Lecture Notes on Dynamic Logic

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

Lecture 3
September 3, 2024

1 Introduction

In the last lecture we introduced *propositional sequent calculus* because it is the foundation of most of the calculi we will investigate and because it helps us understand the basic principles underlying the sequent calculus. Let's summarize them:

- We define *sequents* $\Gamma \vdash \Delta$ with antecedents (assumptions) and succedents (goals).
- We give a *meaning* to sequents (also called a *semantics*): a sequent is *valid* if when all antecedents are true then some succedent is true.
- We also give *inference rules* to prove sequents formally in a bottom-up manner. These are divided into *right rules* (how to prove a succedent) and *left rules* (how to use an antecedent), and the identity rule connecting left and right.
- We connect the inference rules to the semantics by proving (mathematically, at the metalevel) that the sequent calculus is *sound* and *complete*:
  - A rule is sound if the validity of all premises implies the validity of the conclusion. If all rules are sound, the whole system of inference rules is sound.
  - A system of rules is *complete* if every valid sequent can be proved with them.

In the specific case of propositional sequent calculus we were able to prove completeness using the following steps:

- Every rule is *invertible* in the sense that if the conclusion is valid then all the premise must also be valid.

- Every rule is *reductive* in the sense that all premises are smaller than the conclusion by counting the number of connectives and logical constants.

As we move forward, we will have to give up some of these properties while retaining others.

2 Countermodels

We begin this lecture with a brief discussion of another part of the interface between semantics and proof rules. Let's say we have a sequent $\Gamma \vdash \Delta$ that is **not valid**. Clearly, by soundness, we should not be able to derive it. But is there some other information we may wish to obtain? If $\Gamma \vdash \Delta$ is not valid that there should be some way to assign truth values to the propositional variables so that Γ is true but Δ is false. This assignment of truth values corresponds to a *counterexample* to the validity of $\Gamma \vdash \Delta$, and we say it defines a *countermodel* (where a model is given by an assignment of truth values to propositional variables). Having a countermodel is useful if we want to debug our formulas (or later our programs) because it may express something we have overlooked.

So how do we construct a countermodel? Remember that all of the rules are reductive and invertible, and that we have a full set of rules for the connectives on the left or on the right of the turnstile. This means we can always reduce any sequent we wish to prove to leaves of the form

$$p_1, \ldots, p_n \vdash q_1, \ldots, q_m$$

If one of the p_i equals one of the q_j then we close of this branch in the proof by using the rule of identity. If not, then we can construct a countermodel by setting all p_i to true and all q_j to false. Every unprovable leaf will give us a countermodel, although some of them may coincide.

Say we conjecture (somewhat rashly) that $(p \wedge q) \vee (\neg p \wedge \neg q)$, which states that p and q have the same truth value. At attempt to prove this might look as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,\mathsf{id}}{\cdot \vdash p, \neg p}\,\neg R \quad \dfrac{\dfrac{XXX}{q \vdash p}}{\cdot \vdash p, \neg q}\,\neg R}{\cdot \vdash p, \neg p \wedge \neg q}\,\wedge R \quad \dfrac{\dfrac{\dfrac{XXX}{p \vdash q}}{\cdot \vdash q, \neg p}\,\neg R \quad \dfrac{\dfrac{}{q \vdash q}\,\mathsf{id}}{\cdot \vdash q, \neg q}\,\neg R}{\cdot \vdash q, \neg p \wedge \neg q}\,\wedge R}{\cdot \vdash p \wedge q, \neg p \wedge \neg q}\,\wedge R}{\cdot \vdash (p \wedge q) \vee (\neg p \wedge \neg q)}\,\vee R$$

We can close off two leaves with the identity rule, but we also see that there are two branches where the sequent cannot be derived. The first one, $q \vdash p$ gives us a countermodel where $q = \top$ and $p = \bot$. The second one, $p \vdash q$ gives us another

countermodel where $p = \top$ and $q = \bot$. These are exactly the situations where p and q have different truth values.

Some terminology that will come in useful when using tools, reading papers, or for other courses such as 15-311 *Logic and Mechanized Reasoning*.

Unsatisfiable. We call a proposition F *unsatisfiable* if $F \vdash \cdot$ is valid. Because there are no succedents, this can only be valid if there is no assignment of truth values to variables in F to make it true. Equivalently, we could define that F is unsatisfiable if $\cdot \vdash \neg F$ is valid (that is, F is always false).

