Lecture Notes on Propositional Logic and Proof

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

Lecture 2
August 29, 2024

1 Introduction

In this course, we repeatedly define security policies and investigate how they can be *provably* enforced on programs. Such enforcement could be *static* in the sense that we verify that a given program will satisfy our security policy before we ever run it. Or it could be *dynamic* in the sense that we abort the program if it ever tries to perform an action that would violate our security policy. Or we could use a combination of such techniques. This requires us to formalize (a) the security policy, (b) our programming language and how it executes, and (c) the enforcement mechanism. We further have to *prove* that the enforcement mechanism works as intended.

Our not-so-secret weapon will be formal systems of deduction. They are used in multiple roles: they can serve as definitions, they can serve as the basis for implementations, and they can serve as the focal point of proof.

In this first lecture we exemplify some these roles in one of the simplest possible settings: the Boolean logic of propositions. Propositional connectives such as conjunction, disjunction, implication, negation, etc. are at the very heart of logical reasoning, so the insights and techniques from this lecture will be a useful guide to future lectures. There are no programs yet—we'll add them in the next lecture.

2 Propositional Formulas

Propositional formulas $F, G, H, \ldots$ are constructed from propositional variables $p, q, \ldots$ by forming conjunctions, disjunctions, implications, negations, and possibly others like bi-implication. There are also constants $\top$ (for *true*) and $\bot$ (for *false*). This is expressed in the following so-called Backus-Naur Form (BNF).

$$\text{Formulas} \quad F, G, H \quad ::= \quad p \mid F \wedge G \mid F \vee G \mid F \to G \mid \neg F \mid \top \mid \bot$$

Variables can take the value $\top$ (true) or $\bot$ (false), and the meaning of the other connectives are given by their truth tables. For example, $F \wedge G$ is true if both F and G are true, and false otherwise. $F \to G$ is true either F is false or G is true. We also refer to propositional variables as *atomic proposition* because they cannot be further decomposed. In the next lecture we will add other atomic propositions besides variables.

We say a formula F is *valid* if it is true no matter which truth values we assign to the variables contained in them. A formula F is *satisfiable* if there is some way to assign truth values to the variables so that the resulting formula is true.

3 Simple Sequents

We now want to devise a system of inference rules so we can *formally prove* that a given proposition is valid. It should be designed so that if someone doubts the validity of a proposition you can convince them by showing them the proof. Each step in the proof should be correct, and each step should be easy to check.

The design of systems of inference will occupy a lot of our time, and it is far from straightforward. There are many choices, and the same concepts may admit many different formalization, suitable for different purposes. For example, we could have very few inference rules (in propositional logic often just one!) and many axioms, usually called a Hilbert-style system. Or we could have many rules and essentially no axioms. For this lecture we choose Gentzen's *sequent calculus* [Gentzen, 1935]. It has several useful properties which we will come back to starting in Section 8. One of them is that many logics permit formulations as sequent calculi because they are constructed very systematically.

A *simple sequent* has the form $F_1, \ldots, F_n \vdash G$ where $F_1, \ldots, F_n$ are the *antecedents* and G is the *succedent*. The symbol "$\vdash$" separating them is called a *turnstile* and usually pronounced "*entails*". A sequent formalizes the state of a proof where we have the *assumptions* $F_1, \ldots, F_n$ and try to prove the *goal* G. In fact, most proof assistants like Coq or Lean will present the state of a missing proof in the form of a sequent because it reflects the way that mathematicians think about the state of an (incomplete) proof.

We often abbreviate the collection of antecedents as Γ (capital *Gamma*). Also, their order is irrelevant, so we consider, for example, p, q to be the same as q, p.

More formally, we say a sequent $\Gamma \vdash G$ is *valid* if G is true whenever all propositions in Γ are true. We often write this out as "all Γ true implies G true".

4 Right and Left Rules

A pleasant property of the sequent calculus is that the connectives are defined in isolation from each other. This makes it easy to consider subsets of the connectives

or extensions with additional connectives because in many cases we don't have to change any of the existing rules.

The rules comes in two flavors. The *right rules* define how to decompose the consequent, that is, the goal proposition on the right-hand side of the turnstile. Conversely, the *left rules* define how to decompose an antecedent, that is, an assumption on the left-hand side of the turnstile.

Let's start with conjunction. How do we prove $F \wedge G$ from some assumptions Γ? Easy: we prove both F and G separately from the assumptions. Formulated as a rule:

$$\frac{\Gamma \vdash F \quad \Gamma \vdash G}{\Gamma \vdash F \wedge G}\ \wedge R$$

We write (and read) this rule bottom-up. Reading upwards, we see that we eliminate the conjunction from the given formulas.

The sequent below the line is called the *conclusion*, while the two sequents above the line are the *premises*. The name of the rule, $\wedge R$ (read: "and-right"), is written to the right of the line.

We also have to consider how we *use an assumption* $F \wedge G$. The insight is that if we know $F \wedge G$ is true, then F and G must both be true. As a rule (again, read it from the bottom up):

$$\frac{\Gamma, F, G \vdash H}{\Gamma, F \wedge G \vdash H}\ \wedge L$$

For this rule we implicitly apply our convention that the order of the antecedents does not matter. So $F \wedge G$ doesn't actually have to be the last antecedent, it just has to be one of them.

We are almost ready to do our first formal proof, that is, derivation. But we have to be able to complete a partial proof and declare victory. Recall that $\Gamma \vdash F$ is valid (by definition) if whenever all formulas in Γ are true then F is true. This justifies the following *rule of identity*:

$$\frac{}{\Gamma, F \vdash F}\ \mathsf{id}$$

It may look strange at first because it is a rule with no premises. As we construct our sequent calculus proofs bottom-up, applications of the identity rule sit at the top (which are the leaves, if we think of the proof as a tree).

Let's try to prove the sequent $p \wedge q \vdash q \wedge p$. It should be intuitively clear that this is valid, so we should be able to prove it. We construct the derivation bottom-up, starting with

$$\begin{array}{c} \vdots \\ p \wedge q \vdash q \wedge p \end{array}$$

There are two options: we could apply either the right rule of the left rule for conjunction. Let's use the left rule. Our as yet incomplete derivation now looks

like this:

$$\dfrac{\begin{array}{c}\vdots\\ p, q \vdash q \wedge p\end{array}}{p \wedge q \vdash q \wedge p}\ \wedge L$$

At this point we can only apply the right rule for conjunction.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ p, q \vdash q\end{array} \quad \begin{array}{c}\vdots\\ p, q \vdash p\end{array}}{p, q \vdash q \wedge p}\ \wedge R}{p \wedge q \vdash q \wedge p}\ \wedge L$$

We recognize both unproved goals as instances of the identity rule and finish our derivation.

$$\dfrac{\dfrac{\dfrac{}{p, q \vdash q}\ \mathsf{id} \quad \dfrac{}{p, q \vdash p}\ \mathsf{id}}{p, q \vdash q \wedge p}\ \wedge R}{p \wedge q \vdash q \wedge p}\ \wedge L$$

5 Implication

We move on to implication $F \to G$. How do we prove an implication? We assume F and then prove G under this additional assumption.

$$\dfrac{\Gamma, F \vdash G}{\Gamma \vdash F \to G}\ \to R$$

Remember to read the rule bottom-up.

In order to understand the left rule, we have to think about how we use an assumption $F \to G$. Here is a first attempt: if we also know F we are permitted to use G.

$$\dfrac{\Gamma, F, G \vdash H}{\Gamma, F, F \to G \vdash H}\ \to L?$$

While this rule is sound (see Section 9) it is insufficient. Here is an example:

$$\dfrac{\begin{array}{c}\vdots\\ (p \to p) \to q \vdash q\end{array}}{}$$

Is this sequent valid? This is easy to check by a truth table argument. Assume that $(p \to p) \to q$. We have to show that q is true. Since $p \to p$ is always true, so we know that q must also be true.

But in the system so far, no rule can be applied! Since the succedent is a variable and the only antecedent is an implication, the only rule that could apply is a left rule for implication. But it doesn't, since we do not have $p \to p$ as one of our antecedents.

The fix here is not at all obvious. We reinterpret uses of an implication as follows: if we know $F \to G$ and we can prove F then we know G. By "know" here we mean that it is one of our assumptions. Writing this out in the form of a sequent rule:

$$\frac{\Gamma \vdash F \quad \Gamma, G \vdash H}{\Gamma, F \to G \vdash H} \to L$$

As noted by a student in lecture, there seems to be a "temporal" dependency: we only have license to assume G if we have first proved F. But during bottom-up proof construction we could also proceed with the second premise first. That is, we could "check" (by proving) that G entails H and only if that's the case, do we bother to (try to) prove F.

Now we can prove the problematic example from above. As usual, we proceed bottom up, but we only show the final derivation.

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}\,\mathsf{id}}{\cdot \vdash p \to p} \to R \quad \dfrac{}{q \vdash q}\,\mathsf{id}}{(p \to p) \to q \vdash q} \to L$$

An excellent question from lecture: "*What is the difference between $F \to G$ and $F \vdash G$?*" Indeed, $F \to G$ is valid as a formula (that is, always true) if and only if $F \vdash G$ is valid as a sequent. Let's think about how you might go about building a prover for the validity of formulas. You are given a *formula* $F \to G$ and you want to construct a derivation of the sequent $\cdot \vdash F \to G$. Here we use "$\cdot$" to emphasize that there is an empty collection of antecedents. We break this down and now to have to build a derivation of $F \vdash G$, where you have a singleton list F as an antecedent and G as a conclusion. So $F \vdash G$ is the result of decomposing the implication, now exposing the top-level logical connectives of F and G for further (bottom-up) application of inference rules. In contrast, $F \to G$ is just a single thing: a formula that's an implication.

6 Disjunction

Disjunction throws another wrench into the works that's not as easy to repair as for implication. From the truth table definition we know that $F \vee G$ is true if either F is true or G is true. This time, we start with the left rule because it is easier. When have an assumption $F \vee G$ we can proceed in a proof by distinguishing two cases,

proving our goal H in both of them. That is:

$$\frac{\Gamma, F \vdash H \quad \Gamma, G \vdash H}{\Gamma, F \vee G \vdash H} \vee L$$

From the same truth table definition, we can also conjecture that there should be *two* right rules.

$$\frac{\Gamma \vdash F}{\Gamma \vdash F \vee G} \vee R_1? \qquad \frac{\Gamma \vdash G}{\Gamma \vdash F \vee G} \vee R_2?$$

It is clear that during proof construction we now have to make a choice, which also means we may have to backtrack over this choice (consider $q \vdash p \vee q$).

What's worse, there are now some valid sequents that we cannot prove! If you weren't in lecture, or you don't remember, it is quite a brainteaser to find something valid you cannot prove. Think about it before moving on to the next page.

Consider $p \vee (p \to q)$. This is valid: if p is true then the left disjunct is true. If p is false, then the right disjunct $p \to q$ is true. Either way, the disjunction is true. But with the two right rules for disjunction, we cannot prove it. There are only two possible attempts.

$$\dfrac{\dfrac{\text{XXX}}{\cdot \vdash p}}{\cdot \vdash p \vee (p \to q)} \vee R_1 \qquad \dfrac{\dfrac{\dfrac{\text{XXX}}{p \vdash q}}{\cdot \vdash p \to q} \to R}{\cdot \vdash p \vee (p \to q)} \vee R_2$$

The first is not valid because the variable p could be false. The second is not valid because p could be true and q could be false.

Interestingly, even though it is incomplete for the Boolean interpretation of the formulas using two truth values, the sequent rules we have shown so far can be developed further into *intuitionistic logic*. In intuitionistic logic, proofs correspond to (functional) programs, and propositions correspond to their types. For example, an intuitionistic proof of $F \to G$ is a function of type $F \to G$ that takes a proof of F into a proof of G. You can learn more about this in *15-317 Constructive Logic* where we develop a logic and a functional programming language together in a single system.

In this course, starting in the next lecture, we instead define a small imperative programming language Tinyscript and then reason about it externally using a logic where every formula is either true or false.

7 Sequents with Multiple Succedents

Gentzen's solution to the problem with disjunction is quite clever and not obvious because it doesn't reflect the way we do mathematics. The idea is to allow sequents to have multiple succedents. We define

$$(F_1, \ldots, F_n \vdash G_1, \ldots, G_m) \quad \text{is valid}$$

if whenever all F_i are true then at least one G_j is true. Put it another way, the sequent $F_1, \ldots F_n \vdash G_1, \ldots, G_m$ is valid if and only if the formula $(F_1 \wedge \cdots \wedge F_n) \to (G_1 \vee \cdots \vee G_m)$ is valid. We generally abbreviate antecedents by Γ and succedents by Δ (capital *Delta*).

Having multiple succedents that are interpreted *disjunctively* allows us to hedge our bets with a better right rule.

$$\dfrac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \vee G, \Delta} \vee R$$

$$\frac{}{\Gamma, F \vdash F, \Delta}\ \mathsf{id}$$

$$\frac{\Gamma \vdash F, \Delta \quad \Gamma \vdash G, \Delta}{\Gamma \vdash F \wedge G, \Delta}\ \wedge R \qquad \frac{\Gamma, F, G \vdash \Delta}{\Gamma, F \wedge G \vdash \Delta}\ \wedge L$$

$$\frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \to G, \Delta}\ \to R \qquad \frac{\Gamma \vdash F, \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \to G \vdash \Delta}\ \to L$$

$$\frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \vee G, \Delta}\ \vee R \qquad \frac{\Gamma, F \vdash \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \vee G \vdash \Delta}\ \vee L$$

$$\frac{\Gamma, F \vdash \Delta}{\Gamma \vdash \neg F, \Delta}\ \neg R \qquad \frac{\Gamma \vdash F, \Delta}{\Gamma, \neg F \vdash \Delta}\ \neg L$$

Figure 1: Sequent Calculus with Multiple Succedents

The price we pay is that we now have to generalize all of our other rules to allow multiple conclusions. We summarize them in Figure 1. We have added the rules for negation, to come in Section 13.

Let's revisit our earlier problem to make sure we can now derive $p \vee (p \to q)$.

$$\frac{\begin{array}{c}\vdots\\ \cdot \vdash p, p \to q\end{array}}{\cdot \vdash p \vee (p \to q)}\ \vee R$$

At this point we can only apply one rule, the right rule for implication.

$$\frac{\dfrac{\begin{array}{c}\vdots\\ p \vdash p, q\end{array}}{\cdot \vdash p, p \to q}\ \to R}{\cdot \vdash p \vee (p \to q)}\ \vee R$$

The unproved leaf is just an instance of the identity rule and we are done.

$$\frac{\dfrac{\dfrac{}{p \vdash p, q}\ \mathsf{id}}{\cdot \vdash p, p \to q}\ \to R}{\cdot \vdash p \vee (p \to q)}\ \vee R$$

8 Properties of Inference Systems

Question: are we done? We discovered some flaws in earlier rules, so how can we know that they are all fixed? Well, we can appeal to authority (that is, Gentzen) and consider our job done. But that is unsatisfactory because we often have to think about variations (like: introducing programs and reasoning about them) and then the right inference system may not be immediately available. Also, by actually mathematically proving that our systems "works" we gain some insights that can be exploited not only for other, closely related systems but also for an implementation.

Two fundamental properties of inference system such as the one in Figure 1 are *soundness* and *completeness*. For sequents, they are:

Soundness: If we can derive $\Gamma \vdash \Delta$ then $\Gamma \vdash \Delta$ is valid.

Completeness: If $\Gamma \vdash \Delta$ is valid then we can derive $\Gamma \vdash \Delta$.

The flaws we found so far (for example, in $\to L?$ and $\vee R_1?, \vee R_2?$ were failures of completeness: there were valid sequents we could not prove. Fortunately, our generalization to multiple conclusions have led us to a system that is both sound and complete.

There are other properties of interest. For example, we can ask if a logic is *decidable*. On the sequent calculus this would mean that we can effectively either prove or refute the validity of any given sequent. Again, our sequent calculus will support this property. More properties to come.

9 Proving Soundness

We start with soundness. The way we prove it is to show that whenever all premises of a rule are valid, so is the conclusion. Since at the leaves we have no premises, the conclusion must then be valid outright. And therefore any sequent we derive in a (complete) derivation must be valid. A nice consequence of this approach is that we can consider the soundness rule by rule. If we restrict our language, soundness still holds, and if we extend it we only have to check the new rules.

Since we have to talk a lot about validity, we sometimes abbreviate

$$\text{If all } F \in \Gamma \text{ are true then some } G \in \Delta \text{ is true}$$

as

$$\text{All } \Gamma \text{ true implies some } \Delta \text{ true}$$

Identity. Since there are no premises, we need to show that $\Gamma, F \vdash F, \Delta$ is valid. This means:

$$\text{All } (\Gamma, F) \text{ true implies some } (F, \Delta) \text{ true}$$

So we assume that all (Γ, F) are true, which implies that F is true. But that means at least one of (F, Δ) is true (namely F).

Implication Right ($\rightarrow R$).

$$\frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \rightarrow G, \Delta} \rightarrow R$$

We set up the proof as follows.

all (Γ, F) true implies some (G, Δ) true (validity of premise)
all Γ true (assumption)
...
some $(F \rightarrow G, \Delta)$ true (to show)

We proceed with the proof (filling in the gap marked by "...") by distinguishing cases for F: either F is true or F is false.

all (Γ, F) true implies some (G, Δ) true (validity of premise)
all Γ true (assumption)

case: F is true
  ...
  some $(F \rightarrow G, \Delta)$ true (to show)

case: F is false
  ...
  some $(F \rightarrow G, \Delta)$ true (to show)

some $(F \rightarrow G, \Delta)$ true (by cases on F)

Now we can fill in the gaps rather easily.

all (Γ, F) true implies some (G, Δ) true (validity of premise)
all Γ true (assumption)

case: F is true
  all (Γ, F) true (from assumption)
  some (G, Δ) true (from premise)

  subcase: G is true
    $F \rightarrow G$ true (by truth table for $\rightarrow$)
    some $(F \rightarrow G, \Delta)$ true (since $F \rightarrow G$ true)

  subcase: some Δ true
    some $(F \rightarrow G, \Delta)$ true (since some Δ true)

  some $(F \rightarrow G, \Delta)$ true (by cases on G, Δ)

case: F is false

$F \to G$ is true	(by truth table for $\to$)
some $(F \to G, \Delta)$ true	(since $F \to G$ true)
some $(F \to G, \Delta)$ true	(by cases on F)

This is a rather pedantic way to write down the steps, so in the next proof case we'll proceed more compactly.

Implication Left ($\to L$).

$$\frac{\Gamma \vdash F, \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \to G \vdash \Delta} \to L$$

Then we set up:

all Γ true implies some (F, Δ) true	(first premise)
all (Γ, G) true implies some Δ true	(second premise)
all $(\Gamma, F \to G)$ true	(assumption)
...	
some Δ true	(to show)

From the assumption and first premise, we know some (F, Δ) true. If Δ true we are done. Otherwise, F true and therefore (by assumption $F \to G$ true) we know G is true. By the second premise we then know Δ true.

All other rules can be proved sound in a similarly systematic manner.

Theorem 1 (Soundness of the Sequent Calculus)
If we can derive $\Gamma \vdash \Delta$ *then* $\Gamma \vdash \Delta$ *is valid*

Proof: All rules are sound. In particular, for the only leaves of the proof tree (applications of the identity rule), the conclusion is valid outright. All other rules preserve validity, and so any sequent we can derive is valid. □

It is somewhat peculiar (but common), that we motivate all the rules reading them bottom-up, but for soundness we argue with them top-down.

10 Inversion

We would like to build a theorem prover by constructing a proof in a bottom-up manner, using our rules of inference. A basic issue is that for a sequent $F_1, \ldots, F_n \vdash G_1, \ldots, G_m$ many rules may apply: left rules for formulas F_i and right rules for formulas G_j. Backtracking over all such choices quickly makes proof search infeasible.

In important class of rules are those that are *invertible*: if the conclusion is valid then are all the premises. We can always apply (bottom-up!) invertible rules in our proof search without having to backtrack: we don't lose validity.

In most inference systems we classify rules are being invertible or not. Here, remarkably, all rules are invertible! This means when a rule can be applied, it's always safe to do so.

We prove a few cases of the invertible rules. It is quite similar to the soundness proof of the rules.

Identity.

$$\frac{}{\Gamma, F \vdash F, \Delta}\ \mathsf{id}$$

The identity rule has no premises, so there is nothing to show.

