Lecture Notes on Symbolic Evaluation

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

Lecture 8
September 19, 2024

1 Introduction

In the last lecture we introduced the *weakest liberal precondition* which can be calculated algorithmically from a program as long as loop invariants are provided by the programmer. We can then prove $P \to [\alpha]Q$ by delegating $P \to \mathsf{wlp}\ \alpha\ Q$ to a theorem prover for arithmetic, as long as P, Q, and any formulas in α are formulas of pure arithmetic. The algorithm was based on the *axioms* for dynamic logic we developed and proved semantically sound. Variations of this algorithm are used by systems for program verification such as Why3 or Dafny. In this course we are mostly interested in verifying safety, which benefits from the same techniques. Furthermore, functional verification and safety are often inseparably intertwined.

There is a counterpart to the weakest precondition, namely the *strongest postcondition*. This can also be represented in dynamic logic [Streett, 1982, Platzer, 2004]; you can find a summary in Lecture 11 of 15-414 *Bug Catching: Automated Program Verification*. Here, we approach it slightly differently, instead devising an algorithm for proving safety by traversing the program in the order of evaluation. This is just the opposite of the weakest precondition which proceeds through the program in reverse order. This time, instead of using the axioms, we take our inspiration from the rules of the sequent calculus. This results in *symbolic evaluation*, and actual program execution can be seen as a special case. It is often packaged in the form of *bounded model checking* [Biere et al., 2003] and available in tools such as CMBC.

As we will see, there are advantages and disadvantages to both approaches, which is why both have applications in industry.

2 Analysis in Evaluation Order

At the outset, we make the same restriction as in the last lecture: in $P \to [\alpha]Q$, both P and Q are pure, and all formulas occurring in α are also pure. Unfortunately, if we want to analyze the program in the order it is evaluated, this is not quite sufficient. Consider (for now) the axiom for sequential composition of programs:

$$[\alpha \mathrel{;} \beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

Even if we start with a pure postcondition Q, on the right-hand side where we focus on α, the postcondition is suddenly $[\beta]Q$. When you think about it, this makes sense: if we execute a command in a program, we somehow have to remember what else needs to be done. It turns out that the programs that remain to be executed form a *stack*. We write S for such formulas and conjecture that they are sufficient to specify symbolic evaluation. (Turns out we are right.)

$$\text{Stacks} \quad S \quad ::= \quad Q \mid [\alpha]S$$

Revisiting the axiom using stacks:

$$[\alpha \mathrel{;} \beta]S \leftrightarrow [\alpha]([\beta]S)$$

This is now well-formed, because if S is a stack on the left-hand side, the $[\beta]S$ is a stack on the right-hand side. We'll have to keep an eye on it, though.

Next, we consider a specification $P \to [\alpha]S$ in the sequent calculus. We start the derivation:

$$\frac{P \vdash [\alpha]S}{\cdot \vdash P \to [\alpha]S} \to\!R$$

It looks like the succedent consists of a single formula $[\alpha]S$, while the antecedent is also a single (pure) formula P. In order to handle conditionals, we need to generalize the antecedent. Consider

$$\frac{P, P' \vdash [\alpha]S \quad P, \neg P' \vdash [\beta]S}{P \vdash [\mathbf{if}\ P'\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]S} [\mathbf{if}]R$$

We see that we accumulate information in the antecedents, so we generalize from a single formula to a collection Γ consisting entirely of pure formulas.

In summary, we will try to define procedure for proving sequents of the form

$$\underbrace{\Gamma}_{\text{all pure}} \vdash \underbrace{S}_{\text{stack}}$$

3 Inference Rules Defining Algorithms

We already saw one instance where a collection of inference rules described an algorithm: the sequent rules for propositional calculus in Lecture 2. The algorithm was as follows:

1. Starting from the sequent we are trying to prove, we arbitrarily use rules bottom-up. Since all rules are invertible (we preserve validity) and reductive (we make progress), this is a sound and complete strategy.

2. When we arrive at sequents with only propositional variables we have two cases:

   (a) If antecedents and succedents share a variable p, we use the identity rule and this subgoal has been proved successfully.

   (b) If they are disjoint, we can construct a countermodel (contradicting validity) by making all antecedents true and all succedents false.

We now want to use a similar strategy to construct a derivation of $\Gamma \vdash S$, under the restrictions motivated in the previous section. It will look roughly as follows.

