end{aligned}$$

Note the intersection in the last clause: a conditional is only guaranteed to define a variable if both branches define it. This is the case independently of the condition. So

$$\textbf{def}\ (\textbf{if}\ \top\ \textbf{then}\ x := 1\ \textbf{else}\ \textbf{skip})$$

does not include x. Therefore, a program such as

$$(\textbf{if}\ \top\ \textbf{then}\ x := 1\ \textbf{else}\ \textbf{skip})\ ;\ y := x$$

would be rejected, claiming that x may not be defined before its use.

Sequential Composition and Loops. We'll leave it to you to work these out. Especially sequential composition may take a little thought, because that's where def and use interact nontrivially.

Note that for Lab 1 we specified that loop invariants are not to be checked (like asserts).

4 Generating a Verification Condition

Lecture 7 already explains the function to compute the weakest liberal precondition in some detail. We only review a detail we didn't elaborate on: how do we perform substitution? So if we write $Q(x)$, how do we calculate $Q(e)$ for an expression e.

Because for $\mathsf{wlp}\ \alpha\ Q$ we only substitute into pure formulas Q and not into programs, it is actually fairly simple. Because even pure formulas contain expressions we also need to substitute into expressions. So we define

$$\begin{aligned}\mathsf{subst}_{\mathbb{Z}}\ e\ x\ e' &= e'' & &\text{substitute } e \text{ for } x \text{ in } e' \\ \mathsf{subst}_{\mathbb{B}}\ e\ x\ Q &= Q' & &\text{substitute } e \text{ for } x \text{ in } Q\end{aligned}$$

We can relate the second one to our prior notation: $\mathsf{subst}_{\mathbb{B}}\ e\ x\ Q(x) = Q(e)$. Here are some straightforward cases:

$$\begin{aligned}\mathsf{subst}_{\mathbb{Z}}\ e\ x\ c &= c \\ \mathsf{subst}_{\mathbb{Z}}\ e\ x\ x &= e \\ \mathsf{subst}_{\mathbb{Z}}\ e\ x\ y &= y & &\text{provided } x \neq y \\ \mathsf{subst}_{\mathbb{Z}}\ e\ x\ (e_1 + e_2) &= (\mathsf{subst}_{\mathbb{Z}}\ e\ x\ e_1) + (\mathsf{subst}_{\mathbb{Z}}\ e\ x\ e_2)\end{aligned}$$

In the last case, the "$+$" on both sides constructs an expression, because unlike evaluation, substitution returns a general expression and not necessarily a constant. The same is true for "$\leq$" and "$\wedge$" below that construct formulas.

Except for the last case shown below, substitution into a formula is similarly straightforward.

$$\begin{array}{lcl}
\mathsf{subst}_{\mathbb{B}}\ e\ x\ \top & = & \top \\
\mathsf{subst}_{\mathbb{B}}\ e\ x\ \bot & = & \bot \\
\mathsf{subst}_{\mathbb{B}}\ e\ x\ (e_1 \leq e_2) & = & (\mathsf{subst}_{\mathbb{Z}}\ e\ x\ e_1) \leq (\mathsf{subst}_{\mathbb{Z}}\ e\ x\ e_2) \\
\mathsf{subst}_{\mathbb{B}}\ e\ x\ (P \wedge Q) & = & (\mathsf{subst}_{\mathbb{Z}}\ e\ x\ P) \wedge (\mathsf{subst}_{\mathbb{Z}}\ e\ x\ Q) \\
\mathsf{subst}_{\mathbb{B}}\ e\ x\ (\Box P) & = & \Box P
\end{array}$$

Why don't we substitute into a white box? Recall that $\Box P$ is true if P is valid, that is, it is true regardless of the state we are in. We needed to introduce it because variables whose value we know before entering a loop are unknown inside the loop body and after the loop. So occurrences of x inside $\Box P$ should be considered "fresh" variables, implicitly universally quantified.

Whether this turns out to be significant for your implementation depends on how and when you pass formulas to the theorem prover. We specified in Lab 1 that all loop invariants should be checked (even if they occur in "unreachable" code). So a correct strategy is **not** to actually form the conjunction

$$\begin{array}{lcl}
\mathsf{wlp}\ (\mathbf{while}_J\ P\ \alpha)\ Q & = & J \\
 & & \wedge\ \Box(J \wedge P \to \mathsf{wlp}\ \alpha\ J) \\
 & & \wedge\ \Box(J \wedge \neg P \to Q)
\end{array}$$

on the right hand side, but separately test each white-boxed conjunct for validity and just return J as the weakest precondition if both pass.