Disjunction Right ($\vee R$) . Disjunction gave us problems when we conjecture two rules. Indeed, neither of them is invertible. Going to multiple conclusions solved this problem.

$$\frac{\Gamma \vdash F, G, \Delta}{\Gamma \vdash F \vee G, \Delta}\ \vee R$$

We set up:

all Γ true implies some $(F \vee G, \Delta)$ true	(validity of conclusion)
all Γ true	(assumption)
...	
some (F, G, Δ) true	(to show)

From the assumption and the validity of the conclusion we know that some $(F \vee G, \Delta)$ true. If some Δ is true, then also some (F, G, Δ). If $F \vee G$ is true then by the truth table, either F or G must be true. In either case, some of F, G, Δ true.

Disjunction Left ($\vee L$) .

$$\frac{\Gamma, F \vdash \Delta \quad \Gamma, G \vdash \Delta}{\Gamma, F \vee G \vdash \Delta}\ \vee L$$

We set up to show that the validity of the conclusion implies the validity of the first premise. The proof for the second premise is symmetric.

all $\Gamma, F \vee G$ true implies some Δ true	(validity of conclusion)
all Γ, F true	(assumption)
...	
some Δ true	(to show)

Since all Γ, F true, we have all Γ true and F true. Therefore (by truth table), $F \vee G$ true and also $\Gamma, F \vee G$. Then some Δ true by validity of the conclusion.

Theorem 2 (Invertibility)
All rules in the sequent calculus are invertible.

Proof: Case by case, as exemplified above. □

11 Termination

The sequent calculus in Figure 1 has another remarkable property: every premise of every rule is *smaller* than its conclusion. Smaller in which sense? Every rule, if read bottom-up, removes at least one of the connectives from the sequent. Therefore, if we just count the number of connectives, sequents become smaller during bottom-up proof search. Eventually we must either reach a rule with no premises, or we reach a sequent with no connectives:

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

Such a sequent is valid if and only if one of the p_i is equal to one of the q_j, that is, precisely if the identity rule applies. Otherwise, we can set all p_i to true and all q_j to false and observe that the sequent is not valid.

This means we can use proof search in the sequent calculus as a very simple decision procedure. Invertibility tells us we can blindly apply left and right rules without losing validity. Then we check if the identity rule applies to the leaves.

There is a small matter of strategy: in such a prover, we should probably apply 0-premise rules (proof is done!), before 1-premise rules (a proof goal replaced by an equivalent, smaller one), before 2-premise rules (because we now have two sequents to prove). This is only a heuristic and a matter of efficiency.

While sequent calculus search represents a decision procedure for validity in propositional logic, it is not one that is commonly used for efficiency reasons. The course *15-311 Logic and Mechanized Reasoning* goes into great detail about different approaches to proving propositional formulas and how to represent their proof in practice. It also examines many applications in mathematics and a few in computer science.

A consequence of the fact that all rules are invertible is that we could restrict the identity rule to just propositional variables, as in

$$\frac{}{\Gamma, p \vdash p, \Delta}\ \mathsf{id}^*$$

It wouldn't change the sequents we can derive, it might just make a few proofs slightly longer.

12 Completeness

We often have to deal with rules of inference that, taken as a whole, are not complete. That's because formal reasoning about arithmetic is inherently incomplete,

as demonstrated by Gödel [1931] and we generally include integers in programming languages. However, in the case of the sequent calculus for propositional language we do obtain completeness as a consequence of invertibility and termination.

Theorem 3 (Completeness of the Sequent Calculus)
If $\Gamma \vdash \Delta$ *is valid, then* $\Gamma \vdash \Delta$ *is derivable.*

Proof: Assume $\Gamma \vdash \Delta$ is valid. In some arbitrary order we try to construct a derivation of $\Gamma \vdash \Delta$. We can always make progress while preserving the validity of all goal sequents until we reach a sequent consisting entirely of variables. Since it must be valid, one of the antecedent must be the same as one of the succedents and the identity rule applies. Therefore, all branches in the derivation can be closed off. □

We close with word on terminology. In today's lecture, we have used the word "proof" in different ways. For one, we have used it to describe a *formal object* that we can construct, communicate, and check (whether by hand or by machine). For another, we have used it in the usual mathematical sense: a *rigorous mathematical argument* that some claim is true, but not necessarily a formal object. Adding the word "formal" in many places is awkward, so we will try to stick to the convention to say *deduction* or *derivation* for a formal object, while we continue to use the word *proof* in the usual mathematical sense.

13 Addendum: Negation

There is a few common logical connectives we didn't cover. For example, negation. Fortunately, negation is straightforward and preserves the good properties of the sequent calculus: the rules are sound and invertible, and the overall system remains complete and decidable.

$$\frac{\Gamma, F \vdash \Delta}{\Gamma \vdash \neg F, \Delta}\ \neg R \qquad \frac{\Gamma \vdash F, \Delta}{\Gamma, \neg F \vdash \Delta}\ \neg L$$

Let's prove the soundness of the right rule.

all (Γ, F) true implies some Δ true (premise)
all Γ true (assumption)
...
some $(\neg F, \Delta)$ true (to show)

We distinguish two cases. If F is true, then we know by assumption that all (Γ, F) true. Therefore by the premise, some Δ true and hence some $(\neg F, \Delta)$ true. If F is

false, then $\neg F$ is true and so, again some $(\neg F, \Delta)$ true. In either case, some $(\neg F, \Delta)$ true.

The remaining case of soundness and the two cases of invertibility follow similarly. Also, the premises of the two rules are smaller than the conclusion, so the decidability result and the overall completeness theorem continue to hold.

You might wonder about some other connectives, like logical equivalence also called bi-implication, $F \leftrightarrow G$. Instead of giving new rules, we can also consider it a *notational definition* and simply say

$$F \leftrightarrow G \quad \triangleq \quad (F \rightarrow G) \wedge (G \rightarrow F)$$

We can then expand the definition and just use the rule for implication and conjunction. In Assignment 1 you will explore something related.

References

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931. English translation in Solomon Feferman, editor, *Kurt Gödel Collected Works, Vol I*, pages 144–195, Oxford University Press, 1986.

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.

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

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.

Lecture Notes on Semantics of Programs

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

Lecture 4
September 5, 2024

1 Introduction

We begin by completing our initial tour of dynamic logic. Then we develop a semantics of programs so we can prove our rules sound and invertible (where it applies) with respect to the given semantic definition.

Unfortunately, the second part of this lecture is a bit technical, with a lot of new notation. We go into this level of rigor because it is important to understand the semantics of programs and how they relate to formal proofs. Only in this way can we be confident that the verification of a programs actually guarantees safety and other properties. It also provides the background and tools to consider extensions, variations, and implementations.

2 Loops

How does a loop $\mathbf{while}\ P\ \alpha$ execute? If P is true then we execute the loop body α once, followed again by $\mathbf{while}\ P\ \alpha$. If P is false we just exit the loop. The following rule suggests itself:

$$\frac{\Gamma \vdash [\mathbf{if}\ P\ \mathbf{then}\ (\alpha\ ;\ \mathbf{while}\ P\ \alpha)\ \mathbf{else}\ \mathbf{skip}]Q, \Delta}{\Gamma \vdash [\mathbf{while}\ P\ \alpha]Q, \Delta}\ [\mathbf{unwind}]R$$

Here we made up a new program **skip** that doesn't do anything. It behaves like the unit of parallel composition in that $\mathbf{skip}\ ;\ \alpha$ is equivalent to α. We could use $x := x$ instead, but that seems more complicated because it mentions a variable.

The problem with our first rule is that it replaces a program with a larger one, so it is not reductive. We can use the rules we already have to simplify it a bit to

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

This is better, but in the first premise we still have to reason about exactly the same program. So while these two rules are sound, their application is somewhat limited as we will see in the next lecture.

If you think back to 15-122 *Principles of Imperative Computation* you may remember how we reasoned about loops: we used *loop invariants*. In that course, loop invariants (like pre- and post-conditions for functions) where themselves *executable*. Here they are formulas and subject to logical reasoning. How do loop invariants work? Let's look at a trivial program:

$$\textbf{while}\ (x > 1)\ x := x - 2$$

under the precondition that (say) $x \geq 6$. After the loop we know that if the initial x was even, then in the poststate x must be 0, and if the initial x was odd, then in the poststate x must be 1. For safety properties that may a bit specific, so here we only want to ascertain that $0 \leq x \leq 1$ in the poststate.

In dynamic logic we express this as the proposition

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

But how do we prove it? What is the loop invariant? Recall:

- The loop invariant must be true initially.
- The loop invariant must be preserved by the loop body, under the assumption that the loop guard is true.
- The postcondition of the loop must be implied by the loop invariant together with the negated loop guard.

If we can prove all three of these then we can conclude the postcondition of the loop. In this example, we pick the loop invariant J to be $x \geq 0$. Then we have to prove:

True Initially $x \geq 6 \vdash x \geq 0$

Preserved $x \geq 0, x > 1 \vdash [x := x - 2]\, x \geq 0$

Implies Postcondition $x \geq 0, \neg(x > 1) \vdash 0 \leq x \leq 1$

These are all easy to prove (with an oracle for arithmetic), reducing the one in the middle in one step (using rule $[:=]R^{x'}$) to

$$x \geq 0, x > 1, x' = x - 2 \vdash x' \geq 0$$

Summarizing all of this with a rule yields the following, for an arbitrary loop invariant J.

$$\frac{\Gamma \vdash J, \Delta \quad J, P \vdash [\alpha]J \quad J, \neg P \vdash Q}{\Gamma \vdash [\mathbf{while}\ P\ \alpha]Q, \Delta}\ [\mathbf{while}]R$$

Sadly, since J is an arbitrary formula, this rule is not reductive. It would be okay, however, if we forced the programmer to write J in the program (as we do in C0), because then each premise only refers to components of the conclusion. When we want to emphasize this point we may write

$$[\mathbf{while}_J\ P\ \alpha]Q$$

where J is the loop invariant.

An important point about this rule is that we drop Γ and Δ in the second and third premise. This is because we don't know how often we may have to go around the loop. Preservation (the second premise) has to hold for any state we might reach during the iteration, but the antecedents in Γ are only guaranteed *before* the first iteration.

In our example, $x \geq 6$ is only known to be true before the loop starts, and not after each iteration. In fact, it may be false after the first iteration and therefore we cannot use it to prove for the second and third premise. Similarly, the additional succedents Δ also must be dropped. Otherwise we could reformulate the goal

$$\cdot \vdash [\mathbf{while}\ (x > 1)\ x := x - 2]\, 0 \leq x \leq 1, \neg(x \geq 6)$$

and then use $\neg R$ rule to turn the succedent $\neg(x \geq 6)$ into the (unwarranted) antecedent $x \geq 6$.

Putting all of this together, we can prove our program as follows.

$$\dfrac{\dfrac{\text{(by arithmetic)}\ x \geq 6 \vdash x \geq 0 \qquad \dfrac{\overset{\text{(by arithmetic)}}{x \geq 0, x > 1, x' = x - 2 \vdash x' \geq 0}}{x \geq 0, x > 1 \vdash [x := x - 2]\, x \geq 0}\ [:=]R^{x'} \qquad \overset{\text{(by arithmetic)}}{x \geq 0, \neg(x > 1) \vdash 0 \leq x \leq 1}}{x \geq 6 \vdash [\mathbf{while}\ (x > 1)\ x := x - 2]\, 0 \leq x \leq 1}\ [\mathbf{while}]R^*}{\cdot \vdash x \geq 6 \to [\mathbf{while}\ (x > 1)\ x := x - 2]\, 0 \leq x \leq 1}\ \to R$$

$(*)$ with loop invariant $J(x) = (x \geq 0)$

You can find a summary of the relevant rules we have intuited in Figure 1. The $[\mathbf{unfold}]R$ rule has a special status because it is not reductive (but will still turn out to be sound and useful). The $[\mathbf{while}]R$ rules is reductive only if the loop invariant J is specified in the syntax, as indicated by the subscript.

$$\frac{\Gamma, P \vdash [\alpha]Q, \Delta \quad \Gamma, \neg P \vdash [\beta]Q, \Delta}{\Gamma \vdash [\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]Q, \Delta}\,[\textbf{if}]R \qquad \frac{\Gamma, x' = e \vdash Q(x'), \Delta}{\Gamma \vdash [x := e]Q(x), \Delta}\,[:=]R^{x'}$$

$$\frac{\Gamma \vdash [\alpha]([\beta]Q), \Delta}{\Gamma \vdash [\alpha\,;\,\beta]Q, \Delta}\,[;]R \qquad \frac{\Gamma \vdash J, \Delta \quad J, P \vdash [\alpha]J \quad J, \neg P \vdash Q}{\Gamma \vdash [\textbf{while}_J\ P\ \alpha]Q, \Delta}\,[\textbf{while}]R$$

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

Figure 1: Some Rules for Dynamic Logic

3 Semantics of Expressions

Recall from Lecture 3

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	$::=$	$e_1 \leq e_2 \mid e_1 = e_2 \mid \ldots$
		$\mid$	$P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \neg P \mid \top \mid \bot$
		$\mid$	$[\alpha]Q \mid \langle\alpha\rangle Q$

In order to give semantics to formulas we also need to give semantics to programs and expressions. We start with expressions. We assume (for now) that the value of an expression is always an integer. We also need to recall that *states* map all variables to integers and write $\omega(x)$ for the value of variable x in state ω.

Generally, we write $[\![\text{something}]\!]$ for the semantic meaning of "something", so this notation will be overloaded. For expressions, it depends on a state and returns an integer. We write this as $\omega[\![e]\!] = c$.

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

There may be more cases if we consider additional operators. The operators inside the semantic brackets are syntax, the operators outside the semantic brackets are operations on integers. For example, in a state ω where $\omega(x) = 4$ we calculate

$$\omega[\![x + x]\!] = \omega[\![x]\!] + \omega[\![x]\!] = 4 + 4 = 8$$

4 Semantics of Programs

We interpret a program as denoting a relation between a prestates and poststates. Because of loops, for a given prestate there may be zero or one poststates. We again use semantic brackets, writing $\omega[\![\alpha]\!]\nu$ if the program α relates the prestate ω to the poststate ν. We write the program in the middle because, generally in mathematics, we like to write relations in infix form (like $e_1 \leq e_2$).

The definition follows, mostly, the way a program would execute and we only really have to think much when we come to loops.

Assignment. An assignment $x := e$ will evaluate e and then *update* the state so it now maps x to the value of e. We define the notation $\omega[x \mapsto c]$ with

$$\begin{array}{lcl} (\omega[x \mapsto c])(x) & = & c \\ (\omega[x \mapsto c])(y) & = & \omega(y) \quad \text{provided } x \neq y \end{array}$$

Note that the notation $\omega[x \mapsto c]$ has nothing to do with the formula $[\alpha]Q$, square brackets are overloaded as well.

Then we define

$$\omega[\![x := e]\!]\nu \quad \text{iff} \quad \omega[x \mapsto c] = \nu \text{ where } \omega[\![e]\!] = c$$

Sequential Composition. We execute $\alpha \,;\, \beta$ by first executing α and then β in the poststate of α.

$$\omega[\![\alpha \,;\, \beta]\!]\nu \quad \text{iff} \quad \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\beta]\!]\nu \text{ for some state } \mu$$

This definition if implies that if α has no poststate (that is, doesn't terminate) then $\alpha \,;\, \beta$ doesn't either, which is intuitively correct.

Conditionals. For conditionals we need to appeal to the meaning of formulas, because we need to know if the condition is true or false.

$$\begin{array}{lll} \omega[\![\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]\!]\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\nu \quad \text{or} \\ & & \omega \not\models P \text{ and } \omega[\![\beta]\!]\nu \end{array}$$

To be consistent, the meaning of a proposition should really be a truth value denoted by $\omega[\![P]\!]$, but there is a long tradition of writing $\omega \models P$ which means that P is true in state ω. Such a state is often called a *model* in which P is true.

Loops. As you might expect, loops are the trickiest. Here is a possible *recursive* definition.

$$\begin{array}{lll} \omega[\![\textbf{while } P \ \alpha]\!]\nu & \text{iff} & \omega \models P \text{ and } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\textbf{while } P \ \alpha]\!]\nu \quad \text{or} \\ & & \omega \not\models P \text{ and } \omega = \nu \end{array}$$

This definition is recursive in the sense that the program also appears on the right. It could then be ambiguous whether and for which states $\omega[\![\textbf{while } \top \textbf{ skip}]\!]\nu$ should be true.

We can use a more explicit *inductive definition* using an auxiliary relation $[\![\textbf{while } P\ \alpha]\!]^n$ that prescribes the number of iterations to be n.

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

If there is no such n, then there is no poststate for the given prestate.

5 Semantics of Formulas

The semantics of formulas in a given state must appeal to the meaning of expressions and the meaning of programs. Therefore the meanings of programs and formulas mutually depend on each other. We start with some simple cases.

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

For programs, we have to recall the informal definition from the previous lecture. $[\alpha]Q$ is true if Q is true in *every* poststate of α. Because a nonterminating program does not have a poststate, this is a statement about the *partial correctness* of the program α. Conversely, $\langle\alpha\rangle Q$ is true if Q is true in *some* poststate of α.

$$\begin{array}{lll} \omega \models [\alpha]Q & \text{iff} & \text{for every } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ we have } \nu \models Q \\ \omega \models \langle\alpha\rangle Q & \text{iff} & \text{there is a } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ and } \nu \models Q \end{array}$$

Now we say P is *valid* (written as $\models P$) if $\omega \models P$ for every state ω. (Recall that all states are defined on all variables, so this is well-defined.)

A sequent $P_1, \ldots, P_n \vdash Q_1, \ldots, Q_m$ is valid if for every state ω, whenever for all antecedents P_i we have $\omega \models P_i$ then for some succedent Q_j we have $\omega \models Q_j$.

6 Quantification and Substitution[1]

[1] Not covered in lecture

So far, we haven't introduced or used quantifiers, except the implicit quantification over all states in the definition of validity. It is very tempting to define that $\omega \models \forall x.\, P(x)$ if $\omega \models P(c)$ for every $c \in \mathbb{Z}$. Here, $P(c)$ is the notation of substituting c for every occurrence of x in $P(x)$.

This makes sense if $P(x)$ is a formula of pure arithmetic. But if $P(x)$ makes reference to programs this doesn't quite work. For example, given our earlier proof we might expect that

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

is true in every state. But we cannot substitute an integer for x for the same reason we needed to drop the antecedents Γ (and the succedents Δ) in the rule $[\mathbf{while}]R$: the guard of the loop implicitly refers to the x in many states (every state reachable by executing loop) and not just the initial state.

We therefore define instead:

$$\begin{array}{lll} \omega \models \forall x.\, P(x) & \text{iff} & \omega[x \mapsto c] \models P(x) \quad \text{for every } c \in \mathbb{Z} \\ \omega \models \exists x.\, P(x) & \text{iff} & \omega[x \mapsto c] \models P(x) \quad \text{for some } c \in \mathbb{Z} \end{array}$$

The proof rules then become a bit strange, but fortunately we will often be in the quantifier-free fragment, or can delegate quantifier reasoning to the arithmetic oracle. In both of these rules, the x' has to be chosen fresh (that is, it doesn't appear in the conclusion or in e). Also, for technical reasons we need to keep a copy of the existential in the $\exists R$ rule, just as a universally quantified antecedent may be needed more than once.

$$\frac{\Gamma \vdash P(x'), \Delta}{\Gamma \vdash \forall x.\, P(x), \Delta}\ \forall R^{x'} \qquad \frac{\Gamma, \forall x.\, P(x), x' = e, P(x') \vdash \Delta}{\Gamma, \forall x.\, P(x) \vdash \Delta}\ \forall L^{x'}$$

$$\frac{\Gamma, x' = e \vdash P(x'), \exists x.\, P(x), \Delta}{\Gamma \vdash \exists x.\, P(x), \Delta}\ \exists R^{x'} \qquad \frac{\Gamma, P(x') \vdash \Delta}{\Gamma, \exists x.\, P(x) \vdash \Delta}\ \exists L^{x'}$$

Reading the rules $\forall L^{x'}$ and $\exists R^{x'}$ bottom-up, we can freely choose the expression e to instantiate the quantifier with as long as it doesn't mention x'.

7 Rules versus Axioms

In general, it seems more convient to reason with rules since the rules of the sequent calculus give clear view of the state of an incomplete proof. In some situations, though, we can pack a lot of information into *axioms*, by which we mean valid formulas.

Recall the right rule for sequential composition.

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

We reduce the goal of proving $[\alpha\ ;\ \beta]Q$ to the goal of proving $[\alpha]([\beta]Q)$. What would be a corresponding *left* rule $[;]L$? What can we deduce from knowing $[\alpha\ ;\ \beta]Q$? It looks as if we should just be able to replace this with the antecedent $[\alpha]([\beta]Q)$ because $[\alpha\ ;\ \beta]Q$ and $[\alpha]([\beta]Q)$ are equivalent.