1. If S is a pure formula Q, we call an oracle to prove the purely arithmetic sequent.

2. Otherwise, $S = [\alpha]S'$ for some S'. We use the appropriate rule for decomposing the program α. Each premise becomes a subgoal.

The rules for $[\alpha]S'$ are *reductive* in the sense that they only contain constituent programs of α in their premises. Therefore the procedure will terminate if each application of the arithmetic oracle terminates.

For completeness we also need invertibility. As usual, this holds except for loops. However, if we require the programmer to specify loop invariants then a form of completeness does hold. We then say that the algorithm is *complete relative to an oracle for arithmetic*.

Because the form of sequents are restricted compared to the general case of dynamic logic, we introduce a new notation:

$$\Gamma \Vdash S$$

We will ascertain the property that $\Gamma \Vdash S$ if and only if $\Gamma \vdash S$, which guarantees the soundness and (relative) completeness of our algorithm.

4 Rules for Symbolic Evaluation

The first is a general (and straightforward) rule: when the succedent is pure, we call the oracle. We express this as a sequent consisting entirely of pure formulas.

$$\dfrac{\Gamma \vdash Q \quad Q \text{ pure}}{\Gamma \Vdash Q}\;\mathsf{arith}$$

Like several other rules, the rule for composition is just a restriction of the rule of the ordinary sequent calculus.

$$\dfrac{\Gamma \Vdash [\alpha]([\beta]S)}{\Gamma \Vdash [\alpha \,;\, \beta]S}\;[;]R$$

Similarly, assignment remains the same.

$$\dfrac{\Gamma, x' = e \Vdash S(x') \quad x' \text{ fresh}}{\Gamma \Vdash [x := e]S(x)}\;[{:=}]R^{x'}$$

Unlike the situation for the calculation of weakest preconditions, the stack $S(x)$ may contain programs. We therefore can only substitute $S(e)$ in some special cases—generally, we need to make a new assumption and generate a fresh instance of x.

Even for conditionals, not much changes.

$$\dfrac{\Gamma, P \Vdash [\alpha]S \quad \Gamma, \neg P \Vdash [\beta]S}{\Gamma \Vdash [\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]S}\;[\mathbf{if}]R$$

We can notice something interesting here: we accumulate more information in the antecedents as we proceed with the (symbolic) evaluation. This is not the case when calculating the weakest precondition (or at least not in a straightforward manner).

Let's consider an example with conditionals:

$$\begin{array}{lcl} \alpha_1 & = & (\mathbf{if}\ x \geq 0\ \mathbf{then}\ y := x\ \mathbf{else}\ y := -x) \\ \alpha_2 & = & (\mathbf{if}\ x \geq 0\ \mathbf{then}\ z := -x\ \mathbf{else}\ z := x) \\ \alpha & = & (\alpha_1 \,;\, \alpha_2) \end{array}$$

We claim that after this program terminates we have $y + z = 0$.

Let's see how this works out with symbolic evaluation.

$$\dfrac{\dfrac{\overset{(1)}{x \geq 0, y' = x \Vdash [\alpha_2]\,(y' + z = 0)}}{x \geq 0 \Vdash [y := x]([\alpha_2]\,(y + z = 0))}\;[{:=}]R \qquad \overset{(2)}{\neg(x \geq 0) \Vdash [y := -x]([\alpha_2]\,(y + z = 0))}}{\dfrac{\cdot \Vdash [\alpha_1]([\alpha_2]\,(y + z = 0))}{\cdot \Vdash [\alpha_1 \,;\, \alpha_2]\,(y + z = 0)}\;[;]R}\;[\mathbf{if}]R$$

Proceeding to (1), we have to apply the rule for conditionals again.

$$\dfrac{\dfrac{\dfrac{\overset{\checkmark}{x \geq 0, y' = x, x \geq 0, z' = -x \vdash y' + z' = 0}}{x \geq 0, y' = x, x \geq 0, z' = -x \Vdash y' + z' = 0}\;\mathsf{arith}}{x \geq 0, y' = x, x \geq 0 \Vdash [z := -x]\,(y' + z = 0)}\;[{:=}]R \qquad \dfrac{\overset{?}{x \geq 0, y' = x, \neg(x \geq 0), z' = x \Vdash y' + z' = 0}}{x \geq 0, y' = x, \neg(x \geq 0) \Vdash [z := x]\,(y' + z = 0)}\;[{:=}]R}{x \geq 0, y' = x \Vdash [\alpha_2]\,(y' + z = 0)}\;[\mathbf{if}]R$$

In the first branch, everything goes according to play. But in the second branch, we can read off something like $y' + z' = 2x$ but this is not equal to 0! Fortunately, we can still prove the sequent because the assumptions $x \geq 0$ and $\neg(x \geq 0)$ are contradictory.