Satisfiable. If $F \vdash \cdot$ is not unsatisfiable we call it *satisfiable*. A *satisfying assignment* is a way to assign truth values to the propositional variables in F to make F true. We also say that a satisfying assignment is a *model*.

One way to prove the validity of a formula F is to show that $\neg F$ is unsatisfiable. This follows easily semantically, or by proof rules because $\neg F \vdash \cdot$ is valid if and only if $\cdot \vdash F$ is valid by the (sound and invertible) rule $\neg L$.

3 Safety and Liveness

Our goal in this course is to reason about security properties of programs so we can prevent vulnerabilities and fend off attacks. There are different kinds of such properties, and they require different techniques to enforce them. One way to classify them is to think about them as properties of *traces* of a program, that is, the (possibly infinite) sequence of states or events that take place when a program executes.

Safety Properties Intuitively, a safety property means that "*nothing bad happens*" during a computation. So every finite prefix of a trace should satisfy some specification that excludes "bad" states or events. Common examples of "bad" are programs that, in C, have undefined behavior. This includes division by zero, integer overflow, double free, or accessing memory whose value is undefined. The latter is exploited in so-called *buffer overflow attacks*. An example from concurrency are race conditions between threads. Another example is a policy that requires that a principal is authorized before giving them access to a resource, in which case the "bad" thing is unauthorized access.

Liveness Properties Intuitively, a liveness property captures that "*something good happens*" during an execution. For example, a server should eventually respond to a request, or a deleted file should actually disappear and be no longer recoverable.

In the early part of this course we will mostly focus on safety properties. Liveness properties are more intrinsically more difficult to reason about and enforce

than safety properties. For more on various kinds of security properties and their enforcements, see Schneider's seminal paper [2000].

There are also security properties that can not be captured as properties of a single trace. We will consider such a property, namely *information flow*, in the second part of the course.

4 Dynamic Logic: A Logic with Programs

In this course we use a tiny imperative programming language so we can be rigorous about the concepts we introduce. With it we illustrate and analyze the concepts that transfer to realistic languages. There will be small differences regarding the precise extent of the language as we move through various concepts. Here is our first cut. *Expressions* denote integers and are either constants, variables, or operators like addition and multiplication. Programs include variable assignment, sequential composition, conditionals, and loops. Formulas no longer have propositional variables (for simplicity), but we add comparisons between integers to the usual set of logical constants and connectives. New here are two kinds of formulas, $[\alpha]Q$ and $\langle\alpha\rangle Q$ that mention programs. We explain them below the table.

Variables	x, y, z		
Constants	c	$::=$	$\ldots, -1, 0, 1, \ldots$
Expressions	e	$::=$	$c \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid \ldots$
Programs	α, β	$::=$	$x := e \mid \alpha \,;\, \beta \mid \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \mid \textbf{while } P\ \alpha$
Formulas	P, Q	$::=$	$e_1 \leq e_2 \mid e_1 = e_2 \mid \ldots$
		$\mid$	$P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \neg P \mid \top \mid \bot$
		$\mid$	$[\alpha]Q \mid \langle\alpha\rangle Q$

A *state* is a total map from variables to integer values. We use ω, μ, ν for states. A program represents a partial function from an initial state (called *prestate*) to a final state (called *poststate*). It is a partial function because loops may not terminate, so no final state may every be reached. The sequence of states that a program goes through is its *trace* as discussed above.

Characteristic of *dynamic logic* [Harel, 1979, Harel et al., 2000] are two modalities that mention programs:

- $[\alpha]Q$ (pronounced "*box alpha Q*") which is true if, starting in a prestate ω, the formula Q will be true in every poststate ν we can reach by executing program α.
- $\langle\alpha\rangle Q$ (pronounced "*diamond alpha Q*") which is true in a state ω if there is a poststate ν that we can reach by executing α in which Q is true.

We refer to Q in these formulas as a *postcondition*. These definitions are formulated to account for *nondeterministic* programs that may have multiple poststates for a

given prestate. In our case of a deterministic language, this will be zero or one. Therefore, we can already see that $\langle\alpha\rangle Q$ should imply $[\alpha]Q$.