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

Here is a general observation: if P and Q are equivalent (in the sense that $P \leftrightarrow Q$ is valid) then rules such as

$$\frac{\Gamma \vdash Q, \Delta}{\Gamma \vdash P, \Delta} \qquad \frac{\Gamma, Q \vdash \Delta}{\Gamma, P \vdash \Delta}$$

are **both sound and invertible**. That's because P and Q are always either both false or both true, regardless of the state because the bi-implication $P \leftrightarrow Q$ is valid.

In order to obtain good left and right rules from valid equivalences we just have to make sure the rules are reductive. As an example, the $[;]R$ and $[;]L$ rules can both be constructed from the following equivalence.

$$\models [\alpha\ ;\ \beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

Let's prove this. We start with the right-to-left direction, which implies the *soundness* of $[;]R$ and *invertibility* of $[;]L$. We set up the proof:

$\omega \models [\alpha]([\beta]Q)$ (assumption)
...
$\omega \models [\alpha\ ;\ \beta]Q$ (to show)

We now walk through the proof in individual steps, narrowing the gap, sometimes from below and sometimes from above. Typically, one only presents the end result and the reader has to figure out how one might have obtained it.

By definition, the conclusion holds if for every state ν such that $\omega[\![\alpha\ ;\ \beta]\!]\nu$ we have $\nu \models Q$. Now our proof state is (highlighting the new parts in blue):

$\omega \models [\alpha]([\beta]Q)$ (1, assumption)
$\omega[\![\alpha\ ;\ \beta]\!]\nu$ for some ν (2, assumption)
...
$\nu \models Q$ (to show)
$\omega \models [\alpha\ ;\ \beta]Q$ (by defn. of $\models$)

By definition, assumption 2 is true if there is some intermediate state μ such that $\omega[\![\alpha]\!]\mu$ and $\mu[\![\beta]\!]\nu$. Let's write this into the proof as well.

$\omega \models [\alpha]([\beta]Q)$ (1, assumption)
$\omega[\![\alpha\ ;\ \beta]\!]\nu$ for some ν (2, assumption)
$\omega[\![\alpha]\!]\mu$ and $\mu[\![\beta]\!]\nu$ for some μ (3, from 2 by defn. of $[\![-]\!]$)
...
$\nu \models Q$ (to show)
$\omega \models [\alpha\ ;\ \beta]Q$ (by defn. of $\models$)

Next: from assumption 1 and the fact that $\omega[\![\alpha]\!]\mu$ we can conclude that $\mu \models [\beta]Q$, again just by the definition of $\models$.

$\omega \models [\alpha]([\beta]Q)$ (1, assumption)
$\omega[\![\alpha\ ;\ \beta]\!]\nu$ for some ν (2, assumption)
$\omega[\![\alpha]\!]\mu$ and $\mu[\![\beta]\!]\nu$ for some μ (3, from 2 by defn. of $[\![-]\!]$)
$\mu \models [\beta]Q$ (4, from 1 and 3(a) by defn. of $\models$)
...
$\nu \models Q$ (to show)
$\omega \models [\alpha\ ;\ \beta]Q$ (by defn. of $\models$)

Now we use the same argument knowing that $\mu[\![\beta]\!]\nu$ and $\mu \models [\beta]Q$ to conclude that $\nu \models Q$. But that's what we needed to show!

$\omega \models [\alpha]([\beta]Q)$ (1, assumption)
$\omega[\![\alpha\ ;\ \beta]\!]\nu$ for some ν (2, assumption)
$\omega[\![\alpha]\!]\mu$ and $\mu[\![\beta]\!]\nu$ for some μ (3, by defn. of $[\![-]\!]$ from 2)
$\mu \models [\beta]Q$ (4, from 1 and 3(a) by defn. of $\models$)
$\nu \models Q$ (5, from 4 and 3(b) by defn. of $\models$)
$\omega \models [\alpha\ ;\ \beta]Q$ (from 5 and 2 by defn. of $\models$)

We see the proof is actually quite straightforward. We just have to carefully unwind the definitions.

Here is the proof in the other direction.[2] We set up:

$\omega \models [\alpha\ ;\ \beta]Q$ (1, assumption)
...
$\omega \models [\alpha]([\beta]Q)$ (to show)

We show the filled-in proof. You can probably walk through it in the order we made the deductions.

$\omega \models [\alpha\ ;\ \beta]Q$ (1, assumption)
$\omega[\![\alpha]\!]\mu$ for some μ (2, assumption)
$\mu[\![\beta]\!]\nu$ for some ν (3, assumption)
$\omega[\![\alpha\ ;\ \beta]\!]\nu$ (4, from 2 and 3 by defn. of $[\![-]\!]$
$\nu \models Q$ (5, from 1 and 4 by defn of $\models$)
$\mu \models [\beta]Q$ (6, from 5 and 3 by defn of $\models$)
$\omega \models [\alpha]([\beta]Q)$ (7, from 6 and 2 by defn of $\models$)

[2] not shown in lecture

At this point we have shown that the right and left rules for the sequential composition of programs in a box modality are sound and invertible.

8 Some Axioms for Dynamic Logic[3]

Based on the insights from the previous section and the rules for programs in dynamic logic, we can conjecture the following axioms. And, indeed, they are all valid, even though we don't show the proofs. We write the postfix "A" to indicate that what we are naming is not a rule but an axiom.

$[:=]A$	$[x := e]Q(x) \leftrightarrow \forall x'.x' = e \rightarrow Q(x')$ (x' not in e or $Q(x)$)
$[;]A$	$[\alpha\ ;\ \beta]Q \leftrightarrow [\alpha]([\beta]Q)$
$[\mathbf{if}]A$	$[\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q \leftrightarrow (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q)$
$[\mathbf{unfold}]A$	$[\mathbf{while}\ P\ \alpha] \leftrightarrow (P \rightarrow [\alpha][\mathbf{while}\ P\ \alpha]Q) \wedge (\neg P \rightarrow Q)$

Because these axioms are valid biconditionals, we obtain correct left and right rules for the sequent calculus, with the rules for [**unfold**] being somewhat unsatisfactory since they are not reductive.

Unfortunately, the rule $[\mathbf{while}]R$ including loop invariants can't be easily turned into an axiom. Recall:

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

We won't propose a corresponding left rule for this because it takes us into the realm of total correctness and proving termination, which is beyond the scope of this course.

We won't pursue it further, but for the curious, it is possible to obtain an axiom for the direction that corresponds to soundness of $[\mathbf{while}]R$, but we need an additional logical operator $\Box P$.

$$[\mathbf{while}]A \quad [\mathbf{while}\ P\ \alpha]Q \leftarrow J \wedge \Box(J \wedge P \rightarrow [\alpha]J) \wedge \Box(J \wedge \neg P \rightarrow Q)$$

The new component here is the modality $\Box P$ which is defined semantically by $\omega \vdash \Box P$ iff $\nu \vdash P$ for every state ν. Needing to prove this is what removes Γ and Δ in the bottom-up reading of $[\mathbf{while}]R$.

As an aside, the problem with substitution and the $\Box$ modality from modal logic is nothing new but has concerned philosophers long before computer science and dynamic logic. See, for example, Garson [2000, Section 16].

9 Summary

We summarize the semantic definitions and axioms; the sequent calculus rules can be found in Figure 1.

[3] Only $[;]A$ covered in lecture; the remainder may be useful for reference or further reading.

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

Figure 2: 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 3: Semantics of Programs

$$
\begin{array}{lll}
\omega \models e_1 \leq e_2 & \text{iff} & \omega[\![e_1]\!] \leq \omega[\![e_2]\!] \\
\omega \models e_1 = e_2 & \text{iff} & \omega[\![e_1]\!] = \omega[\![e_2]\!] \\
\\
\omega \models P \wedge Q & \text{iff} & \omega \models P \quad \text{and} \quad \omega \models Q \\
\omega \models P \vee Q & \text{iff} & \omega \models P \quad \text{or} \quad \omega \models Q \\
\omega \models P \rightarrow Q & \text{iff} & \omega \models P \quad \text{implies} \quad \omega \models Q \\
\omega \models \neg P & \text{iff} & \omega \not\models P \\
\omega \models P \leftrightarrow Q & \text{iff} & \omega \models P \quad \text{iff} \quad \omega \models Q \\
\\
\omega \models [\alpha]Q & \text{iff} & \text{for every } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ we have } \nu \models Q \\
\omega \models \langle\alpha\rangle Q & \text{iff} & \text{there is a } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ and } \nu \models Q
\end{array}
$$

Figure 4: Semantics of Formulas

References

James Garson. Modal logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Spring 2024 edition edition, 2000. URL https://plato.stanford.edu/archives/spr2024/entries/logic-modal/.

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. We can think of the symbol $\lightning$ as denoting an unsafe state, different from the states we have been using so far (ω, μ, ν) from which the program can proceed as expected. Formally,

we don't change our definition of state as a total map from variables to integers, which is why we instead introduce a new notation and new relation. 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:

$$\omega[\![x := e]\!]\lightning \quad \text{never}$$

To make our semantic definitions more uniform, we will often state this equivalently as

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

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.

$$\omega[\![\alpha\ ;\ \beta]\!]\lightning \quad \text{iff} \quad \begin{array}{l}\text{either } \omega[\![\alpha]\!]\lightning \\ \text{or } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\beta]\!]\lightning \text{ for some } \mu\end{array}$$

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.

$$\omega[\![\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]\!]\lightning \quad \text{iff} \quad \begin{array}{l}\text{either } \omega \models P \text{ and } \omega[\![\alpha]\!]\lightning \\ \text{or } \omega \not\models P \text{ and } \omega[\![\beta]\!]\lightning\end{array}$$

While loop. **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).

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

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

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

Using the rule $[\mathbf{divide}]R$, 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:

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

By the loop guard we see that $x > 1$ inside the loop body, so the division is safe. However, if we had written $\mathbf{divide}\ x\ y$ 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

$$\mathbf{assert}\ P$$

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

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

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

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 $x := \mathbf{divide}\ e_1\ e_2$ by $\mathbf{assert}\ \neg(e_2 = 0)\ ;\ x := e_1/e_2$ the two programs would have the same meaning in every state (either both unsafe, or both safe and determinate).

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

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

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)
...
$\omega \models P \wedge Q$ (to show)

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. □

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/.

[1] mentioned, but not explicitly stated in lecture

Lecture Notes on Memory Safety

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

Lecture 6
September 12, 2024

1 Introduction

The classic *buffer overflow attack* (about which you learn in 15-213 *Computer Systems*) allows a program to take control of your machine while otherwise innocuous code is executed. It exploits that accessing memory that hasn't been explicitly allocated by a program is undefined behavior in C and therefore *unsafe*.

In this lecture we define a simplistic model of memory and introduce memory write and read operations into our language. We then define unsafe behavior and investigate how to prove safety of programs accessing memory.

What can we do if we cannot prove safety, but we'd still like to run a program? One option is to rewrite the program by checking that any memory access is in bounds before running it. We show how this can be done for our language, and prove safety for the resulting program. This is one of techniques used in *sandboxing* which refers to running untrusted programs in a manner that prevents them from doing damage ("*inside a sandbox*"). There are commercial tools such as Intel's Pin that can instrument binary code for Intel instruction set architectures.

2 Writing and Reading Memory

Mathematically, we model memory as a map from $\mathbb{Z}$ (the index domain) to $\mathbb{Z}$ (the value domain). It is total, but may be *indeterminate* on some indices. In the programming language we assume indices are limited to the range from 0 to a fixed U, and accessing memory outside these bounds in *unsafe*.

As explained in Lecture 5, we would like to keep expressions *safe*, but possibly *indeterminate*. Unsafe behavior is then exhibited only by commands and the programs constructed from them. Sticking to this approach, we add a new kind of variable, M, that stands for memory, and the new commands

- $M[e_1] := e_2$ write e_2 into memory M at address e_1. This is unsafe if the value of e_1 is out of bounds.
- $x := M[e]$ set x to the contents of memory at address e into x. This is unsafe if the value of e is out of bounds.

The design decision that memory access takes place via commands means that you have to rewrite hypothetical code such as

$$M[x] := (M[x-1] + M[x+1])/2$$

in the more verbose form

$$\begin{array}{l} t_1 := M[x-1]\ ; \\ t_2 := M[x+1]\ ; \\ t_3 := \mathbf{divide}\ (t_1 + t_2)\ 2\ ; \\ M[x] := t_3 \end{array}$$

Next, we need to rigorously define the semantics of the new commands so that (a) we can implement them, and (b) we can prove soundness of our axioms and rules to reason about them (including their safety).

The first issue is how to track the contents of memory. For that purpose, we change our definition of the state ω. So far the state has been a total function from variables to integers, $\omega : \mathsf{Var} \to \mathbb{Z}$ where Var is the (countably infinite) set of variables. Now variables can also map to memory, which is a total function from integers to integers.

$$\omega : \mathsf{Var} \to (\mathbb{Z} \cup (\mathbb{Z} \to \mathbb{Z}))$$

We assume that programs map lowercase variables to integers and uppercase variables to memory, so that there is never any confusion between the two forms. Mathematically, we use the letter $H : \mathbb{Z} \to \mathbb{Z}$ (suggesting a *heap*). All our previous semantic definitions remain unchanged since all variables in those definitions stand for integers.

We begin with safe and unsafe memory reads.

$$\omega[\![x := M[e]]\!]\nu \quad \text{iff} \quad \omega[\![e]\!] = i \text{ and } \omega[\![M]\!] = H \text{ and } \nu = \omega[x \mapsto H(i)] \quad \text{provided } 0 \le i < \mathsf{U}$$

$$\omega[\![x := M[e]]\!]\lightning \quad \text{iff} \quad \omega[\![e]\!] = i \text{ and } \textbf{not } 0 \le i < \mathsf{U}$$

It should be clear that unsafe programs continue to satisfy no postcondition.

Memory write follows the same intuition, we just have to make sure to suitably update the map defining the state of memory.

$$\omega[\![M[e_1] := e_2]\!]\nu \quad \text{iff} \quad \omega[\![e_1]\!] = i \text{ and } \omega[\![e_2]\!] = a \text{ and } \omega[\![M]\!] = H \text{ and } H' = H[i \mapsto a] \text{ and } \nu = \omega[M \mapsto H'] \quad \text{provided } 0 \le i < \mathsf{U}$$

$$\omega[\![M[e_1] := e_2]\!]\lightning \quad \text{iff} \quad \omega[\![e_1]\!] = i \text{ and } \textbf{not } 0 \le i < \mathsf{U}$$

3 Reasoning about Memory[1]

In order to reason about memory we need to introduce expressions that capture what we know about the state of memory. Mathematically, this leads us to the *theory of arrays* [McCarthy, 1962] which we can view as being constructed on top of the theory of arithmetic we have assumed so far. Because of the importance of arrays in imperative programming, some efficient decision procedures have been devised (see, for example, Stump et al. [2001]) and implemented in provers such as Z3 or the CVC family.

In the theory of arrays we have two expressions $\mathbf{read}\ H\ i$ and $\mathbf{write}\ H\ i\ a$ where H denotes an array, i and index into an array, and a a value stored in the array. Note that the expression $\mathbf{write}\ H\ i\ a$ denotes a "new" array; semantically $H[i \mapsto a]$. For us, both the index domain and the values are integers.

There are two axioms, called *read over write*, that allow us to reason about these expressions.

$$i = k \rightarrow \mathbf{read}\ (\mathbf{write}\ H\ i\ a)\ k = a$$
$$i \neq k \rightarrow \mathbf{read}\ (\mathbf{write}\ H\ i\ a)\ k = \mathbf{read}\ H\ k$$

In addition we have an *axiom of extensionality* which states that two arrays are equal if they agree on all elements. In our use of the theory of arrays, the quantifier ranges over integers.

$$(\forall i.H(i) = H'(i)) \rightarrow H = H'$$

As usual, we treat the new expressions as always denoting either an array or an integer, although the value may sometimes be indeterminate.

The right rules of the sequent calculus for these new commands are now relatively straightforward.

$$\frac{\Gamma \vdash 0 \leq e < \mathsf{U}, \Delta \quad \Gamma, x' = \mathbf{read}\ M\ e \vdash Q(x'), \Delta}{\Gamma \vdash [x := M[e]]Q(x), \Delta}\ [\mathbf{read}]R^{x'}$$

As for assignment, the x' must be chosen fresh in $[\mathbf{read}]R^{x'}$. The same is true for M' in the following rule.

$$\frac{\Gamma \vdash 0 \leq e_1 < \mathsf{U}, \Delta \quad \Gamma, M' = \mathbf{write}\ M\ e_1\ e_2 \vdash Q(M'), \Delta}{\Gamma \vdash [M[e_1] := e_2]Q(M), \Delta}\ [\mathbf{write}]R^{M'}$$

Even if an program can only mention a single array M, while reasoning about a program we need to be able to relate arrays before and after an assignment. Therefore, the knowledge about M in the antecedents Γ (or succedents Δ) must not conflict with knowledge about the state of the array after the write operation and M' must be fresh.

The aspect of these rules critical for safety is the first premise that requires us to prove safety (independently of the postcondition).

[1] only part of the presented in lecture; the remainder promised and provided here for reference

4 A Small Example of Memory Safety

Consider the following program to initialize memory up to n:

$$i := 0 \,;\, \textbf{while}\ (i < n)\{\, M[i] := i \,;\, i := i + 1 \,\}$$

This is patently unsafe: just consider $n = \mathsf{U} + 1$. Then the last time around the loop we will have $i = \mathsf{U}$, leading to an unsafe memory access at $M[\mathsf{U}]$.

We can add a precondition $n \leq \mathsf{U}$ and then try to prove safety with

$$n \leq \mathsf{U} \rightarrow [i := 0 \,;\, \textbf{while}\ (i < n)\{\, M[i] := i \,;\, i := i + 1 \,\}]\top$$

We could try $i \leq n$ but that's insufficient. Here is what preservation would require:

$$i \leq n, i < n \vdash [M[i] := i \,;\, i := i + 1]i \leq n$$

We note two problems: (1) safety will fail because we cannot prove that $0 \leq i$ and (2) we have lost the precondition $n \leq \mathsf{U}$ so we also cannot conclude that $i < \mathsf{U}$.

Let's try a more complex invariant:

$$0 \leq i \leq n \leq \mathsf{U}$$

Now preservation requires

$$0 \leq i \leq n \leq \mathsf{U}, i < n \vdash [M[i] := i \,;\, i := i + 1]\,(0 \leq i \leq n \leq \mathsf{U})$$

This reduces in two steps to

$$0 \leq i \leq n \leq \mathsf{U}, i < n, M' = \textbf{write}\ M\ i\ i, i' = i + 1 \vdash 0 \leq i' \leq n \leq \mathsf{U}$$

Fortunately, this is valid (even with the useless assumption about M'). Because our postcondition is just $\top$, that is easily seen to be true, but the loop invariant does not hold initially because

$$n \leq \mathsf{U} \vdash 0 \leq 0 \leq n \leq \mathsf{U}$$

is not valid (counterexample: $n = -1$). So we need to strengthen our precondition to

$$0 \leq n \leq \mathsf{U}$$

5 Guards

Consider the scenario where you are given the program from the previous section but no loop invariant. We might be able to guess a loop invariant, but if not we are stuck. The program looks unsafe, even with the precondition $0 \leq n \leq \mathsf{U}$. If we still need to run it, what can we do? One option would be *dynamic monitoring*: we track

memory accesses as the program executes and abort it if it attempts to do something unsafe. Another one is to *instrument it with guards* before memory accesses. These guards abort the program if the access would be unsafe and let it go on if they are safe. Aborting programs is considered safe, because aborting is actually a well-defined operation that does no harm (except to the running program, but it is its own fault if it tries to execute an unsafe command). For this purpose we need a new command $\mathbf{test}\ P$. In the literature on dynamic logic this is called a *guard* and written as $?P$. It has the following specification.

$$\begin{array}{lll} \omega[\![\mathbf{test}\ P]\!]\nu & \text{iff} & \omega \models P \text{ and } \nu = \omega \\ \omega[\![\mathbf{test}\ P]\!]\lightning & \text{iff} & \text{false} \end{array}$$

The program $\mathbf{test}\ \bot$ will not have a poststate, but it is also safe because it aborts. As a result, based on the definition of $\omega \models [\alpha]Q$ it is the case that

$$\omega \models [\mathbf{test}\ \bot]Q$$

for any ω and Q. This in turn means that $[\mathbf{test}\ \bot]Q$ is logically valid and $\cdot \vdash [\mathbf{test}\ \bot]Q$ should be derivable.