In fact, we could have stopped just before, when the succedent still contained a program because the antecedents are already contradictory! The following rule allows us to to this in general: we can stop our proof construction as soon as the antecedents become inconsistent.

$$\dfrac{\Gamma \vdash \bot}{\Gamma \Vdash S}\;\mathsf{infeasible}$$

Since the premise is a pure arithmetic sequent, we directly appeal to the oracle. The savings of this rule can be considerable, because the stack S could contain a complex program. We will return to this in Section 6 after completing our remaining rules.

We skip the subgoal (2) since it can be proved in a symmetric manner.

5 Assertions, Tests and Loops

Assertions and tests are entirely straightforward, since we just repurpose the ordinary sequent rules.

$$\dfrac{\Gamma \vdash P \quad \Gamma \Vdash S}{\Gamma \Vdash [\mathbf{assert}\ P]S}\;[\mathbf{assert}]R \qquad\qquad \dfrac{\Gamma, P \Vdash S}{\Gamma \Vdash [\mathbf{test}\ P]S}\;[\mathbf{test}]R$$

Just note that our assumptions about purity and the stack structure of the succedents are satisfied. Also, the first premise immediately appeals to the oracle since P must be pure.

For loops with invariants, again we mimic the ordinary sequent rule.

$$\dfrac{\Gamma \vdash J \quad J, P \Vdash [\alpha]J \quad J, \neg P \Vdash S}{\Gamma \Vdash [\mathbf{while}_J\ P\ \alpha]S}\;[\mathbf{while}]R$$

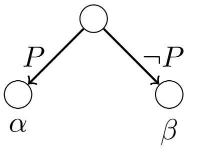
The first premise is pure, but the other two are not by contain stack formulas, so we remain within the algorithmic rules ($\Vdash$).

An interesting point here is the symbolic evaluation in the presence of loop invariants is actually **not** a special case of evaluation: we analyze the loop body only once, rather than possibly many times as we do when a program is executed. This, and the fact that we often won't have loop invariants, leads to the idea of *bounded symbolic evaluation* or *bounded model checking*.

6 Control Flow Graphs

A pictorial representation of imperative programs, in particular for thinking about program analysis, are *control flow graphs*. Since they are also suitable for low level programs (for example, in assembly language), they are particularly common in compiler design.

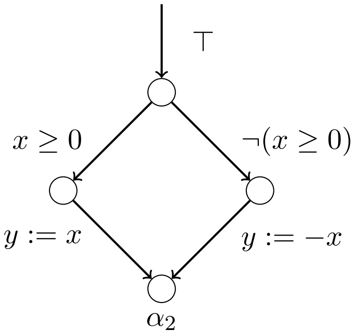
For our purposes, we use an especially simple form. Small circles denote points in the program, and arrows indicate possible transitions from one point in the program to the next. On the side of the arrow we indicate the *information gained* for symbolic evaluation along this particular transition. For example, for a conditional $\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta$ we would draw



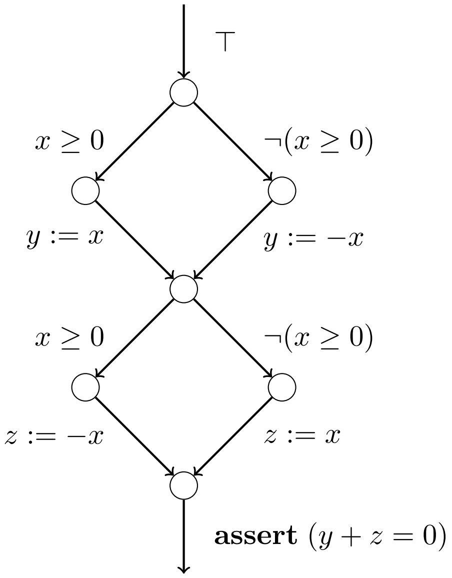
A precondition P is drawn as a label on the incoming edge to the root. Starting our example,

$$\begin{array}{lcl}\alpha_1 & = & (\textbf{if } x \geq 0 \textbf{ then } y := x \textbf{ else } y := -x) \\ \alpha_2 & = & (\textbf{if } x \geq 0 \textbf{ then } z := -x \textbf{ else } z := x) \\ \alpha & = & (\alpha_1 \mathrel{;} \alpha_2)\end{array}$$

The precondition is $\top$ and then we have a conditional with two branches. They come back together before α_2.