In the next lecture we introduce a rigorous *semantics* for dynamic logic (which, by necessity, also includes programs and expressions). For today, we instead try to derive some rules keeping in mind the informal semantics and our understanding how an imperative programming language executes.

5 Conditionals and Assignments

We start with a very simple program to assign the absolute value of x to y:

$$\textbf{if } x < 0 \textbf{ then } y := -x \textbf{ else } y := x$$

As is often the case for security properties we are not interested in a full specification of the behavior of this code. Instead, we try to analyze *ranges* of values. For example, it would be safe to subsequently try to take an integer square root of y because y will always be nonnegative. We express this property of the program and its poststate as a *formula* in dynamic logic:

$$[\textbf{if } x < 0 \textbf{ then } y := -x \textbf{ else } y := x]\, y \geq 0$$

The informal reason here is clear: if $x < 0$ then we know $y = -x \geq 0$ in the poststate. If $x \geq 0$ then $y = x \geq 0$.

To make this formal, we now extend our earlier sequent calculus with rules for $[\alpha]Q$. For now, we are only concerned with right rules. The goal is to reduce properties for compound programs (like conditionals) to properties of smaller programs (like its branches). Here is an attempt:

> *Assume we are in a prestate ω. We want to show that Q holds in every poststate of "$\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta$". We can get to a poststate in two ways: if P is true then by executing α, and if P is false then by executing β.*

Translating this reasoning into a sequent calculus rule yields:

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

There are two premises, one for the **then** branch and one for the **else** branch. For the **then** branch we are allowed to assume P, while for the **else** branch we are allowed to assume $\neg P$.

For assignments $x := e$ one might at first think we should be allowed to assume that $x = e$ in the poststate.

$$\frac{\Gamma, x = e \vdash Q, \Delta}{\Gamma \vdash [x := e]Q, \Delta}\ [:=]R? \qquad (unsound!)$$

Before reading on, you might want to think about why this is unsound.

Consider $x = 2 \vdash [x := 1]\, x = 3$. With the rules above we could reason

$$\frac{x = 2, x = 1 \vdash x = 3}{x = 2 \vdash [x := 1]\, x = 3}\ [:=]R?$$

The premise here is actually valid because the antecedents are contradictory. What went wrong is that at the purely logical level we are confusing the value of x in two different states: before and after the execution of the assignment. In lecture we used the program $x := x + 1$ which doesn't even require a precondition to obtain a contradiction.

In order to avoid this paradox, we track the value of x after the assignment in a *fresh* variable x'. By *fresh* here we mean it doesn't appear in the conclusion sequent at all. We just need to make sure that the postcondition now talks about x' instead of x. We write $Q(x)$ for the formula $Q(x)$ which may mention x, and $Q(x')$ for the result of substituting x' for *all* occurrences of x in $Q(x)$. Then the rule becomes:

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

The superscript on R expresses that x' must be "fresh" and cannot appear in the conclusion of the rule. That is, it cannot be in Γ, e, $Q(x)$ or Δ. In some cases, we can then eliminate the antecedent $x' = e$ by substituting e for x' in $Q(x')$. This can considerably simplify proofs but isn't always possible. We come back to this in the next lecture.

Let's return to our motivating example to see if we can prove it now.

$$\frac{\dfrac{x < 0, y' = -x \vdash y' \geq 0}{x < 0 \vdash [y := -x]\, y \geq 0}\ [:=]R^{y'} \quad \dfrac{\neg(x < 0), y' = x \vdash y' \geq 0}{\neg(x < 0) \vdash [y := x]\, y \geq 0}\ [:=]R^{y'}}{\cdot \vdash [\textbf{if}\ x < 0\ \textbf{then}\ y := -x\ \textbf{else}\ y := x]\, y \geq 0}\ [\textbf{if}]R$$

The unproved leaves here are valid formulas of integer arithmetic. Rather than defining somewhat tedious proof rules for them, we just imagine that they can be proved by an oracle. In an implementation, we would use a theorem prover like Z3 [Moura and Børner, 2008] or cvc5 [Barbosa et al., 2022] (which are actually be decision procedures on nontrivial fragments of arithmetic). This is somewhat similar to the way we handled sequents consisting only of propositional variables, except in that case it was very easy to see if the identity rule applied or a countermodel could be constructed.