Just to be sure, let's recall the definition of $\omega \models [\alpha]Q$ from Lecture 5, page L5.4.

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

This is a *partial correctness* statement: if there is no poststate ν such that $\omega[\![\omega]\!]\nu$, then the first part of the condition is vacuously true.

What does this mean for the axiom for $[\mathbf{test}\ P]Q$? If P is true, then Q should also be true. But if the test succeeds then we know P, so we conjecture (somewhat rashly, perhaps)

$$[\mathbf{test}\ P]Q \leftrightarrow (P \rightarrow Q)$$

What if P is false? Then the program $\mathbf{test}\ P$ has no poststate, and yet it is safe. Consequently $[\mathbf{test}\ P]Q$ should be true, and by this axiom it will be becase $\bot \rightarrow Q$ is valid.

Let's prove that this axiom is valid, just as in Theorem 1 of Lecture 5 we proved that $[\mathbf{assert}\ P]Q \leftrightarrow (P \wedge Q)$.

Theorem 1 *The axiom*

$$[\mathbf{test}\ P]Q \leftrightarrow (P \rightarrow Q)$$

is valid.

Proof: From right to left we set up for an arbitrary ω

$\omega \models P \rightarrow Q$ (assumption)

$\ldots$

$\omega \models [\mathbf{test}\ P]Q$ (to show)

The conclusion holds if for every ν such that $\omega[\![\mathbf{test}\ P]\!]\nu$ we have $\nu \models Q$ (and **not** $\omega \models [\![\mathbf{test}\ P]\!]\lightning$, which is true).

So we assume $\omega[\![\mathbf{test}\ P]\!]\nu$ and have to show that $\nu \models Q$. By definition, this second assumption give us $\omega \models P$ and $\nu = \omega$.

By the first assumption also $\omega \models Q$ and since $\nu = \omega$ we have $\nu \models Q$

For the left to right direction, we set up for an arbitrary ω

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

So we assume $\omega \models P$ and it remains to show that $\omega \models Q$. Since $\omega \models P$, the first assumption gives us $\nu \models Q$ for any ν with $\omega[\![\mathbf{test}\ P]\!]\nu$. We can use this for $\nu = \omega$ (since $\omega \models P$) to obtain $\omega \models Q$. □

We can easily turn the two directions of the axiom into right and left rules of the sequent calculus.

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

Here is a little table on the differences between $\mathbf{assert}\ P$ and $\mathbf{test}\ P$.

Poststate	$\omega[\![\mathbf{assert}\ P]\!]\nu$ iff $\omega \models P$ and $\nu = \omega$	$\omega[\![\mathbf{test}\ P]\!]\nu$ iff $\omega \models P$ and $\nu = \omega$
Safety	$\omega[\![\mathbf{assert}\ P]\!]\lightning$ iff $\omega \not\models P$	$\omega[\![\mathbf{test}\ P]\!]\lightning$ never
Axiom	$[\mathbf{assert}\ P]Q \leftrightarrow P \wedge Q$	$[\mathbf{test}\ P]Q \leftrightarrow (P \to Q)$

6 Sandboxing

Sandboxing unsafe behavior (including memory access through the read or write commands) proceeds as follows. We replace

- every memory read $x := M[e]$ with the program $\mathbf{test}\ 0 \le e < \mathsf{U}\ ;\ x := M[e]$
- every memory write $M[e_1] := e_2$ with the program $\mathbf{test}\ 0 \le e_1 < \mathsf{U}\ ;\ M[e_1] := e_2$
- every division $x := \mathbf{divides}\ e_1\ e_2$ with the program $\mathbf{test}\ e_2 \neq 0\ ;\ x := \mathbf{divides}\ e_1\ e_2$.

- every assertion **assert** P with the program **test** P

Now we can safely execute the program. Equally importantly, perhaps, we can prove the safety of the program transformed in this manner.

Theorem 2 (Safety of Sandboxed Programs) *Given a program α under precondition P, we obtain the sandboxed α' as defined in the preceding paragraph.*
Then $\cdot \vdash P \to [\alpha']\top$.

Proof: We prove safety using the loop invariant $\top$ for every loop. Since any potentially unsafe command is immediately preceded by a guard, the safety condition incorporated into the rule will be provable since it is exactly the assumption enabled by the postcondition of the guard.

More formally, this proof would be carried out by an *induction over the structure of formulas and programs*. □

There are two optimizations that come to mind. We can introduce fresh temporaries in order to avoid recomputing the value of expressions. For example, instead of **test** $0 \leq e < \mathsf{U}$; $x := M[e]$ we would insert $t := e$; **test** $0 \leq t < \mathsf{U}$; $x := M[t]$ for a fresh temporary variable t.

The other optimization is a bit trickier. At first one might think that if we can *prove* P when we encounter **test** P during the verification of safety we can replace it by **skip** (or **assert** $\top$, which should be equivalent). However, in conditionals the postcondition is replicated:

$$[\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]Q \leftrightarrow (P \to [\alpha]Q) \wedge (\neg P \to [\beta]Q)$$

When the postcondition contains a program (which arises from sequential composition, for example), this program may be proved twice, once in each branch. A similar remark applies if we unfold loops because the program α is replicated.

So we can only replace **test** P with **skip** only if for all occurrences of $[\textbf{test } P]Q$ in a safety proof we can prove P.

Returning to our earlier example, we can sandbox

$$n \leq \mathsf{U} \to [i := 0 \,;\, \textbf{while } (i < n)\{\, M[i] := i \,;\, i := i + 1 \,\}]\top$$

as

$$n \leq \mathsf{U} \to [i := 0 \,;\, \textbf{while } (i < n)\{\, \textbf{test} 0 \leq i < \mathsf{U} \,;\, M[i] := i \,;\, i := i + 1 \,\}]\top$$

Without a loop invariant and stronger precondition we can't eliminate the guard. With the additional information from Section 4 it would be redundant and can be dropped.

7 Summary

Since it has been a while, we summarize the language so far. We restrict programs from containing certain expressions with indeterminate behavior to retain the property that for every given prestate ω, every program has three possible outcomes: a poststate ν, or no poststate in which case it may be safe or unsafe ($\lightning$).

Variables	x, y, z			
Memory	M			
Constants	c	$::=$	$\ldots, -1, 0, 1, \ldots$	
Expressions	e	$::=$	$c \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2$	determinate
		$\mid$	$e_1/e_2 \mid \textbf{read}\ M\ e \mid \textbf{write}\ M\ e_1\ e_2$	may be indeterminate
Programs	α, β	$::=$	$x := e \mid \alpha\ ;\ \beta \mid \textbf{skip}$	
		$\mid$	$\textbf{if}\ P\ \textbf{then}\ \alpha\ \textbf{else}\ \beta \mid \textbf{while}\ P\ \alpha$	
		$\mid$	$\textbf{test}\ P$	safe
		$\mid$	$\textbf{assert}\ P \mid x := \textbf{divide}\ e_1\ e_2$	may be unsafe
		$\mid$	$x := M[e] \mid M[e_1] := e_2$	may be unsafe
Formulas	P, Q	$::=$	$e_1 \leq e_2 \mid e_1 = e_2 \mid \top \mid \bot$	
		$\mid$	$P \wedge Q \mid P \vee Q \mid P \to Q \mid P \leftrightarrow Q \mid \neg P$	
		$\mid$	$\forall x.\, P(x) \mid \exists.\, P(x)$	
		$\mid$	$[\alpha]Q \mid \langle\alpha\rangle Q$	

References

John McCarthy. Towards a mathematical science of computation. In *2nd IFIP Congress on Information Processing*, pages 21–28, Munich, Germany, August 1962. North-Holland.

Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Symposium on Logic in Computer Science (LICS 2001)*, pages 29–37, Boston, Massachusetts, June 2001. IEEE Computer Society.

Lecture Notes on Generating Verification Conditions

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

Lecture 7
September 17, 2024

1 Introduction

Dynamic logic is very general. Among other things, it allows us to prove program equivalence, which you explored in Assignment 2 (Task 5) and Assignment 3 (Task 1). Here is another instance. Just using the axiom

$$[\alpha\ ;\ \beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

we can formally prove that sequential composition is associative.

$[\alpha\ ;\ (\beta\ ;\ \gamma)]Q$
$\leftrightarrow [\alpha]([\beta\ ;\ \gamma])Q$
$\leftrightarrow [\alpha]([\beta]([\gamma]Q))$
$\leftrightarrow [\alpha\ ;\ \beta]([\gamma]Q)$
$\leftrightarrow [(\alpha\ ;\ \beta)\ ;\ \gamma]Q$

When we are proving safety (or even correctness) we'd like to take advantage of the special form of pre- and post-conditions that are formulated purely in the theory of arithmetic (or maybe the theory of arrays if we model memory), namely

$$P \rightarrow [\alpha]Q$$

where P is the precondition, α is the program we are trying to verify, and Q is the postcondition. This is often written as $\{P\}\ \alpha\ \{Q\}$ and called a *Hoare triple*.

For this and the next lecture, we make the following assumptions:

1. The precondition P and postcondition Q are formulas of pure arithmetic. While they may contain expressions, they may not contain programs.

2. All formulas occurring in the program α (in conditionals, loop guards, assertions, and tests) are formulas of pure arithmetic.

The formulas of *pure arithmetic* are a subset of all formulas defined by

$$\text{Pure formulas} \quad P, Q \quad ::= \quad e_1 \leq e_2 \mid e_1 = e_2 \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid \neg P \mid \top \mid \bot$$

We probably should use a new notation for such formulas, but since in today's lecture we consider only pure ones (except when recalling axioms), maybe we can track this in our heads.

As we proceed, we should also consider the status of quantifiers. On the one hand we sometimes need to refer to them in pre- and post-conditions (especially when reasoning about arrays). On the other hand, they make the theorem proving problem much harder. Plus, they shouldn't appear in program formulas like conditionals or loop guards because then rather than computing a Boolean value, we'd have to crank up a theorem prover while the program is running (potentially have to solve a problem in some undecidable class).

2 Weakest Liberal Precondition

If we are given a program α and a postcondition Q, then a *sufficient precondition* is a pure formula P such that $P \rightarrow [\alpha]Q$. For example, $P = \bot$ is a sufficient precondition for any α and Q since $\bot \rightarrow [\alpha]Q$ is valid. We would like the *weakest* such precondition which means that it is implied by any other precondition. We call this the weakest *liberal* precondition if we consider *partial correctness*, which is exactly what $[\alpha]Q$ captures. As has become common practice we may drop the adjective *liberal* since we essentially never consider *total correctness* (that is, require termination to be proved).

In order to explain the term "weakest": we say P is *stronger than* Q if P implies Q. Then $P = \bot$ is the strongest formula (it implies everything), while $P = \top$ is the weakest: it only implies Q if Q is already true without the help of P.

Writing $\mathsf{wlp}\ \alpha\ Q$ for the weakest liberal precondition we want the following two properties:

1. $\mathsf{wlp}\ \alpha\ Q \rightarrow [\alpha]Q$ (it is a precondition)
2. If $P \rightarrow [\alpha]Q$ then $P \rightarrow \mathsf{wlp}\ \alpha\ Q$ (it the weakest among them).

It is easy to check that $\mathsf{wlp}\ \alpha\ Q \leftrightarrow [\alpha]Q$ because $[\alpha]Q$ satisfies both conditions. Sadly, we cannot just define $\mathsf{wlp}\ \alpha\ Q = [\alpha]Q$ because the right-hand side is not pure, and we therefore cannot just hand it off to a theorem prover for arithmetic.

When thinking about the desired property it quickly becomes clear that loops are a major obstacle to computing the weakest liberal precondition algorithmically. So we require the programmer to supply a *loop invariant* J for every loop and then construct a weakest liberal precondition *with respect to the given loop invariants*. We'll come back to this point in Section 4. For all the other constructs, we can derive an algorithm for computing it from the axioms.

When given a problem $P \to [\alpha]Q$, then we call $P \to \mathsf{wlp}\ \alpha\ Q$ the *verification condition*. If it can be shown valid by an arithmetic prover, then the original dynamic logic formula $P \to [\alpha]Q$ is valid.

3 Programs Without Loops

The function $\mathsf{wlp}\ \alpha\ Q$ is defined by *induction* over α. That is, we can make recursive calls to wlp, but only on constituents programs for α. The result, P, should be a pure formula as defined before (and, in particular, it should not contain any programs).

We go through the program constructs one by one, reminding ourselves of the axioms and then deriving from that a case in the definition of wlp. The key here is $\mathsf{wlp}\ \alpha\ Q \leftrightarrow [\alpha]Q$.

Sequential Composition. Recall from earlier in this lecture

$$[\alpha\ ;\ \beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

Blindly using equivalences:

$$\mathsf{wlp}\ (\alpha\ ;\ \beta)\ Q = \mathsf{wlp}\ \alpha\ (\mathsf{wlp}\ \beta\ Q)$$

This actually works! $\mathsf{wlp}\ \beta\ Q$ will be the weakest liberal precondition of Q with respect to β, and we can use this as the postcondition for the next recursive call on α because it must be pure.

An interesting aspect of this clause is that we proceed through the program from right to left: when computing the weakest precondition of $\alpha\ ;\ \beta$ we first compute it for β and then for α. This is characteristic of this approach. Going in the other direction is also possible, but it either leads us to the *strongest postcondition* (which we will not discuss) or the closely related *symbolic evaluation* (which is the subject of Lecture 8.

Assignment. For assignment, we will take particular advantage of the purity of the postcondition. First, let's recall the axiom:

$$[x := e]Q(x) \leftrightarrow \forall x'.\, x' = e \to Q(x') \quad (x' \text{ fresh})$$

where x' is chosen so it does not already occur in e or $Q(x)$. We need this side condition for two reasons. (1) if we have some previous knowledge about x (like: $x = 2$) then after an assignment (like: $x := 3$) we would reach inconsistent assumptions because they the two values of x would be in conflict. (2) We cannot just substitute e for x in $Q(x)$ because $Q(x)$ might contain programs. For example, in $[x := 5]([\textbf{while}\ x > 0\ x := x - 1])$ the variable x in the program successively refers to $5, 4, 3, 2, 1, 0$, so substituting in the initial value 5 ist just plain incorrect.

Fortunately, when calculating the weakest precondition we know that $Q(x)$ is a formula of pure arithmetic. Significantly, it does not contain any programs. Because of that, substituting e for x is actually not problematic. We write this as $Q(e)$. So we define

$$\mathsf{wlp}\ (x := e)\ Q(x) = Q(e)$$

Let's run through an example to see the two clauses in the definition of wlp in action. The program $z := x\ ;\ x := y\ ;\ y := z$ swaps the contents of variables x and y using the auxiliary variable z. Let's say the postcondition is $x = b \wedge y = a$. We expect the weakest precondition to imply that before the execution of the program, $x = a \wedge y = b$ must be true.

$$\begin{array}{ll} & \mathsf{wlp}\ (z := x\ ;\ x := y\ ;\ y := z)\ (x = a \wedge y = b) \\ = & \mathsf{wlp}\ (z := x)\ (\mathsf{wlp}\ (x := y\ \ ;\ y := z)\ (x = a \wedge y = b)) \\ = & \mathsf{wlp}\ (z := x)\ (\mathsf{wlp}\ (x := y)\ (\mathsf{wlp}\ (y := z)\ (x = a \wedge y = b))) \end{array}$$

At this point in rightmost call will use the rule for assignment, substituting z for y in the postcondition. The new constructed postcondition then feeds into the prior call to wlp, and so on.

$$\begin{array}{ll} = & \mathsf{wlp}\ (z := x)\ (\mathsf{wlp}\ (x := y)\ (x = a \wedge z = b)) \\ = & \mathsf{wlp}\ (z := x)\ (y = a \wedge z = b) \\ = & y = a \wedge x = b \end{array}$$

In this case, we get essentially *exactly* the precondition we expected. It does not mention z, since z is written to in the first assignment so its value prior to execution is irrelevant.

Conditionals. Recall the axiom

$$[\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q \leftrightarrow (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q)$$

Again, we can use this straightforwardly, making two recursive calls on subprograms.

$$\mathsf{wlp}\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta)\ Q = (P \rightarrow \mathsf{wlp}\ \alpha\ Q) \wedge (\neg P \rightarrow \mathsf{wlp}\ \beta\ Q)$$

The postcondition does not change in both calls and is therefore pure, while P is a program condition and therefore pure by our general assumption (2) from this lecture. Therefore, the result of wlp is pure.

Assertions and Tests. Again, the axioms:

$$[\mathbf{assert}\ P]Q \leftrightarrow (P \wedge Q)$$
$$[\mathbf{test}\ P]Q \leftrightarrow (P \rightarrow Q)$$

So:

$$\begin{array}{lcl} \mathsf{wlp}\ (\mathbf{assert}\ P)\ Q & = & P \wedge Q \\ \mathsf{wlp}\ (\mathbf{test}\ P)\ Q & = & P \to Q \end{array}$$

As in the case of conditionals, these results are pure because P is a program condition.

This leads us to the case of loops.

4 Loops

First, let's recall the right rule for $[\mathbf{while}]R$. Roughly, it states that the loop invariant J must hold initially, that it must be preserved by one trip around the loop, and that it must imply the postcondition.

$$\frac{\Gamma \vdash J, \Delta \quad J, P \vdash [\alpha]J \quad J, \neg P \vdash Q}{\Gamma \vdash [\mathbf{while}\ P\ \alpha]Q, \Delta}\ [\mathbf{while}]R$$

In applying this rule we have to choose the right loop invariant J. Since this a very difficult problem (both in theory and in practice), we will assume for the remainder of this lecture and part of the next lecture that the programmer has supplied it—maybe they were lucky enough to have taken 15-122 *Principles of Imperative Computation*! Our notation here is just $\mathbf{while}_J\ P\ \alpha$ where J is the loop invariant.

Another particularly tricky aspect of this rule is that neither Γ nor Δ are allowed to be used in the second and third premise. That's because Γ and Δ hold formulas that reference variables *in their state as we enter the loop*. However, the loop invariant must be preserved no matter how many times we go around the loop, so we cannot rely on any assumptions that holds as enter it. That's the same reason we cannot substitute an expression for a variable that appears in the loop.

So what to do? It turns out that we need a new logical connective to model this as a formula. We write $\Box P$ (pronounced "*white box* P") which is true exactly if P is valid (which means: true in every state, as we have defined when proving validity of our axioms). The right rule for $\Box P$ then wipes out any knowledge we might have about the current state.

$$\frac{\cdot \vdash P}{\Gamma \vdash \Box P, \Delta}\ \Box R$$

Semantically, it is also easy to define

$$\omega \models \Box P \quad \text{iff} \quad \nu \models P \quad \text{for all states } \nu$$

It is a useful exercise to show the soundness of the $\Box R$ rule given this definition. Unfortunately, it is not invertible. For example, we have $\bot \vdash \Box(\bot)$, but we cannot prove the premise of the rule $\cdot \vdash \bot$.

We won't discuss the left rule or axioms for $\Box P$, because we don't need them in the context of this course. The particular modal logic we need here is called S4,

and if you are curious you can read more about it in the Stanford Encyclopedia of Philosophy [Garson, 2000].

With this in hand, we can turn the $[\mathbf{while}]R$ rule into an axiom.

$$\begin{array}{lcll} [\mathbf{while}_J\ P\ \alpha]Q & \leftrightarrow & J & \text{(true initially)} \\ & & \wedge\ \Box(J \wedge P \rightarrow [\alpha]J) & \text{(preserved)} \\ & & \wedge\ \Box(J \wedge \neg P \rightarrow Q) & \text{(implies postcondition)} \end{array}$$

We have packaged up the *allowed antecedents* (like J and P) together with the succedent and then stashed under a white box in order to "erase" any other antecedents or succedents we might have.

We have written this as a bi-implication. If the loop invariant J were not specified in the syntax of the program, it would only be a right-to-left implication. Keeping this in mind, we can now turn this axiom into a definition of the weakest precondition.

$$\begin{array}{lcll} \mathsf{wlp}\ (\mathbf{while}_J\ P\ \alpha)\ Q & = & J & \text{(true initially)} \\ & & \wedge\ \Box(J \wedge P \rightarrow \mathsf{wlp}\ \alpha\ J) & \text{(preserved)} \\ & & \wedge\ \Box(J \wedge \neg P \rightarrow Q) & \text{(implies postcondition)} \end{array}$$

We should check a few things. First, are all postconditions in calls to wlp pure? There is only one recursive call, and its postcondition J must be pure because J appears in the program and was therefore assumed to be pure. Also, the formula returned by wlp is pure *if we allow* $\Box P$ *as a pure formula*. This seems reasonable since it does not contain any program. So we revise:

$$\begin{array}{lrcl} \text{Pure formulas} & P, Q & ::= & e_1 \leq e_2 \mid e_1 = e_2 \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid \neg P \mid \top \mid \bot \\ & & \mid & \Box P \end{array}$$

The theorem provers we use don't understand the white box modality, so we have to take care to turn these pure formulas into their language. We comment on that in Section 6.