For an assignment, we just label the arrow with the assignment, although the "information gained" will be an equality on a renamed variable. The program α_2 is represented by a similar graph, starting at the bottom node.



A *path* through this program just follows the arrows from the root down to the final node. A priori, each path represents a potential execution of the program. It is easy to see that the number of paths through a control flow graph could be exponential in its size.

Viewed in terms of the sequent calculus, each path represents a branch in the proof tree, looking upwards. A *path formula* is the conjunction of the information gained along a path (which includes some renaming for variable assignments). The output **assert** represents the postcondition in the succedent of the sequent. For example, going left both times gives us the sequent

$$x \geq 0, y' = x, x \geq 0, z' = -x \vdash y' + z' = 0$$

If we first go left and then right we can stop even before the assignment because the path reads

$$x \geq 0, y' = x, \neg(x \geq 0) \vdash \ldots$$

which is *infeasible*. In this example, there are only two feasible paths, and the postcondition holds for both of them.

7 Bounded Symbolic Evaluation

When there are no loop invariants (or maybe they aren't sufficient for our purposes), symbolic evaluation offers another option. We can *unroll each loop* to a certain specified depth. The depth is necessary in many cases in order to avoid nontermination of the algorithm. Recall the axiom

$$[\mathbf{while}\ P\ \alpha]Q \leftrightarrow (P \to [\alpha]([\mathbf{while}\ P\ \alpha]Q)) \wedge (\neg P \to Q)$$

and the corresponding rule:

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

In this rule we don't lose Γ and Δ because we go around the loop exactly 1 or 0 times. The unfold rule is not reductive. In order to represent *bounded* evaluation we annotate the **while** with n, a constant natural number not accessible to the programmer. Instead, it would usually be a parameter to an invocation of a bounded model checker. Then we have two rules: one when we are still allowed to unroll the loop, and one when we have reached the bound.

$$\frac{\Gamma, P \Vdash [\alpha]([\mathbf{while}^n\ P\ \alpha]S) \quad \Gamma, \neg P \Vdash S}{\Gamma \Vdash [\mathbf{while}^{n+1}\ P\ \alpha]S}\ \mathsf{unfold}^{n+1} \qquad \frac{\Gamma \Vdash S}{\Gamma \Vdash [\mathbf{while}^{0}\ P\ \alpha]S}\ \mathsf{unfold}^{0}$$

The rules are now reductive in the sense that the pair (α, n) decreases: either n becomes smaller and the program remains the same, or n has reached 0 and then the program becomes smaller. This is called a *lexicographic ordering* because two pairs are compared first in their first component and then in their second if the first are equal. This is like the ordering of words in a dictionary.

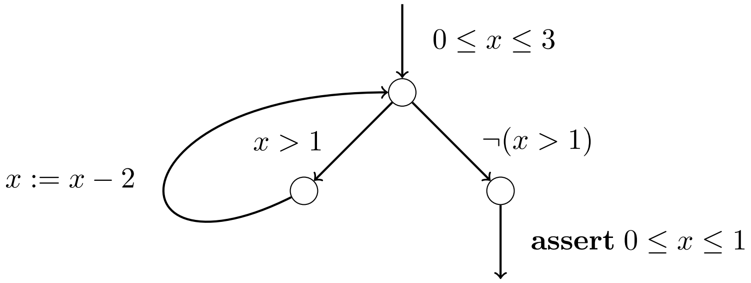
These rules have several problems. A critical one is that we may not be guaranteed that the loop without the depth annotation actually satisfies the postcondition. If we can prove all subgoals we know at least that there isn't an obvious problem that would arise by limited execution. And if there is a bug, we might still find it by generating a subgoal that is not provable. Therefore we might say that bounded symbolic evaluation is for *bug finding*, but generally not for full verification.