6 Sequential Composition

How do we handle the formula $[\alpha\ ;\ \beta]Q$? Intuitively, we run the program α, starting in some prestate ω reaching some poststate μ and the run β starting in μ. Then

Q must be true in every poststate of β. The way to express this is to require that $[\beta]Q$ be true after α. In the form of a rule:

$$\frac{\Gamma \vdash [\alpha]([\beta]Q), \Delta}{\Gamma \vdash [\alpha\ ;\ \beta]Q, \Delta}\ [;]R$$

The good news is that, once again, we have broken down the properties of the program $(\alpha\ ;\ \beta)$ into properties of α and β.

Let's use this rule to prove a property of the following program that combines assignment and sequential composition

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

This program *swaps* the values of x and y without using a new temporary variable. (But be careful: in a language like C where integers have limited range, this will often have undefined behavior!)

How can we express this? We couldn't just state $x = y \land y = x$ as a postcondition. Instead we "remember" the initial values of x and y. So the verification will consist of a proof of the implication

$$\underbrace{x = a \land y = b}_{\text{precondition}} \rightarrow [x := x + y\ ;\ y := x - y\ ;\ x := x - y]\ \underbrace{(x = b \land y = a)}_{\text{postcondition}}$$

In 15-122 *Principles of Imperative Computation* we used `//@requires` for preconditions and `//@ensures` for postconditions, limited to functions. Here, they are not part of the program but part of a logical formula and can enclose any program.

We now build a proof of this using our rules. This construction proceeds bottom-up, but we only show the final resulting derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x = a, y = b, x' = x + y, y' = x' - y, x'' = x' - y' \vdash x'' = b \land y' = a}{x = a, y = b, x' = x + y, y' = x' - y \vdash [x' := x' - y']\ (x' = b \land y' = a)}\ [:=]R^{x''}}{x = a, y = b, x' = x + y \vdash [y := x' - y]([x' := x' - y]\ (x' = b \land y = a))}\ [:=]R^{y'}}{x = a, y = b, x' = x + y \vdash [y := x' - y\ ;\ x' := x' - y]\ (x' = b \land y = a)}\ [;]R}{x = a, y = b \vdash [x := x + y]([y := x - y\ ;\ x := x - y]\ (x = b \land y = a))}\ [:=]R^{x'}}{x = a, y = b \vdash [x := x + y\ ;\ y := x - y\ ;\ x := x - y]\ (x = b \land y = a)}\ [;]R}{x = a \land y = b \vdash [x := x + y\ ;\ y := x - y\ ;\ x := x - y]\ (x = b \land y = a)}\ \land L}{\cdot \vdash x = a \land y = b \rightarrow [x := x + y\ ;\ y := x - y\ ;\ x := x - y]\ (x = b \land y = a)}\ \rightarrow R$$

The leaf here is a sequent of pure arithmetic. Now we can carry out some substitution, like a for x and b for y and we get

$$x' = a + b, y' = x' - b, x'' = x' - y' \vdash x'' = b \land y' = a$$

and then

$$x' = a + b, y' = a, x'' = b \vdash x'' = b \land y' = a$$

and we recognize it as valid.

We also see that explicit, step-by-step reasoning in the sequent calculus is quite tedious and better left to a machine. Fortunately, it is not difficult to mechanize using some tools that researchers have built over the years.

7 Rule Summary So Far

Our rules so far for $[\alpha]Q$ have the good properties we come to expect: they are sound (argued only informally), they are invertible (not even considered yet), and they are reductive (by reducing the program to its components or eliminating it altogether).

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

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

$$\frac{\Gamma \vdash [\alpha]([\beta]Q), \Delta}{\Gamma \vdash [\alpha\ ;\ \beta]Q, \Delta}\ [;]R$$

Figure 1: Some Rules for Dynamic Logic

In the next lecture we will look at **while** loops and also provide a semantics so we prove the soundness and invertibility of some of the rules.

References

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, pages 415–442, Munich, Germany, April 2022. Springer LNCS 13243.

David Harel. *First-Order Dynamic Logic*. Springer LNCS 68, 1979. 136 pp.

David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000. 476 pp.

Leonardo De Moura and Nikolaj Børner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340, Budapest, Hungary, mar 2008. Springer LNCS 4963.

Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information Systems Security*, 3(1):30–50, February 2000.