We can also see that if the loop invariant is **not** preserved or does **not** imply the postcondition, the weakest liberal precondition is equivalent to falsehood ($\bot$) because the white boxed formula is always either true or false.

5 A Loop Example

Let's reconsider an example from Lecture 4 and Lecture 5.

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

We have already written in the loop invariant we discovered the first time. Our precondition was $x \geq 6$, but we purposely omit it here to see what the the weakest

liberal precondition might be. We expect that whatever it is should be *implied by* $x \geq 6$.

So we start to calculate (with $J = (x \geq 0)$).

$$\begin{aligned}&\mathsf{wlp}\ (\mathbf{while}_{x\geq 0}\ (x > 1)\ x := x - 2)\ (0 \leq x \leq 1)\\ &= x \geq 0\\ &\quad \wedge \Box\,(x \geq 0 \wedge x > 1 \rightarrow \mathsf{wlp}\ (x := x - 2)\ (x \geq 0))\\ &\quad \wedge \Box\,(x \geq 0 \wedge \neg\,(x > 1) \rightarrow 0 \leq x \leq 1)\end{aligned}$$

There is one recursive call to wlp, which we can work out by substitution (since it is an assignment):

$$\mathsf{wlp}\ (x := x - 2)\ (x \geq 0) = (x - 2 \geq 0)$$

Plugging this back in we get

$$\begin{aligned}&\mathsf{wlp}\ (\mathbf{while}_{x\geq 0}\ (x > 1)\ x := x - 2)\ (0 \leq x \leq 1)\\ &= x \geq 0\\ &\quad \wedge \Box\,(x \geq 0 \wedge x > 1 \rightarrow x - 2 \geq 0)\\ &\quad \wedge \Box\,(x \geq 0 \wedge \neg\,(x > 1) \rightarrow 0 \leq x \leq 1)\end{aligned}$$

Since we can verify the two boxed formulas as being valid, the weakest liberal precondition is equivalent to $x \geq 0$. As expected, this is implied by $x \geq 6$.

6 White Box[1]

First a note on substitution. When we write $Q(x)$ where Q may contain formulas $\Box P$, then occurrence of x in P are allowed, but excluded from substitution. That's because P has to be *valid*, which means true in every state. Therefore any occurrence of x in P does not refer to the same x as outside the white box. For example, if

$$Q(x) = (x > 1 \wedge \Box(x < 0 \rightarrow -x > 0))$$

then

$$Q(5) = (5 > 1 \wedge \Box(x < 0 \rightarrow -x > 0))$$

We might say that "substitution is blocked by white boxes". This is related to the fact that substitution into $[\alpha]P$ may be prohibited, in particular if α contains loops or assignments.

Can we translate a formula with white boxes into arithmetic without boxes? This is what we need in order to pass it to a theorem prover for arithmetic. It turns out to be quite easy: we "pull out" all white boxed formulas to the top level and replace them by $\top$ or $\bot$, depending on whether they can be verified or not.

[1] not covered in lecture, but important for the first lab

In the example from the previous section (not with the precondition), we'd like to verify

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

We calculate the weakest liberal precondition,

$$\begin{array}{l} \mathsf{wlp}\ (\mathbf{while}_{x\geq 0}\ (x > 1)\ x := x - 2)\,(0 \leq x \leq 1) \\ = x \geq 0 \\ \quad \wedge \Box\,(x \geq 0 \wedge x > 1 \to x - 2 \geq 0) \\ \quad \wedge \Box\,(x \geq 0 \wedge \neg\,(x > 1) \to 0 \leq x \leq 1) \end{array}$$

form the implication from the precondition, and pull out the white boxed formulas. We thus have to ask our theorem prover if all of the following are valid and plug in the results:

$$\begin{array}{lcl} p_1 & = & \mathsf{valid}\ (x \geq 0 \wedge x > 1 \to x - 2 \geq 0) \\ p_2 & = & \mathsf{valid}\ (x \geq 0 \wedge \neg\,(x > 1) \to 0 \leq x \leq 1) \\ \mathsf{valid}\ (x \geq 6 \to x \geq 0 \wedge p_1 \wedge p_2) & & \end{array}$$

If we determine the validity of p_1 and p_1 first (which will be true or false), we can then replace p_1 and p_2 with $\top$ or $\bot$ in the third line.

Fortunately, in this example, all of these are quite easy and valid. Note that we cannot just signal an error if p_1 or p_2 are invalid. For example, in

$$\mathsf{wlp}\ (\mathbf{if}\ \top\ \mathbf{then}\ x := x + 1\ \mathbf{else}\ (\mathbf{while}_{x=0}\ (x \geq 0)\ x := x - 1))\,(x = 5)$$

the loop invariant $x = 0$ is not preserved, but, logically, the weakest liberal precondition should be $x = 4$ since the implication $\neg\top \to \mathsf{wlp}\ (\mathbf{while}\ldots)\,(x = 5)$ is valid, even if $\mathsf{wlp}\ (\mathbf{while}\ldots)\,(x = 5) = \bot$.

Alternatively we could stipulate that loop invariants must be loop invariants wherever they occur and just abort with an error when given a program such as the one above.

7 Summary

We summarize the definition of $\mathsf{wlp}\ \alpha\ Q$.

$$\begin{array}{lcl} \mathsf{wlp}\ (\alpha\ ;\ \beta)\ Q & = & \mathsf{wlp}\ \alpha\ (\mathsf{wlp}\ \beta\ Q) \\ \mathsf{wlp}\ (x := e)\ Q(x) & = & Q(e) \\ \mathsf{wlp}\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta)\ Q & = & (P \to \mathsf{wlp}\ \alpha\ Q) \wedge (\neg P \to \mathsf{wlp}\ \beta\ Q) \\ \mathsf{wlp}\ (\mathbf{assert}\ P)\ Q & = & P \wedge Q \\ \mathsf{wlp}\ (\mathbf{test}\ P)\ Q & = & P \to Q \\ \mathsf{wlp}\ (\mathbf{while}_J\ P\ \alpha)\ Q & = & J \\ & & \wedge \Box(J \wedge P \to \mathsf{wlp}\ \alpha\ J) \\ & & \wedge \Box(J \wedge \neg P \to Q) \end{array}$$

References

James Garson. Modal logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Spring 2024 edition edition, 2000. URL https://plato.stanford.edu/archives/spr2024/entries/logic-modal/.

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:

$$[\alpha\,;\,\beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

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:

$$[\alpha\,;\,\beta]S \leftrightarrow [\alpha]([\beta]S)$$

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

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

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.

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

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

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

Similarly, assignment remains the same.

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

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.

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

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:

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

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

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

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

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

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

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 do 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.

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

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.

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

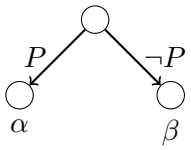
The first premise is pure, but the other two are not by contain stack formulas, so we remain within the algorithmic rules ($\Vdash$).

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.

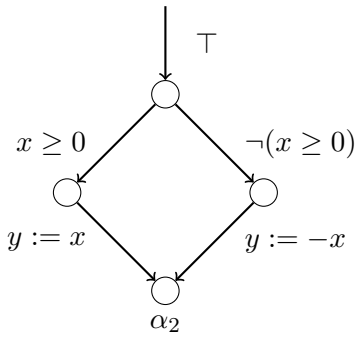
For our purposes, we use an especially simple form. Small circles denote points in the program, and arrows indicate possible transitions from one point in the program to the next. On the side of the arrow we indicate the *information gained* for symbolic evaluation along this particular transition. For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



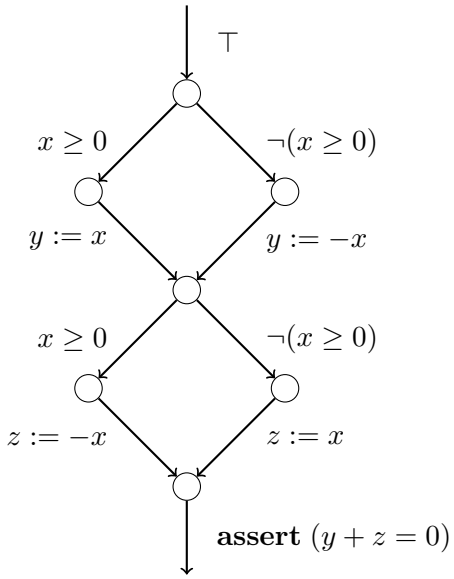
A precondition P is drawn as a label on the incoming edge to the root. Starting our example,

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

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.

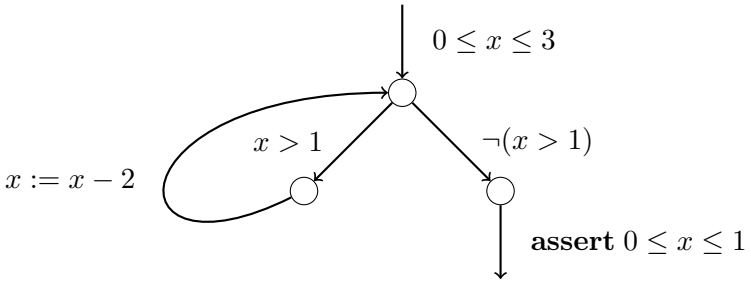
In some circumstances, even bounded checking could amount to full verification. That's when the paths around the loop become infeasible before the bound is reached. Sequent derivations are large and awkward to show, so we use a control flow graph instead for the (by now familiar) program

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

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

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.

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

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:

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

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

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

does not include x. Therefore, a program such as

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

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

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

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

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

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.

Lecture Notes on
Beyond Safety Properties

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

Lecture 10
September 26, 2024

1 Introduction

Recall that a *trace* of a program is the (potentially infinite) sequence of states that make up its computation. A *safety property* of a trace is defined as one whose violation can be determined from a finite prefix. A *liveness property* is one whose violation may depend on the whole infinite trace. Operations such as division by zero or out-of-bounds memory access are examples of safety properties. We can prove safety in dynamic logic via propositions of the form $P \to [\alpha]\top$. Examples of liveness properties would be that a server responds to a query or that a lock acquired in a concurrent computation is eventually released. We can prove liveness properties in dynamic logic via propositions of the form $P \to \langle\alpha\rangle Q$ (pronounced "*diamond alpha Q*"). Recall that the formula $\langle\alpha\rangle Q$ is true if there is a way to reach a final state such that Q is true. This implies that, among other things, loops appearing in the computation must be proved terminating. We have deemphasized the diamond modality, not investigating its properties.

There are techniques for transforming liveness properties into safety properties. For example, we can require that a server respond within a certain number of steps or milliseconds. However, it may still be difficult to enforce such transformed liveness properties, and it may be even more difficult to take appropriate corrective action.

Today we will start analyzing an important class of security policies, called *information flow policies*, that go beyond both safety and liveness properties in the sense that we cannot determine if they are violated by analyzing a single program trace.

2 Information Flow, Informally

When you log in to your favorite banking site, you would like to be able to see information about your own account, but you should not be able to see anyone else's. In other words, we don't want information to flow from other accounts to the program serving you. In our small imperative language we model this using *high security variables* and *low security variables*. Reading from a high security variable and writing the value to a low security variable would be a violation of our information flow policy. It would be slightly more realistic to consider reading and writing from memory, but it would be more complex without changing the fundamental ideas we study.

As a small running example we consider the following program.

$$\begin{array}{l} x := 1 \ ; \\ y := x + 5 \ ; \\ z := y - 1 \end{array}$$

We consider x to be a high security variables, while y and z are classified as low security. We write this information flow policy as

$$x : \mathsf{H}, y : \mathsf{L}, z : \mathsf{L}$$

Intuitively, the program above would not satisfy our security policy: we read from x (high security) and then write $x + 5$ to y, which is low security. In fact, we can exactly recover x as $y - 5$, so we gain perfect information about a secret.

Here is a trace of this program, assuming all variables initially have value 0.

$$\begin{array}{rl} & (x = 0, y = 0, z = 0) \\ \Rightarrow & (x = 1, y = 0, z = 0) \\ \Rightarrow & (x = 1, y = 6, z = 0) \\ \Rightarrow & (x = 1, y = 6, z = 5) \end{array}$$

Now consider the following alternative program shown on the right.

$$\begin{array}{l|l|l} (x = 0, y = 0, z = 0) & x := 1 \ ; & x := 1 \ ; \\ (x = 1, y = 0, z = 0) & y := x + 5 \ ; & y := 6 \ ; \\ (x = 1, y = 6, z = 0) & z := y - 1 & z := 5 \\ (x = 1, y = 6, z = 5) & & \end{array}$$

We see that both programs have *exactly the same trace*, but the first one violates the policy (information flows from x to y and then to z) while no information flows at all on the right.

This shows that information flow is not a property of a single trace of a program, but requires something more. We'll see what in the next lecture. Meanwhile, we'll try to intuit a program analysis that ensures adherence to an information flow policy. In the next lecture, we will check if it accomplishes what a semantic definition of information flow demands.

3 Tracking Security Levels

We now consider ways analyze programs with the goal of proving that a given information flow policy is respected by a program. Because we haven't rigorously defined what that is, the remainder of this lecture is rather speculative. In the next lecture we will nail it down precisely.

For now, we imagine a security policy is given by an assignment of security levels to variables, like H for *high* and L for *low*. We use Σ as a map from variables to security levels. We write $x : \ell$ if the variable x has security level ℓ. Our system of inference rules derive

$$\Sigma \vdash \alpha \text{ secure}$$

which expresses that, given the security policy Σ, the program α is secure. We name the inference rules as $nameT$, where T stands for *taint* (see Section 6).

Assignment. At the root of the system is that an assignment $x := e$ is a violation of the security policy if $x : \mathsf{L}$ and $e : \mathsf{H}$. The security level of an expression is the maximal level of the variables occurring in it. That is, we also define $\Sigma \vdash e : \ell$, meaning that expression e has security level ℓ.

Variables just have the level prescribed by the policy, and constants are always of low security.

$$\frac{\Sigma(x) = \ell}{\Sigma \vdash x : \ell}\ \mathsf{var}T \qquad \frac{}{\Sigma \vdash c : \mathsf{L}}\ \mathsf{const}T$$

For a binary operator, we take the maximal security level of the constituents, where $\mathsf{H} > \mathsf{L}$.

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2 \quad \ell = \max(\ell_1, \ell_2)}{\Sigma \vdash e_1 + e_2 : \ell}\ +T$$

For an assignment, there are several possible combinations. The first: we can always write to a high security variable, since this does not represent a flow from high to low. An example of this could be appending to a (secure) log file using a low-security value.

$$\frac{\Sigma(x) = \mathsf{H}}{\Sigma \vdash x := e \text{ secure}}\ :=T_1$$

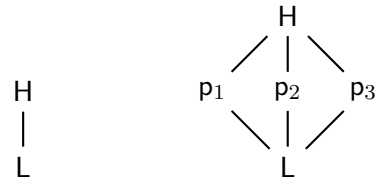
If x is of low security, then e also has to be (or we fail).

$$\frac{\Sigma(x) = \mathsf{L} \quad \Sigma \vdash e : \mathsf{L}}{\Sigma \vdash x := e \text{ secure}}\ :=T_2 \qquad \frac{\textbf{no rule for} \quad \Sigma(x) = \mathsf{L} \quad \Sigma \vdash e : \mathsf{H}}{\Sigma \vdash x := e \text{ secure}}$$

4 A Lattice of Security Levels

Before we go further, we generalize the security levels from just two (high and low) to potentially multiple ones. We imagine them being arranged in a *lattice*, where information is allowed to flow upwards but not downwards. I think that technically we just need a *join-semilattice*, although different authors make slightly different assumptions.

Below are two examples. This first just has the high and low security levels we have been using. The second is one where we imagine three principals, p_1, p_2, and p_3, say bank account holders. They cannot see high security values (level H) and they cannot see each other's data since information can only flow up but not down. The lowest level L is "public" (anyone can see it).



A join-semilattice is defined by a partial order $\ell_1 \sqsubseteq \ell_2$ (ℓ_1 is a lower security level than ℓ_2) and the operation of $\ell_1 \sqcup \ell_2$ (the *least upper bound* of two security levels). We also need a least element $\bot$ which is the unit of $\sqcup$ and is below every other level. We will not go into details regarding all the algebraic laws of the semilattice, but here are some from properties of the least upper bound and the least element.

$$\bot \sqcup \ell = \ell \sqcup \bot = \ell$$

$$\bot \sqsubseteq \ell$$

$$\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$$
$$\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$$
$$\ell_1 \sqsubseteq \ell \text{ and } \ell_2 \sqsubseteq \ell \text{ implies } \ell_1 \sqcup \ell_2 \sqsubseteq \ell$$

We can generalize the rules so far from two levels to a lattice of security levels.

$$\frac{\Sigma(x) = \ell}{\Sigma \vdash x : \ell}\ \mathsf{var}T \qquad \frac{}{\Sigma \vdash c : \bot}\ \mathsf{const}T \qquad \frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2 \quad \ell = \ell_1 \sqcup \ell_2}{\Sigma \vdash e_1 + e_2 : \ell}\ +T$$

$$\frac{\Sigma \vdash e : \ell \quad \ell \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e\ \mathsf{secure}}\ :=T$$

The last rule expresses succinctly that information can only flow from lower to higher levels of security, and not in any other circumstances.

5 Tracking Security Levels, Continued

With assignment specified, we move on to other language constructs.

Sequential Composition. We check both subprograms independently with respect to the same security policy Σ.

$$\frac{\Sigma \vdash \alpha \text{ secure} \quad \Sigma \vdash \beta \text{ secure}}{\Sigma \vdash \alpha \,;\, \beta \text{ secure}} \,;T$$

At this point we have enough to check that one of our two example programs is secure while the other one is not. We use the two-element lattice with $\mathsf{H} \sqsupset \mathsf{L}$ and the security policy

$$\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L}, z : \mathsf{L})$$

We construct the following derivation bottom-up, failing at the second assignment as expected. We elide some rule names for the sake of brevity.

$$\dfrac{\dfrac{\dfrac{}{\Sigma_0 \vdash 1 : \mathsf{L}} \quad \mathsf{L} \sqsubseteq \Sigma_0(x) = \mathsf{H}}{\Sigma_0 \vdash x := 1 \text{ secure}} :=T \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Sigma_0(x) = \mathsf{H}}{\Sigma_0 \vdash x : \mathsf{H}} \quad \dfrac{}{\Sigma_0 \vdash 5 : \mathsf{L}} \quad \mathsf{H} = \mathsf{H} \sqcup \mathsf{L}}{\Sigma_0 \vdash x + 5 : \mathsf{H}} +T \qquad \text{fails} \quad \mathsf{H} \sqsubseteq \Sigma_0(y) = \mathsf{L}}{\Sigma_0 \vdash y := x + 5 \text{ secure}} :=T \qquad \dots}{\Sigma_0 \vdash (y := x + 5 \,;\, z := y - 1) \text{ secure}} \,;T}{\Sigma_0 \vdash (x := 1 \,;\, y := x + 5 \,;\, z := y - 1) \text{ secure}} \,;T$$

Conditionals. Conditionals are interesting. The condition may have a security level, but we ignore that for now because it doesn't perform an assignment.

$$\frac{\Sigma \vdash \alpha \text{ secure} \quad \Sigma \vdash \beta \text{ secure}}{\Sigma \vdash \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \text{ secure}} \,\textbf{if}T$$

Loops. Loops are similar to conditionals.

$$\frac{\Sigma \vdash \alpha \text{ secure}}{\Sigma \vdash \textbf{while } P \ \alpha \text{ secure}} \,\textbf{while}T$$

6 Taint Analysis

The rules we have so far can be used for *taint checking*. We think of high security variables as being sources of taint and we track how their values are propagated throughout a program. If a tainted value reaches a variable that is of low security,

the program can be rejected or aborted as insecure. This can be done statically (so insecure programs are never executed) or dynamically (say, with an extra taint bit attached to memory locations or values).

If a tainted value reaches a low-security value, we definitely have a violation of the (for now informal) security policy. We don't even have to declare the security level of all variables, because the analysis can infer them. For more on taint analysis, we recommend Schwartz et al. [2010].

However, there are some obvious situations where a flow of information does occur, but taint analysis will not discover it. We recommend you think about possible holes in the system before moving on.