In some circumstances, even bounded checking could amount to full verification. That's when the paths around the loop become infeasible before the bound is reached. Sequent derivations are large and awkward to show, so we use a control flow graph instead for the (by now familiar) program

$$0 \le x \le 3 \to [\mathbf{while}\ (x > 1); x := x - 2]\ (0 \le x \le 1)$$

with a new precondition.

Because we have a loop, the control flow graph will now have a back edge, pointing up higher in the graph.



We enumerate the sequents along the paths, which are marked with R (for chosing the right alternative, leaving the loop) and L for chosing the left one (proceeding into the loop).

R	:	$0 \leq x \leq 3, \neg(x > 1) \vdash 0 \leq x \leq 1$	valid!
LR	:	$0 \leq x \leq 3, x > 1, x' = x - 2, \neg(x' > 1) \vdash 0 \leq x' \leq 1$	valid!
LL	:	$0 \leq x \leq 3, x > 1, x' = x - 2, x' > 1 \vdash \ldots$	infeasible!

The last path doesn't go all the way to the end, because the partial path LL is already contradictory. The path being infeasible means that the sequent is valid using the rule infeasible. So in this example the precondition was strong enough that we were able to prove validity with just two iterations of the loop.

8 Summary

We summarize the rules for symbolic evaluation, alternatively with loop invariants or bounds on loop unrolling, in Figure 1. In order to prove $P \to [\alpha]Q$ for pure P and Q (and α only containing pure conditions), we search bottom-up for a proof of $P \Vdash [\alpha]Q$. Also recall the definition of stacks

$$\text{Stacks} \quad S \quad ::= \quad Q \mid [\alpha]S$$

By the way, we can recover ordinary execution from symbolic evaluation by proceeding as in bounded evaluation (unrolling the loop), without regard to any bound. This only makes sense if our antecedents assign a constant value to each variable that is used by the program. In that case, each time we might branch due to a conditional or loop, one side will immediately be infeasible and we proceed deterministically. Of course, evaluation may not terminate.

This is not a particularly clever way to evaluate a program because of the frequent renaming. Essentially, the antecedents keep track of the whole history of the computation, that is, every value that a variable had on the current (and only) path. One could improve on that, but there are also more direct ways to obtain evaluation from the semantic definition of $\omega[\![\alpha]\!]\nu$.

$$\frac{\Gamma \vdash Q \quad Q \text{ pure}}{\Gamma \Vdash Q}\ \mathsf{arith} \qquad \frac{\Gamma \vdash \bot}{\Gamma \Vdash S}\ \mathsf{infeasible}$$

$$\frac{\Gamma \Vdash [\alpha]([\beta]S)}{\Gamma \Vdash [\alpha \,;\, \beta]S}\ [;]R \qquad \frac{\Gamma, x' = e \Vdash S(x') \quad x' \text{ fresh}}{\Gamma \Vdash [x := e]S(x)}\ [:=]R^{x'}$$

$$\frac{\Gamma, P \Vdash [\alpha]S \quad \Gamma, \neg P \Vdash [\beta]S}{\Gamma \Vdash [\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]S}\ [\mathbf{if}]R$$

$$\frac{\Gamma \vdash P \quad \Gamma \Vdash S}{\Gamma \Vdash [\mathbf{assert}\ P]S}\ [\mathbf{assert}]R \qquad \frac{\Gamma, P \Vdash S}{\Gamma \Vdash [\mathbf{test}\ P]S}\ [\mathbf{test}]R$$

$$\frac{\Gamma \vdash J \quad J, P \Vdash [\alpha]J \quad J, \neg P \Vdash S}{\Gamma \Vdash [\mathbf{while}_J\ P\ \alpha]S}\ [\mathbf{while}]R$$

$$\frac{\Gamma, P \Vdash [\alpha]([\mathbf{while}^n\ P\ \alpha]S) \quad \Gamma, \neg P \Vdash S}{\Gamma \Vdash [\mathbf{while}^{n+1}\ P\ \alpha]S}\ \mathsf{unfold}^{n+1} \qquad \frac{\Gamma \Vdash S}{\Gamma \Vdash [\mathbf{while}^0\ P\ \alpha]S}\ \mathsf{unfold}^0$$

Figure 1: Symbolic Evaluation

References

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

André Platzer. Using a program verification calculus for constructing specifications from implementations. Minor Thesis (Studienarbeit), University of Karlsruhe, Department of Computer Science, February 2004. URL https://lfcps.org/logic/Minoranthe.html.

Robert S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54:121–141, 1982.