7 Indirect Flows

Consider the following simple program, where $x : \mathsf{H}$ and $y : \mathsf{L}$.

$$\textbf{if } x = 0 \textbf{ then } y := 1 \textbf{ else } y := 0$$

With our policy so far, this program checks! Both assignments to y are with constants, which have low security level. It should be clear that, intuitively, information (illegally!) flows from x to y because x is tested in the condition, and x has a high security level.

In order to track this kind of *indirect flow* we need to somehow "remember" which tests we have performed to get to the current point in the program, and whether these tests involved high security variables. For example, the program above would be safe if both x and y were of the same security level (whether high or low).

Our solution is based on the seminal paper by Volpano et al. [1996], although the details of the formalization vary.

For that purpose we introduce a *ghost variable* named pc. It is called a ghost variable because it is not allowed to appear in the program, only in our analysis rules. We usually assume that the program starts executing at the lowest level of security ($\bot$ in general, in many of our examples L). Let's get right to the critical rule. We now call the rules $name F$, where F suggests *flow*.

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \text{ secure} \quad \Sigma' \vdash \beta \text{ secure}}{\Sigma \vdash \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \text{ secure}} \textbf{if} F$$

Let's tease apart what's in this rule. First, we determine the security level of P (which will end up being the least upper bound of all variables occurring in P). If the current program is at the lowest security level, that would be the level in which we have to check α and β. But if we have already branched on conditions before, we might have to pick a higher one. So $\ell' = \Sigma(pc) \sqcup \ell$ is the least upper bound of the current pc and the test P. We update the security level of the pc to ℓ' and then check the branches.

In our example, we have $x : \mathsf{H}, y : \mathsf{L} \vdash x = 0 : \mathsf{H}$, so both branches are checked with $pc : \mathsf{H}$.

Now we have to reconsider the other constructs to take the pc into account.

Assignment. For assignment $x := e$, we have to take the least upper bound of the level of e and level of pc and compare the result to the level of x. This rules out our motivating example, as it should.

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \text{ secure}} :=F$$

Sequential Composition. It seems like as we proceed through the program the security level of the pc keeps going up, as in the rule for conditionals. Does it ever go down? Not explicitly, but when we have processed both branches of a conditional and proceed with the program that follows it, the security level implicitly reverts to what it was before.

$$\frac{\Sigma \vdash \alpha \ \mathsf{secure} \quad \Sigma \vdash \beta \ \mathsf{secure}}{\Sigma \vdash \alpha \ ; \ \beta \ \mathsf{secure}} \ ;F$$

For example, the following program is secure. Even though the assignments to y are checked at a high security level (with $pc : \mathsf{H}$), the assignment to z is checked with $pc : \mathsf{L}$.

$$\begin{array}{l}(x : \mathsf{H}, y : \mathsf{H}, z : \mathsf{L}, pc : \mathsf{L}) \\ x := 1 \ ; \\ \textbf{if } x = 0 \textbf{ then } y := 1 \textbf{ else } y := 0 \ ; \\ z := 5\end{array}$$

Loops. Loops are similar to conditionals in the sense that we may have to upgrade the security level of the pc in the loop body. Somehow we lose information about the failed test when we exit the loop. We have to see if this is really sound in the next lecture. Let's mark it as suspicous for now.

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \ \mathsf{secure}}{\Sigma \vdash \textbf{while } P \ \alpha \ \mathsf{secure}} \ \textbf{while}F \ ?$$

Tests. Tests also seem a bit strange. Do we need to check that the security level of P is lower than the pc or not?

$$\frac{\Sigma \vdash P : \ell \quad \ell \sqsubseteq \Sigma(pc)}{\Sigma \vdash \textbf{test } P \ \mathsf{secure}} \ \textbf{test}F \ ?$$

Formulas. The security level of a formula is easy to determine. We show some sample rules. In the rule $\top F$ the "$\bot$" is the least element of the lattice, not falsehood (an unfortunate clash of notation).

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash e_1 \leq e_2 : \ell_1 \sqcup \ell_2} \leq F \qquad \frac{}{\Sigma \vdash \top : \bot} \ \top F \qquad \frac{\Sigma \vdash P : \ell_1 \quad \Sigma \vdash Q : \ell_2}{\Sigma \vdash P \wedge Q : \ell_1 \sqcup \ell_2} \wedge F$$

Let's read the last rule:

> *The formula $P \wedge Q$ has security level $\ell_1 \sqcup \ell_2$ if P has security level ℓ_1 and Q has security level ℓ_2.*

If we wrote this as a function seclev, it would something like

$$\mathsf{seclev}\ \Sigma\ (P \wedge Q) = \mathsf{lub}\ (\mathsf{seclev}\ \Sigma\ P)\ (\mathsf{seclev}\ \Sigma\ Q)$$

References

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Symposium on Security and Privacy (2010)*, pages 317–331, Oakland, California, May 2010. IEEE.

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

Lecture Notes on Information Flow

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

Lecture 11
October 1, 2024

1 Introduction

In the last lecture we informally introduced a notion of *information flow* and a type system to ensure *confidentiality*, meaning that high security data do not flow to low security variables. The rules are summarized in Section 5. They express an information flow policy as a mapping Σ from variables to security levels, arranged in a semilattice, including a ghost variable pc to track *implicit flows* due to conditional tests. We often worked with the two element lattice with levels H (for *high*) and L (for *low*) with $\mathsf{L} \sqsubset \mathsf{H}$.

We assumed that an attacker can see the program and control or see the values of all low security variables before and after computation. We also assumed the attacker can not observe nontermination (including abort after a failed test); some of the rules would have to be changed to account for that and still capture confidentiality.

These rules are *syntactic* and it is straightforward to see that they define an algorithm for deriving that a program α is secure with respect to a security policy Σ that assigns security levels.

What was missing was a formal *semantic* definition of information flow security and a proof that the rules are *sound* with respect to this definition. That's the goal of today's lecture. The material is adapted mostly from Volpano et al. [1996].

2 Noninterference

We have already seen that information flow is not a property of a single trace. Simplifying the example further, consider the security policy $\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L})$, the initial state $x = 0, y = 0$ and the programs

$$y := x + 1 \qquad\qquad y := 1$$

The traces are the same, consisting of just one transition

$$(x = 0, y = 0) \Rightarrow (x = 0, y = 1)$$

The program on the left violates our policy, flowing information from x to y to the extent an attacker can fully recover the initial value of x from the value of y in the final state. The program on the right permits no such inference.

If we have just two security levels (L and H) we'd like to formalize our attacker model using a notion of *observation*. We consider the program secure if two initial states are equivalent as far as the attacker can see, then the final states must also be equivalent as far as the attacker can see. If that were not the case, the attacker may be able to make an inference about the secret (hidden) part of the initial state.

We define when two states ω_1 and ω_2 are *observationally equivalent at level* L *with respect to security policy* Σ, written as $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$:

> We define $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ iff $\omega_1(x) = \omega_2(x)$ for all x such that $\Sigma(x) = \mathsf{L}$.

In our tiny example (with $\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L})$), we have (among others):

$$\begin{array}{lcl} \Sigma_0 \vdash (x = 0, y = 0) & \approx_{\mathsf{L}} & (x = 0, y = 0) \\ \Sigma_0 \vdash (x = 0, y = 0) & \approx_{\mathsf{L}} & (x = 1, y = 0) \\ \Sigma_0 \vdash (x = 0, y = 1) & \not\approx_{\mathsf{L}} & (x = 0, y = 2) \\ \Sigma_0 \vdash (x = 0, y = 1) & \not\approx_{\mathsf{L}} & (x = 1, y = 2) \end{array}$$

Next we define that a program α satisfies *noninterference* if values of high security variables do not affect the outcome as it may be observed at low security. The outcome here is simply the result of evaluation (as defined in Lecture 9).

> We define that program α satisfies *noninterference with respect to policy* Σ
> iff for all ω_1, ω_2, ν_1, and ν_2,
> $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$, $\mathsf{eval}\ \omega_1\ \alpha = \nu_1$, and $\mathsf{eval}\ \omega_2\ \alpha = \nu_2$ implies $\Sigma \vdash \nu_1 \approx_{\mathsf{L}} \nu_2$.

Since evaluation is deterministic, given ω_1 and ω_2, there will be at most one pair ν_1 and ν_2. Noninterference does not talk about the case where there is no final state, which is why this condition is called *termination-insensitive noninterference*.

Let's apply it to our example. We want to show that the first program, $y := x+1$ does **not** satisfy noninterference. That is, we need to find to states, indistinguishable at level L that has distinguishable outcomes. So we pick

$$\omega_1 = (x = 0, y = 0) \text{ and } \omega_2 = (x = 1, y = 0)$$

As we already determined, we have $\Sigma_0 \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$. We further have

$$\begin{array}{lcl} \mathsf{eval}\ \omega_1\ (y := x + 1) & = & (x = 0, y = 1) \\ \mathsf{eval}\ \omega_2\ (y := x + 1) & = & (x = 1, y = 2) \end{array}$$

and $\Sigma_0 \vdash (x = 0, y = 1) \not\approx_{\mathsf{L}} (x = 1, y = 2)$ because the low-observable outcomes for y are different.

On the other hand, for any two (relevant) states that are low-observably equivalent

$$\omega_1 = (x = a, y = b) \text{ and } \omega_2 = (x = a', y = b)$$

we have

$$\begin{array}{rcl} \mathsf{eval}\ \omega_1\ (y := 1) & = & (x = a, y = 1) \\ \mathsf{eval}\ \omega_2\ (y := 1) & = & (x = a', y = 1) \end{array}$$

and

$$\Sigma_0 \vdash (x = a, y = 1) \approx_{\mathsf{L}} (x = a', y = 1)$$

So the program $y := 1$ *does* satisfy noninterference.

By analogous reasoning,[1] the program $y := (x - x) + 1$ also satisfies noninterference, even though our type system would reject it. This is the first example showing the *incompleteness* of the proposed type system with respect to information flow. Static type systems often have to make *sound* (and decidable) approximations to some underlying semantic property (which is often undecidable).

Before we move on, we generalize the noninterference property to an arbitrary semilattice of security levels. First, observations are restricted to all levels *below* a given security level ℓ:

> We define $\Sigma \vdash \omega_1 \approx_\ell \omega_2$ iff $\omega_1(x) = \omega_2(x)$ for all x such that $\Sigma(x) \sqsubseteq \ell$.

From this, semantic security with respect to an information flow policy Σ generalizes the previous definition in a straightforward way.

> We define $\Sigma \models \alpha$ secure iff for all ω_1, ω_2, ν_1, ν_2, and ℓ
> $\Sigma \vdash \omega_1 \approx_\ell \omega_2$, $\mathsf{eval}\ \omega_1\ \alpha = \nu_1$, and $\mathsf{eval}\ \omega_2\ \alpha = \nu_2$ implies $\Sigma \vdash \nu_1 \approx_\ell \nu_2$.

The soundness property now states:

> If $\Sigma \vdash \alpha$ secure then $\Sigma \models \alpha$ secure

The completeness property would go in the other direction, but it does not hold as the small example $y := (x - x) + 1$ from above shows. We will have another example at the end of this lecture.

3 Read Levels of Expressions and Formulas

We will need some properties of expressions and formulas. Usually, we would wait until we find we need them in proving the main theorem (in this case, soundness), but we show them first because they bring up the important proof technique of *rule induction*.

[1] we did not discover this in lecture

The first is that expressions read only below their security level. The proof is by rule induction, which means we have to show that all rules preserve the stated property: if we assume it for all premises then it must hold for the conclusion. We have already implicitly used it when proving soundness of inference rules (for example, for sequent calculus for propositional logic and dynamic logic). The reason that every sequent we can derive is valid is that each inference rule preserves validity.

Lemma 1 (Expression Read Level) *If* $\Sigma \vdash e : \ell$ *then for every* $x \in \mathsf{use}\ e$, $\Sigma(x) \sqsubseteq \ell$.

Proof: By rule induction on the derivation of $\Sigma \vdash e : \ell$. We consider each case in turn.

Case:

$$\frac{}{\Sigma \vdash c : \bot}\ \mathsf{const}F$$

Then $\mathsf{use}\ c = \{\,\}$ and the property is vacuously true.

Case:

$$\frac{\Sigma(x) = \ell}{\Sigma \vdash x : \ell}\ \mathsf{var}F$$

Then $\mathsf{use}\ x = \{x\}$ and $\Sigma(x) = \ell$, so our property is satisfied by reflexivity ($\Sigma(x) = \ell \sqsubseteq \ell$).

Case:

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash e_1 + e_2 : \ell_1 \sqcup \ell_2}\ +F$$

We have

for all $x \in \mathsf{use}\ e_1$, $\Sigma(x) \sqsubseteq \ell_1$	(by ind. hyp.)
for all $x \in \mathsf{use}\ e_2$, $\Sigma(x) \sqsubseteq \ell_2$	(by ind. hyp.)
$x \in \mathsf{use}\ (e_1 + e_2)$	(assumption)
$x \in \mathsf{use}\ e_1$ or $x \in \mathsf{use}\ e_2$	(by defn. of use)
$\Sigma(x) \sqsubseteq \ell_1$ or $\Sigma(x) \sqsubseteq \ell_2$	(from uses of the ind. hyp.)
$\Sigma(x) \sqsubseteq \ell_1 \sqcup \ell_2$	(by property of $\sqcup$)

□

The next lemma has the same kind of proof, so we won't bother writing it out.

Lemma 2 (Formula Read Level) *If* $\Sigma \vdash P : \ell$ *then for every* $x \in \mathsf{use}\ P$, $\Sigma(x) \sqsubseteq \ell$.

Proof: By rule induction, as in the proof of Lemma 1. □

4 Soundness of the Information Flow Type System

We want to prove that $\Sigma \vdash \alpha$ secure then $\Sigma \models \alpha$ secure. As we might expect from the proofs in the preceding section, this should follow by rule induction on $\Sigma \vdash \alpha$ secure.

Before doing this more rigorously, we examine assignment, one of the base cases.

Theorem 3 (Soundness of Information Flow Types) *If* $\Sigma \vdash \alpha$ secure *then* $\Sigma \models \alpha$ secure

Proof: By rule induction on $\Sigma \vdash \alpha$ secure. We'll need one more lemma, which we state and prove later for pedagogical purposes.

Case:

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \text{ secure}} :=F$$

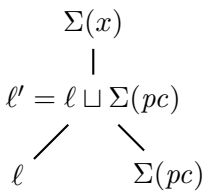
We have to show that $\Sigma \models x := e$ secure. We have the following setup:

$\Sigma \vdash \omega_1 \approx_k \omega_2$	(assumption)
eval ω_1 $(x := e) = \nu_1$	(assumption)
eval ω_2 $(x := e) = \nu_2$	(assumption)
...	
$\Sigma \vdash \nu_1 \approx_k \nu_2$	(to show)

By the rules for evaluation we obtain

$\nu_1 = \omega_1[x \mapsto c_1]$ for $c_1 = \text{eval}_{\mathbb{Z}}\ \omega_1\ e$	(by defn. of eval)
$\nu_2 = \omega_2[x \mapsto c_2]$ for $c_2 = \text{eval}_{\mathbb{Z}}\ \omega_2\ e$	(by defn. of eval)

Let's also depict what we know about the security levels.



At this point we distinguish two cases: $\Sigma(x) \sqsubseteq k$ and $\Sigma(x) \not\sqsubseteq k$.

If $\Sigma(x) \sqsubseteq k$ then also $\ell \sqsubseteq k$ by transitivity. Since $\Sigma \vdash e : \ell$ and $\Sigma \vdash \omega_1 \approx_k \omega_2$ we conclude by Lemma 1 that $c_1 = c_2$ and so $\Sigma \vdash \omega_1[x \mapsto c_1] \approx_k \omega_2[x \mapsto c_2]$.

If $\Sigma(x) \not\sqsubseteq k$ then $\Sigma \vdash \omega_1[x \mapsto c_1] \approx_k \omega_2[x \mapsto c_2]$ because x is not observable at level k and $\Sigma \vdash \omega_1 \approx_k \omega_2$.

Case:

$$\frac{\Sigma \vdash \alpha \text{ secure} \quad \Sigma \vdash \beta \text{ secure}}{\Sigma \vdash \alpha \,;\, \beta \text{ secure}} \,;F$$

We set up this case:

$\Sigma \vdash \omega_1 \approx_k \omega_2$ (assumption)
$\mathsf{eval}\ \omega_1\ (\alpha \,;\, \beta) = \nu_1$ (assumption)
$\mathsf{eval}\ \omega_2\ (\alpha \,;\, \beta) = \nu_2$ (assumption)
...
$\Sigma \vdash \nu_1 \approx_k \nu_2$ (to show)

We can reason with the definition of eval.

$\mathsf{eval}\ \omega_1\ \alpha = \mu_1$ and $\mathsf{eval}\ \mu_1\ \beta = \nu_1$ for some μ_1 (by defn. of eval)
$\mathsf{eval}\ \omega_2\ \alpha = \mu_2$ and $\mathsf{eval}\ \mu_2\ \beta = \nu_2$ for some μ_2 (by defn. of eval)

Next, by induction hypothesis $\Sigma \models \alpha$ secure and together with the assumption about $\Sigma \vdash \omega_1 \approx_k \omega_2$ we get

$\Sigma \vdash \mu_1 \approx_k \mu_2$ (by ind. hyp. on first premise)

This is the fact we need to apply the induction hypothesis on the security of β, because β is evaluated in states μ_1 and μ_2.

$\Sigma \vdash \nu_1 \approx_k \nu_2$ (by ind. hyp. on second premise)

Fortunately, this is just what we needed to show.

Case:

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \text{ secure} \quad \Sigma' \vdash \beta \text{ secure}}{\Sigma \vdash \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \text{ secure}} \,\textbf{if}F$$

This is the most difficult case since it involves implicit flows and therefore the ghost variable pc. It will require a lemma. But first we set up:

$\Sigma \vdash \omega_1 \approx_k \omega_2$ (assumption)
$\mathsf{eval}\ \omega_1\ (\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta) = \nu_1$ (assumption)
$\mathsf{eval}\ \omega_2\ (\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta) = \nu_2$ (assumption)
...
$\Sigma \vdash \nu_1 \approx_k \nu_2$ (to show)

Here is what we know about the security levels in the rule.

$$\begin{array}{ccc} \ell' = \ell \sqcup \Sigma(pc) & & \\ \diagup & \diagdown & \\ \ell & & \Sigma(pc) \end{array}$$

Again, we distinguish two cases: $\ell' \sqsubseteq k$ and $\ell' \not\sqsubseteq k$.

If $\ell' \sqsubseteq k$ then also $l \sqsubseteq k$ and $\mathsf{eval}_{\mathbb{B}}\ \omega_1\ P = \mathsf{eval}_{\mathbb{B}}\ \omega_2\ P = b$ for some Boolean b by Lemma 2. If $b = \top$ we can apply the induction hypothesis to the first premise and the evaluations $\mathsf{eval}\ \omega_1\ \alpha = \nu_1$ and $\mathsf{eval}\ \omega_2\ \alpha = \nu_2$.

If $\ell' \not\sqsubseteq k$ then $\mathsf{eval}_{\mathbb{B}}\ \omega_1\ P$ might be different from $\mathsf{eval}_{\mathbb{B}}\ \omega_2\ P$ and the different branches might be taken. In that case we need to prove that, nevertheless, $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_k \mathsf{eval}\ \omega_2\ \beta$.

At this point we can only conclude that because the security level of the pc is increased to ℓ'. The salient property is that if $\Sigma' \vdash \alpha\ \mathsf{secure}$ and $\Sigma'(pc) = \ell'$, then α *will only write to variables with security level above* ℓ'. And the same for β. (These are instances of Lemma 4.) Fortunately, $\ell' \not\sqsubseteq k$, so none of the writes of α and β will be observable at level k or below and $\Sigma \vdash \nu_1 \approx_k \nu_2$.

Case:

$$\frac{}{\Sigma \vdash \mathbf{test}\ P\ \mathsf{secure}}\ \mathbf{test}F$$

We set up:

$$\begin{array}{ll} \Sigma \vdash \omega_1 \approx_k \omega_2 & \text{(assumption)} \\ \mathsf{eval}\ \omega_1\ (\mathbf{test}\ P) = \nu_1 & \text{(assumption)} \\ \mathsf{eval}\ \omega_2\ (\mathbf{test}\ P) = \nu_2 & \text{(assumption)} \\ \ldots & \\ \Sigma \vdash \nu_1 \approx_k \nu_2 & \text{(to show)} \end{array}$$

By definition of eval we know that P must evaluate to true in both ω_1 and ω_2 and $\nu_1 = \omega_1$ and $\nu_2 = \omega_2$. So the desired conclusion follows immediately from the strong assumption that a poststate actually exists.

Case:

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha\ \mathsf{secure}}{\Sigma \vdash \mathbf{while}\ P\ \alpha\ \mathsf{secure}}\ \mathbf{while}F$$

This case is similar to the case for conditional and we leave it to the reader. Writing it out is a good test of your understanding!

□

What remains is the *confinement lemma* relating the security level of the ghost variable pc to the writing behavior of the program. We define maydef α as the set of variables that a program may assign a value to—just all the left-hand sides of assignments in α. This can include more variables than the set def α which includes only those variables that α must write to.

Lemma 4 (Confinement) *If* $\Sigma \vdash \alpha$ secure *then for every* $x \in$ maydef α, $\Sigma(pc) \sqsubseteq \Sigma(x)$.

Proof: By rule induction on $\Sigma \vdash \alpha$ secure. The key observation is that for an assignment $x := e$ the rules require that $\Sigma(pc) \sqsubseteq \Sigma(x)$ and that the other rules only maintain or raise the level. We elide the case-by-case detail. □

We conclude the lecture with another example that, according to the definition, satisfies noninterference but we cannot derive that within the type system.

$$\textbf{if } x = 0 \textbf{ then } y := 1 \textbf{ else } y := 1$$

In the type system this fails with $\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L})$ because the assignments in the two branches are to low-security variables. It is nevertheless secure because the two values of y are the same, so the observable outcomes at low level are equal.

5 Summary: Information Flow Type System

References

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

$\boxed{\Sigma \vdash e : \ell}$

$$\frac{\Sigma(x) = \ell}{\Sigma \vdash x : \ell}\ \mathsf{var}F \qquad \frac{}{\Sigma \vdash c : \bot}\ \mathsf{const}F \qquad \frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2 \quad \ell = \ell_1 \sqcup \ell_2}{\Sigma \vdash e_1 + e_2 : \ell}\ +F$$

$\boxed{\Sigma \vdash P : \ell}$

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash e_1 \leq e_2 : \ell_1 \sqcup \ell_2}\ \leq F \qquad \frac{}{\Sigma \vdash \top : \bot}\ \top F \qquad \frac{\Sigma \vdash P : \ell_1 \quad \Sigma \vdash Q : \ell_2}{\Sigma \vdash P \wedge Q : \ell_1 \sqcup \ell_2}\ \wedge F$$

$\boxed{\Sigma \vdash \alpha\ \mathsf{secure}}$

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e\ \mathsf{secure}}\ :=F \qquad \frac{\Sigma \vdash \alpha\ \mathsf{secure} \quad \Sigma \vdash \beta\ \mathsf{secure}}{\Sigma \vdash \alpha\ ;\ \beta\ \mathsf{secure}}\ ;F$$

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha\ \mathsf{secure} \quad \Sigma' \vdash \beta\ \mathsf{secure}}{\Sigma \vdash \mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta\ \mathsf{secure}}\ \mathbf{if}F$$

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha\ \mathsf{secure}}{\Sigma \vdash \mathbf{while}\ P\ \alpha\ \mathsf{secure}}\ \mathbf{while}F$$

$$\frac{}{\Sigma \vdash \mathbf{test}\ P\ \mathsf{secure}}\ \mathbf{test}F$$

Figure 1: Information Flow Type System
Termination insensitive, ensuring confidentiality

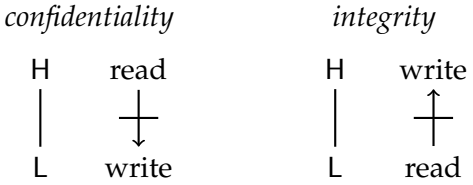
Lecture Notes on Declassification

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

Lecture 12
October 3, 2024

1 Introduction

Before we move on the core topic of today's lecture, namely declassification, we briefly talk about another use of information flow. So far we have focused on *confidentiality*, that is, preventing flow reading high security data and writing them to low security variables. The dual, *integrity*, prevents reading low security data and writing them to high security variables. The prevents an attacker from violating the integrity of high security information.



Reasoning about integrity is dual to what we have developed so far for confidentiality and can be addressed with the same techniques.

In view of the examples showing incompleteness of the type system, one wonders if it is possible to reason about information flow with more precision. In other words, can we perhaps use dynamic logic at the cost of potentially incurring an undecidable problem? The answer is yes, and we will think this through in Section 2.

After that (in Sections 3 and 4) we consider some common scenarios where the kind of information flow control we have discussed so far is too strict, and we need to allow some information to be leaked. But hopefully not too much!

2 Information Flow in Dynamic Logic

Can we reason about information flow more precisely than in the type system by employing dynamic logic? Since noninterference is not a property of a single exe-

cution of a program, it seems at first that $P \to [\alpha]Q$ would be insufficient, because we always just reason about the poststate of a single execution of α.

Let's review the definition, restricting ourselves here to just high (H) and low (L) security levels.

> We define $\Sigma \models \alpha$ secure
> iff for all ω_1 and ω_2, $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ implies $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_{\mathsf{L}} \mathsf{eval}\ \omega_2\ \alpha$

The notation in the conclusion is meant to imply that both evaluations terminate.

Maybe we start with how to represent $\Sigma \models \omega_1 \approx_{\mathsf{L}} \omega_2$. For each variable in the program α there should be two versions. The low-level versions must be equal before the program is executed, but no such constraint is imposed on the high-level versions.

Let's take the example

$$\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b$$

where $\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L})$ and a and b are constants. We create two versions of each variable x, x', y, and y'. The condition that the low-security variables must be equal is represented by

$$y = y'$$

Cool. How do we reason about the evaluation of α in the two different initial states? One solution is the run them sequentially, but rename one of them to use only the primed variables. So:

$$\begin{aligned} y = y' \to [&(\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b)\ ; \\ &(\textbf{if } x' = 0 \textbf{ then } y' := a \textbf{ else } y' := b)]\ Q(x, x', y, y') \end{aligned}$$

But what should the postcondition be? It should once again express that the low-security variables must be equal. So it is the same as the precondition! It will automatically talk about the values of these variables after evaluation for two reasons: (1) the two versions of the program compute over different variables, and (2) $[\alpha]Q$ means that Q must be true in every possible poststate of α (of which there is at most one). If there is none, then any postcondition is provable and the program is considered secure.[1]

So in summary, we define for all variables occurring in the program:

$$Q = \bigwedge_{\Sigma(x)=\mathsf{L}} (x = x')$$

and then prove noninterference by proving

$$Q \to [\alpha\ ;\ \alpha']Q$$

[1] We missed that point in lecture, but fortunately it works out. But one would still have to consider the issue of loop invariants.

where α' is the renaming of α by priming all variables.

Let's apply this technique in our example $\alpha = (\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b)$ by computing the weakest liberal precondition of $y = y'$ with respect to

$$\mathsf{wlp}\ (\alpha\ ;\ \alpha')\ (y = y')$$

We do this by first constructing $\mathsf{wlp}\ \alpha'\ (y = y')$.

$$\begin{array}{ll} & \mathsf{wlp}\ (\textbf{if } x' = 0 \textbf{ then } y' := a \textbf{ else } y' := b)\ (y = y') \\ = & (x' = 0 \to \mathsf{wlp}\ (y' := a)\ (y = y')) \wedge (x' \neq 0 \to \mathsf{wlp}\ (y' = b)\ (y = y')) \\ = & (x' = 0 \to y = a) \wedge (x' \neq 0 \to y = b) \\ = & Q(x', y) \end{array}$$

Now we use $Q(x', y)$ as a postcondition for (the unrenamed) α.

$$\begin{array}{ll} & \mathsf{wlp}\ (\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b)\ Q(x', y) \\ = & (x = 0 \to \mathsf{wlp}\ (y := a)\ Q(x', y)) \wedge (x \neq 0 \to \mathsf{wlp}\ (y' = b)\ Q(x', y)) \\ = & (x = 0 \to Q(x', a) \wedge (x \neq 0 \to Q(x', b)) \\ = & R(x, x') \end{array}$$

Let's work out what this is (with some small simplifications):

$$\begin{array}{lll} R(x, x') & \leftrightarrow & \quad (x = 0 \wedge x' = 0 \to a = a) \\ & & \wedge \quad (x = 0 \wedge x' \neq 0 \to a = b) \\ & & \wedge \quad (x \neq 0 \wedge x' = 0 \to b = a) \\ & & \wedge \quad (x \neq 0 \wedge x' \neq 0 \to b = b) \end{array}$$

The first and last conjunct are true, since $a = a$ and so we obtain

$$\begin{array}{lll} R(x, x') & \leftrightarrow & \quad (x = 0 \wedge x' \neq 0 \to a = b) \\ & & \wedge \quad (x \neq 0 \wedge x' = 0 \to b = a) \end{array}$$

If the constants a and b are equal, this is valid (regardless of x and x') and if a and b are not equal, this is not valid because we can pick x and x' to be different and falsify it. Note that our precondition $y = y'$ is irrelevant here since y and y' are both assigned to by the program.

So we conclude that the program

$$\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b$$

is secure if and only if the constants a and b are equal.

3 Checking PINs

Imagine there is a variable pin holding a personal identification number (in lieu of a password) and a variable $guess$ holding the user's "guess". The job of the

following small program is to match the the user's guess against the pin and set the variable $auth$ to 1 (user authenticated) or 0 (not authenticated).

$$\textbf{if } guess = pin \textbf{ then } auth := 1 \textbf{ else } auth := 0$$

Let's see what the security level of these three variables need to be.

pin**:** Clearly, this shouldn't be leaked so $pin : \mathsf{H}$

$guess$**:** This is the user input, so it is low security $guess : \mathsf{L}$

$auth$**:** This needs to be exposed to the user so they know if they are logged in or not. So $auth : \mathsf{L}$.

At this point we can see that this does **not** satisfy our definition of noninterference and is therefore not secure according to the only reasonable policy. It is easy to cook up an example of two states where the guesses are the same, but lead to different outcomes depending on the pin. In the type system, the problem is reflected in that the comparison is high security and therefore the two branches must be checked with $pc = \mathsf{H}$. But they write to the low security variable $auth$.

In order to account for this kind of situation, Volpano and Smith [2000] introduced a **match** construct into the language, comparing a guess to a secret password. We use a new kind of formula $\mathbf{match}(e_1, e_2)$ with the typing rule

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash \mathbf{match}(e_1, e_2) : \ell_1 \sqcap \ell_2}\ \mathbf{match}F$$

For this rule to make sense we need to generalize the semilattice of security levels we had so far to a *lattice*, where there is a *greatest lower bound* operation $\ell_1 \sqcap \ell_2$ (often pronounced "*meet*"). For the **match** expression we lower the security level of the result, but only as far as necessary so the result is below ℓ_1 and ℓ_2.

Recasting our particular example

$$\textbf{if } \mathbf{match}(guess, pin) \textbf{ then } auth := 1 \textbf{ else } auth := 0$$

it is now well-typed because we have

$$pin : \mathsf{H}, guess : \mathsf{L}, auth : \mathsf{L} \vdash \mathbf{match}(guess, pin) : \mathsf{L}$$

Of course, without any change to our definition of noninterference, it will still not be semantically secure.

This raises several questions:

1. Can we relax our definition of noninterference to account for **match**?
2. What are the consequences of adding it to the language? We intend the effect to be somehow confined, but we might be making a mistake and suddenly all secrets may be leaked.

3. How can we square the fact that the match operation doesn't preserve confidentiality with the fact that it is commonly used and "seems to be fine" (ignoring some practical issues like insecure passwords).

One particularly worrisome aspect of this construct might be the following program:

$$\mathit{guess} := 0 \; ; \; \textbf{while} \; \neg \textbf{match}(\mathit{guess}, \mathit{pin}) \; \mathit{guess} := \mathit{guess} + 1$$

If pins are known to be nonnegative, this program would allow us to determine the password!

The reason this seems to be okay in practice is that each time around the loop, unless the pin has been determined, the attacker only rules out a single possible pin. If the size of the pin is, say, 256 bits, then if the pins are uniformly distributed it would take something like 2^{255} guesses on average to identify the password (which is clearly not feasible).

Volpano and Smith [2000] turn this into a theorem that states that a polynomial attacker can determine a k-bit integer (drawn from a uniform distribution) with probably of at most $(\mathrm{poly}(k) + 1)/2^k$. They also point out that it is essential that the security level of $\textbf{match}(e_1, e_2)$ is $\ell_1 \sqcap \ell_2$ and cannot always be simply of low security (L, or $\bot$ in the general case).

4 Explicit Declassification

When we generalize away from the **match** construct, the situation becomes a lot more complex, even if the new rule is deceptively simple:

$$\frac{}{\Sigma \vdash \textbf{declassify}_\ell(e) : \ell} \; \textbf{declassify} F$$

$\textbf{declassify}_\ell(e)$ is an expression (where ℓ defaults to $\bot$ in general and L in our typical example), but if we had a corresponding formula we could model the **match** with

$$\textbf{if declassify}(\mathit{guess} = \mathit{pin}) \; \textbf{then} \; \mathit{auth} := 1 \; \textbf{else} \; \mathit{auth} := 0$$

However, the generality comes at a price, namely that there no simple reasoning principles covering the many applications of declassification.

There are many dimension to declassification, and we recommend Sabelfeld and Sands [2009] for a broad discussion of the many relevant issues. A key question is *what* is being declassified, and some thought needs to be given *why*. One class of examples are aggregates, like averages. For example, in this course we won't reveal to you the scores of others, but we might reveal class averages. Of course, if there were only two students, and you were one of them, you could infer the other students score if you knew the average. Another example is given by

releasing (possibly anonymized) samples. Neither of these can be done without declassification.

What is being declassified is of critical importance in determining the impact and what an attacker might learn. For example, imagine that instead of **match** we had **compare** that reveals the result of comparing a high security and a low security variable, say, with less-or-equal. Unlike the **match** construct can then learn the pin by using binary search through the space of possibilities.

Can we say anything generically about the **declassify** construct that is useful? In other words, can we generalize noninterference to take declassification into account? In some circumstances we can, but the general definition may still require a lot of work in each case to prove something about how much the attacker can learn.

So let's assume we have a program α with a single occurrence of a construct $\mathbf{declassify}_{\mathsf{L}}(e)$. We would like to express that the attacker can essentially only learn about e, but nothing else. We express this by weakening the noninterference definition to allow the attacker not only know the value of the low-security variables, but also the value of the expression e.

> We attempt to define $\Sigma \models \alpha$ secure where α contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$ iff $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ and $\mathsf{eval}\ \omega_1\ e = \mathsf{eval}\ \omega_2\ e$ imply $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_{\mathsf{L}} \mathsf{eval}\ \omega_2\ \alpha$

Unfortunately, this definition is not quite correct. For example, consider the following program that attempts to declassify an average of $x_1, \ldots, x_n$, all high security variables as $\mathit{average} : \mathsf{L}$.

$$
\begin{array}{l}
x_2 := x_1\ ; \\
x_3 := x_2\ ; \\
\ldots \\
x_n := x_{n-1}\ ; \\
\mathit{average} := \mathbf{declassify}_{\mathsf{L}}((x_1 + x_2 + \cdots + x_n)/n)
\end{array}
$$

Note that this program leaks the value of x_1. That's because in the condition $\mathsf{eval}\ \omega_1\ e = \mathsf{eval}\ \omega_2\ e$ we evaluate e in the original environments, but in this example we assign to the free (high security) values of e before declassifying the aggregate. So we modify our definition to:

> Assume α contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$ and for all $x \in \mathsf{use}\ e$ implies $x \notin \mathsf{maydef}\ \alpha$.
>
> For such α we define $\Sigma \models \alpha$ secure iff
> $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ and $\mathsf{eval}\ \omega_1\ e = \mathsf{eval}\ \omega_2\ e$ imply $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_{\mathsf{L}} \mathsf{eval}\ \omega_2\ \alpha$

With this definition we can prove that the inference rules with the additional rules for declassification are *sound*.

Theorem 1 *If* $\Sigma \vdash \alpha$ secure *and* α *contains exactly one instance of* $\mathbf{declassify}(e)$ *and for all* $x \in$ use e *implies* $x \notin$ maydef α*, then* $\Sigma \models \alpha$ secure *according to the preceding definition.*

Proof: See Sabelfeld and Myers [2003]. □

References

Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *International Symposium on Software Security (ISSS 2003)*, pages 174–191. Springer LNCS 3233, November 2003.

Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

Dennis M. Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In M. N. Wegman and T. W. Reps, editors, *Symposium on Principles of Programming Languages*, pages 268–276, Boston, Massachusetts, January 2000. ACM.

Lecture Notes on Termination-Sensitive Noninterference

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

Lecture 13
October 22, 2024

1 Introduction

Our assumption so far in the semantic definition of information flow has been that an attacker can only compare the values of low-security variables in the poststates.

> We define $\Sigma \models \alpha$ secure (program α satisfies *termination-insensitive noninterference with respect to policy* Σ)
> iff
>
> > for all ℓ, ω_1, ω_2, ν_1, and ν_2,
> > whenever $\Sigma \vdash \omega_1 \approx_\ell \omega_2$,
> > and eval ω_1 $\alpha = \nu_1$,
> > and eval ω_2 $\alpha = \nu_2$
> > then $\Sigma \vdash \nu_1 \approx_\ell \nu_2$.

This ignores that possibility that an attacker might notice that a program does not terminate. For example, with $(x : \mathsf{H})$ and $\mathsf{L} \sqsubset \mathsf{H}$

$$\textbf{if } x > 5 \textbf{ then } (\textbf{while } \top \textbf{ skip}) \textbf{ else skip}$$

an observer can tell if $x > 5$ based on whether the program terminates. But according to our policy it is secure at level L! In this example, that's easy to see because whenever ν_1 and ν_2 both exist, then $\nu_1 \approx_{\mathsf{L}} \nu_2$ because there aren't even any low-security variables. If α does not terminate (that is, has no poststate) in one or both of the two initial states, then the implication is vacuously true.

The first goal of today's lecture is to sharpen our definition of noninterference so that nontermination is considered observable, and the example above would be rejected. The second goal will be to revise the information flow type system so it is sound with respect to the sharpened definition. For general expositions

of today's topic, see Hedin and Sabelfeld [2012], Sabelfeld and Myers [2003], and earlier seminal work by Volpano and Smith [1997].

This lecture is also a preparation for the next lecture where we talk about *timing attacks* where information can leak by observing how long a program takes to execute. These kind of attacks are quite common and real concerns in actual systems [Brumley and Boneh, 2005], so it is important to study them from the programming perspective.

2 Termination-Sensitive Noninterference

Intuitively, we'd like to say that if eval $\omega_1 \alpha$ and eval $\omega_2\ \alpha$ both return a poststate, then they still must be equivalent to an observer at level ℓ. Moreoever, if one does not terminate then the other one doesn't either. In this latter case, there are no poststates to compare. The traditional definition [Volpano and Smith, 1997] captures this by stating that if eval $\omega_1\ \alpha = \nu_1$ then there must also be an ν_2 such that eval $\omega_2\ \alpha = \nu_2$, and the two poststates must be observably equivalent. This slightly asymmetric definition is correct due to the symmetry of the $\approx_\ell$ relation. If α terminates in prestate ω_1 but does not in ω_2, then we can just swap the two prestates to show that such a program is not secure. We write $\Sigma \models \alpha$ secure$^\infty$ for termination-sensitive noninterference.

> We define $\Sigma \models \alpha$ secure$^\infty$ (program α satisfies *termination-sensitive noninterference with respect to policy* Σ)
> iff
>
> > for all ℓ, ω_1, ω_2, and ν_1,
> > whenever $\Sigma \vdash \omega_1 \approx_\ell \omega_2$,
> > and eval $\omega_1\ \alpha = \nu_1$,
> > then eval $\omega_2\ \alpha = \nu_2$ for some ν_2
> > such that $\Sigma \vdash \nu_1 \approx_\ell \nu_2$.

Let's make sure our example program (let's call it α_0) is not secure under this definition. For this purpose we have to find a counterexample, that is, two states ω_1 and ω_2 that are low-security equivalent such that α_0 terminates in state ω_1 but not in ω_2. Fortunately, that is easy: for

$$\alpha_0 = (\textbf{if}\ x > 5\ \textbf{then}\ (\textbf{while}\ \top\ \textbf{skip})\ \textbf{else}\ \textbf{skip})$$

we can use

$\omega_1 = (x \mapsto 0)$	eval $\omega_1\ \alpha_0 = (x \mapsto 0)$
$\omega_2 = (x \mapsto 7)$	there exists no ν_2 with eval $\omega_2\ \alpha_0 = \nu_2$

3 Sharpening the Information Flow Type System

The information flow type system so far will admit the program α_0 as secure. The idea is now to revisit the rules to determine how we might need to update them to enforce termination-sensitive noninterference. To parallel the definition of noninterference, we write $\Sigma \vdash \alpha$ secure$^\infty$.

The first thing we note is that the security level of expressions and formulas do not change—they remain the least upper bound of the levels of all variables occurring in them. This is due to our decision that expressions and Boolean conditions always terminate (and are always safe).

So we can focus our attention on the program constructs. In the fragment we treat, every command will be safe, that is, we handle SAFETINY.

Assignment. We first show the prior (termination-insensitive) version of the rule.

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \text{ secure}} :=F$$

Because expressions are not involved with nontermination, this rule remains unchanged! This is not a proof, of course, it is not difficult to update the one from the termination-insensitive case. If $x := e$ terminates in ω_1 then it also will in ω_2, so the two formulations are equivalent in this case.

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \text{ secure}^\infty} :=F^\infty$$

Sequential Composition. Our previous rule can be summarized as saying that termination-insensitive information flow is *compositional*.

$$\frac{\Sigma \vdash \alpha \text{ secure} \quad \Sigma \vdash \beta \text{ secure}}{\Sigma \vdash \alpha \,;\, \beta \text{ secure}} ;F$$

Is that still the case, that is, is termination-sensitive information flow also compositional? It is not entirely obvious when α does not terminate, but we conjecture:

$$\frac{\Sigma \vdash \alpha \text{ secure}^\infty \quad \Sigma \vdash \beta \text{ secure}^\infty}{\Sigma \vdash \alpha \,;\, \beta \text{ secure}^\infty} ;F^\infty$$

Let's try to prove or refute the soundness of this. So we set up:

$\Sigma \models \alpha$ secure$^\infty$	(1, first premise)
$\Sigma \models \beta$ secure$^\infty$	(2, second premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$	(3, assumption)

eval ω_1 $(\alpha\ ;\ \beta) = \nu_1$	(4, ssumption)
...	
eval ω_2 $(\alpha\ ;\ \beta) = \nu_2$ for some ν_2	(to show)
with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(to show)

We start by expanding the definition of evaluation for sequential composition. We show the new lines in blue after each step.

$\Sigma \models \alpha$ secure$^\infty$	(1, first premise)
$\Sigma \models \beta$ secure$^\infty$	(2, second premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$	(3, assumption)
eval ω_1 $(\alpha\ ;\ \beta) = \nu_1$	(4, assumption)
eval ω_1 $\alpha = \mu_1$ for some μ_1	(5 and)
and eval μ_1 $\beta = \nu_1$	(6, from (4) by defn. of eval)
...	
eval ω_2 $(\alpha\ ;\ \beta) = \nu_2$ for some ν_2	(to show)
with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(to show)

At this point we can use the assumption (coming from the first premise) that α is secure.

$\Sigma \models \alpha$ secure$^\infty$	(1, first premise)
$\Sigma \models \beta$ secure$^\infty$	(2, second premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$	(3, assumption)
eval ω_1 $(\alpha\ ;\ \beta) = \nu_1$	(4, assumption)
eval ω_1 $\alpha = \mu_1$ for some μ_1	(5, and)
and eval μ_1 $\beta = \nu_1$	(6, from (4) by defn. of eval)
eval ω_2 $\alpha = \mu_2$ for some μ_2	(7 and)
with $\Sigma \vdash \mu_1 \approx_\ell \mu_2$	(8, from (1), (3), and (5))
...	
eval ω_2 $(\alpha\ ;\ \beta) = \nu_2$ for some ν_2	(to show)
with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(to show)

At this point we have an evaluation of β from a prestate μ_1 and a state μ_2 that is ℓ-equivalent to it. So we can use the security for β (coming from the second premise).

$\Sigma \models \alpha$ secure$^\infty$	(1, first premise)
$\Sigma \models \beta$ secure$^\infty$	(2, second premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$	(3, assumption)
eval ω_1 $(\alpha\ ;\ \beta) = \nu_1$	(4, assumption)
eval ω_1 $\alpha = \mu_1$ for some μ_1	(5, and)
and eval μ_1 $\beta = \nu_1$	(6, by defn. of eval)
eval ω_2 $\alpha = \mu_2$ for some μ_2 (7 and)	
with $\Sigma \vdash \mu_1 \approx_\ell \mu_2$	(8, from (1), (3), and (5))
eval μ_2 $\beta = \nu_2$ for some ν_2	(9 and)

with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(10, from (2), (6), and (8))
...	
eval ω_2 $(\alpha\ ;\ \beta) = \nu_2$ for some ν_2	(to show)
with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(to show)

At this point we have evaluations of α and β in corresponding states so we can close the gap simply by composing them. The ν_2 we had to find is (not surprisingly) the poststate of β when evaluated in prestate μ_2.

$\Sigma \models \alpha$ secure$^\infty$	(1, first premise)
$\Sigma \models \beta$ secure$^\infty$	(2, second premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$	(3, assumption)
eval ω_1 $(\alpha\ ;\ \beta) = \nu_1$	(4, assumption)
eval ω_1 $\alpha = \mu_1$ for some μ_1	(5, and)
and eval μ_1 $\beta = \nu_1$	(6, by defn. of eval)
eval ω_2 $\alpha = \mu_2$ for some μ_2 (7 and)	
with $\Sigma \vdash \mu_1 \approx_\ell \mu_2$	(8, from (1), (3), and (5))
eval μ_2 $\beta = \nu_2$ for some ν_2	(9 and)
with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(10, from (2), (6), and (8))
eval ω_2 $(\alpha\ ;\ \beta) = \nu_2$	(by defn. of eval)
with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(copied from (10))

Good. Our intuition that even termination-sensitive noninterference is compositional has been confirmed.

Skip. This does nothing, so the rule trivially remains the same.

$$\frac{}{\Sigma \vdash \textbf{skip}\ \text{secure}^\infty}\ \textbf{skip}F^\infty$$

Conditionals. The if-then-else construct somehow doesn't seem to be involved in nontermination, only loops are. We therefore expect that the rule doesn't change.

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha\ \text{secure}^\infty \quad \Sigma' \vdash \beta\ \text{secure}^\infty}{\Sigma \vdash \textbf{if}\ P\ \textbf{then}\ \alpha\ \textbf{else}\ \beta\ \text{secure}^\infty}\ \textbf{if}F^\infty$$

The proof is just like before, in the termination-insensitive case. The fact that we now have to account for the nontermination of α or β changes the details of reasoning, of course, but the possibility of nontermination just comes from the nontermination of each branch. So we omit this case.

Loops. Here, we'd expect the crux of the changes because loops introduce nontermination into the language. First, the earlier rule:

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \text{ secure}}{\Sigma \vdash \textbf{while } P\ \alpha \text{ secure}}\ \textbf{while}F$$

We see that there are essentially two reasons why a while loop may leak information using nontermination (with $(x : \mathsf{H})$)

1. In the example **if** $x > 5$ **then** $(\textbf{while } \top \textbf{ skip})$ **else skip** it is because the loop is in a context where we have $pc : \mathsf{H}$.

2. In the example **while** $x = 8$ **skip** it is because the loop guard is of high security.

Since nontermination is observed "at the outermost level" and not by a variable in the local context, it seems we have to require that both the security level of the pc and the guard are $\bot$, that is, the least element of the lattice. This means that $\ell' = \Sigma(pc) \sqcup \ell$ should also be $\bot$, and we don't need to update the security level of pc.

$$\frac{\Sigma \vdash P : \bot \quad \Sigma(pc) = \bot \quad \Sigma \vdash \alpha \text{ secure}^\infty}{\Sigma \vdash \textbf{while } P\ \alpha \text{ secure}^\infty}\ \textbf{while}F^\infty$$

Is this sufficient to guarantee termination-sensitive noninterference? We set up:

$\Sigma \models \alpha$ secure$^\infty$	(1, premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$	(2, assumption)
eval ω_1 $(\textbf{while } P\ \alpha) = \nu_1$	(3, assumption)
...	
eval ω_2 $(\textbf{while } P\ \alpha) = \nu_2$ for some ν_2	(to show)
with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$	(to show)

At this point, there is a small hiccup: the evaluation of a loop may refer back again to the evaluation of the same loop. We therefore use an auxiliary induction on the number of times we iterate the loop when computing eval ω_1 $(\textbf{while } P\ \alpha)$ to ν_1. We know it does compute a poststate, so the number of steps is finite and the induction makes sense. Let's call that number n (which is also the notation we chose in the semantic definition of $\omega[\![\textbf{while } P\ \alpha]\!]^n\nu$, see page L4.6 of Lecture 4).

Case: $n = 0$. Then $\text{eval}_{\mathbb{B}}\ \omega_1\ P = \bot$ and $\nu_1 = \omega_1$. Because P only contains variables of security level $\bot \sqsubseteq \ell$ and $\Sigma \vdash \omega_1 \approx_\ell \omega_2$, we also have $\text{eval}_{\mathbb{B}}\ \omega_2\ P = \bot$. (See Lemma 2 on page L11.4 of Lecture 11.) Therefore $\nu_2 = \omega_2$ will satisfy the requirements of the theorem.

Case: $n > 0$. Then $\mathsf{eval}_{\mathbb{B}}\ \omega_1\ P = \top$ and $\mathsf{eval}\ \omega_1\ (\alpha\ ;\ \mathbf{while}\ P\ \alpha) = \nu_1$ As in the previous case $\mathsf{eval}_{\mathbb{B}}\ \omega_2\ P = \top$ and it remains to show that $\mathsf{eval}\ \omega_2\ (\alpha\ ;\ \mathbf{while}\ P\ \alpha) = \nu_2$ for some ν_2 with $\Sigma \vdash \nu_1 \approx_\ell \nu_2$.

As in the case for composition (and exploiting the security of α that comes from the premise) this reduces to showing that $\mathsf{eval}\ \mu_1\ (\mathbf{while}\ P\ \alpha) = \nu_1$ implies $\mathsf{eval}\ \mu_2\ (\mathbf{while}\ P\ \alpha) = \nu_2$ for some ν_2 indistinguishable from ν_1 at level ℓ, with the knowledge that $\Sigma \vdash \mu_1 \approx_\ell \mu_2$. This is true because of the induction hypothesis on $n - 1 < n$, which is the remaining number of times the loop is traversed.

Tests. Recall that $\mathbf{test}\ P$ aborts if P is false. This means it represents another type of program (besides a loop) that does not have a poststate. Lumping together aborting a program and nontermination may be questionable in practice, but if we consider "nontermination" simply as representing the absence of a poststate, we would have defined

$$\mathbf{test}\ P \triangleq \mathbf{while}\ \neg P\ \mathbf{skip}$$

Then the rule derived from these considerations would be:

$$\frac{\Sigma \vdash P : \bot \quad \Sigma(pc) = \bot}{\Sigma \vdash \mathbf{test}\ P\ \mathsf{secure}^\infty}\ \mathbf{test}F^\infty$$

All together we obtain soundness.

Theorem 1 (Soundness of termination-sensitive information flow types) *If* $\Sigma \vdash \alpha\ \mathsf{secure}^\infty$ *then* $\Sigma \models \alpha\ \mathsf{secure}^\infty$

Proof: All the rules are *sound*, that is, the preserve semantic validity of the premises. By a trivial induction over the derivation of $\Sigma \vdash \alpha\ \mathsf{secure}^\infty$, every program judged secure satisfies termination-sensitive noninterference. □

4 Further Discussion

The soundness of all rules for $\Sigma \vdash \alpha\ \mathsf{secure}^\infty$ guarantees that it implies $\Sigma \models \alpha\ \mathsf{secure}^\infty$, that is, termination-sensitive information flow. The other direction (completeness) is not true, and there are simple counterexamples.

We also have the intuition that it should be *stricter* than the termination-insensitive one. That should be true syntactically, in the type system, and semantically, with respect to noninterference. Let's check that:

Theorem 2 *If* $\Sigma \models \alpha\ \mathsf{secure}^\infty$ *then* $\Sigma \models \alpha\ \mathsf{secure}$.

Proof: We set up:

$\Sigma \models \alpha$ secure$^\infty$ (1, assumption)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$ (2, assumption)
eval $\omega_1\ \alpha = \nu_1$ (3, assumption)
eval $\omega_2\ \alpha = \nu_2$ (4, assumption)
...
$\Sigma \vdash \nu_1 \approx_\ell \nu_2$ (to show)

We can now apply the definition of secure$^\infty$.

$\Sigma \models \alpha$ secure$^\infty$ (1, assumption)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$ (2, assumption)
eval $\omega_1\ \alpha = \nu_1$ (3, assumption)
eval $\omega_2\ \alpha = \nu_2$ (4, assumption)
eval $\omega_2\ \alpha = \nu_2'$ for some ν_2' with (5 and)
$\Sigma \vdash \nu_1 \approx_\ell \nu_2'$ (6, from (1), (2), and (3))
...
$\Sigma \vdash \nu_1 \approx_\ell \nu_2$ (to show)

Since our language is deterministic, we have $\nu_2 = \nu_2'$ and the proof is complete.

$\Sigma \models \alpha$ secure$^\infty$ (1, assumption)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$ (2, assumption)
eval $\omega_1\ \alpha = \nu_1$ (3, assumption)
eval $\omega_2\ \alpha = \nu_2$ (4, assumption)
eval $\omega_2\ \alpha = \nu_2'$ for some ν_2' with (5 and)
$\Sigma \vdash \nu_1 \approx_\ell \nu_2'$ (6, from (1), (2), and (3))
$\Sigma \vdash \nu_1 \approx_\ell \nu_2$ (7, from (4), (5), and (6) by determinism)

□

Any syntactic (decidable) type system for a Turing-complete language may be considered an approximation of a true semantic property. This is the essence of Rice's theorem. Any type system that restricts information flow then represents a tradeoff between the kind of programs that are allowed, and the simplicity and uniformity of the system. This is difficult to discuss in the abstract, so in Lab 2 you will have the opportunity to explore it a bit yourself. The information flow type systems from the last three lectures all seems to represent reasonable compromises: the rules are sound and quite simple, and many programs can be written (even if some require declassification).

Perhaps the most significant issue is that there exist further *side channels* by which information can be obtain, most notably perhaps by observing program runtime. We will complete this investigation in the next lecture.q

References

David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, August 2005.

Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, pages 319–347. IOS Press, 2012.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

Dennis M. Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *10th Computer Security Foundations Workshop (CSFW 1997)*, pages 156–168. IEEE Computer Society, June 1997.

Lecture Notes on Timing Attacks

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

Lecture 14
October 24, 2024

1 Introduction

Deducing secrets (like passwords or private keys) by observing how long a program runs is a commonly used attack on computer security. This is true despite the uncertainties about how long a program will actually run (which is only stochastically connected to the source code). For example, Brumley and Boneh [2005] demonstrate that remote timing attacks are quite feasible and can be launched against security-critical code such as an implementation of SSL. They had to run a program many times, but in the end they were successful in exploiting data-dependent optimizations in the cryptographic primitives underlying SSL. Due to the serious nature of such security threats, defenses have been devised. The primary ones are (1) introduce randomness into the computation so that the time variations don't give away the secret information, and (2) sharpen the information flow type system to ensure "constant-time" computation. We explore the second and briefly touch on the first. Timing attacks against cryptographic primitives are still being discovered even today.

There are other so-called *side channels* for information. Here are some:

- Power consumption
- Memory access patterns
- Sequence of instructions executed
- Electromagnetic radiation emitted from the hardware

Some of them require direct access to some hardware, other can be launched remotely. What they have in common in that they use physical properties of what takes place in a computer when code is executed that is outside of the usual abstract model of computation we work with when reasoning about our code. This

is a significant difference between functional correctness (e.g., "this code will sort a list") and security. In the latter case we can only prove security against a policy and a particular threat model—we can't make any blanket statements about attacks that might fall outside the model.

As an example of the complexity of side-channel attacks that occur in real life, we briefly consider Spectre [Kocher et al., 2019]. It exploits a some features of a modern processor, namely speculative execution, together with timing attack on the memory cache. They use the following sample code to illustrate their technique:

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

This code tries to guard access to `array1` to prevent out-of-bounds access. In the first phase of the attack, it is given many valid values for `x`, training the processor's branch prediction algorithm to believe that this branch will usually be taken. In the second phase, it is given and input that is out of bounds, that is, greater than `array1_size`. Branch prediction will assume the test turns out to be true and starts executing `y = array2[array1[x] * 4096]`. When the processor discovers that the condition is actually false, it will abandon all the actions taken, such as retrieving `array1[x]`, multiplying it to obtain some address `addr`, retrieving a value from `array2[addr]`, and any other registers, condition, codes, etc. However, one thing it does **not** undo is storing some fetched data in the memory cache. Note that the "secret" data at `array1[x]` are used as an address, so the data at this address might still be in the cache. Now in the third phase the attacker launches a timing attack against the cache to determine which memory region is now in the cache, which can reveal the underlying data at `array1[x]`.

2 Some Simple Timing Attacks

Real timing attacks [Brumley and Boneh, 2005] are complex, among other things due to timing variations in processor execution. We abstract away from these details and it turns out that the main defenses, like "constant-time programming", are ultimately the same. Here are some timing attacks, intentionally somewhat hypothetical (slightly outside our language) to give you the opportunity to devise your own in Lab 2.

With $\Sigma_0 = (pin : \mathsf{H})$, we write

$$\textbf{while}\ (pin > 0)\ pin := pin - 1$$

Assuming the original value of pin (say c) is positive, this will take something like $4{*}c{+}2$ steps to terminate. Here we count every arithmetic operation or comparison as 1 step, the assignment as 1 step, and processing of **while** as another step. While

details may vary with the cost model, it should be clear that it is easy to determine the secret value of pin from the running time.

This code, however, does not represent a very useful attack, because it is too slow, for example, if the pin has 256 bits. But we can modify it into a standard strawman example (omitting any details regarding possibly necessary declassification):

$auth := 1$;
$i := 0$;
while $(i < len)$
  if $pin[i] = guess[i]$
  then $i := i + 1$
  else $(auth := 0 \; ; \; i = len)$

Here $str[i]$ could either access the ith character of a string str or the ith bit of the number. len would represent either the length of the string or the width of the pin in bits.

Let's assume there are nonnegative integers, and len is the number of bits. If the lowest bit of the guess is incorrect, the program will terminate with $auth$ almost immediately. If it is correct, it will go around the loop at least one more time, and therefore take longer. That will allow us to infer the lowest bit of the secret pin. Our next guess will have that bit correct and now find the next bit, etc. With at most $2 * len$ experiments we should be able to determine all bits.

3 Time-Sensitive Noninterference

As in the last lecture, our first job will be to define the correct semantic notion of noninterference and measure it against the examples. We would like that if both programs terminate from prestates that are indistinguishable for a low-security observer, they take the same number of steps and terminate in low-security equivalent states. In order to express the number of steps, we change our evaluation function to return notcc only a poststate, but also the number of steps taken. The latter will be defined shortly.

> We define $\Sigma \models \alpha$ secure^t (program α satisfies *time-sensitive noninterference with respect to policy* Σ)
> iff
>
> > for all ℓ, ω_1, ω_2, ν_1, ν_2, n_1, and n_2,
> > whenever $\Sigma \vdash \omega_1 \approx_\ell \omega_2$,
> > and eval ω_1 $\alpha = (n_1, \nu_1)$,
> > and eval ω_2 $\alpha = (n_2, \nu_2)$,

then $\Sigma \vdash \nu_1 \approx_\ell \nu_2$ and $n_1 = n_2$.

Note that we have disregarded nontermination in this definition. It is easy to combine it with the requirement to be termination-sensitive, but we prefer to separate these concerns for now.

Now our first example (with $\Sigma_0 = (pin : \mathsf{H})$) clearly does not satisfy this definition, which is what we would hope.

$$\alpha_0 = (\textbf{while}\ (pin > 0)\ pin := pin - 1)$$

A simple counterexample is

$$\begin{array}{ll} \omega_1 = (pin \mapsto 0) & \mathsf{eval}\ \omega_1\ \alpha_0 = (2, pin \mapsto 0) \\ \omega_2 = (pin \mapsto 1) & \mathsf{eval}\ \omega_2\ \alpha_0 = (6, pin \mapsto 0) \end{array}$$

We have $\Sigma_0 \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ (since $pin : \mathsf{H}$), and the poststates are indistinguishable (not only for a low-security observer, but in general), but the steps $n_1 = 2 \neq 6 = n_2$.

The second example is subject to a similar analysis, but we'd need to be careful about declassification which we would like to avoid.

4 Evaluation with Time

Next we should define evaluation with step-counting. This is quite straightforward, just a bit tedious. You can find the summary of this exercise in Figure 1. We have to recognize that this form of timing isn't realistic and hope that to some extent the design of the information flow type system can make up for this lack of realism.

One point about the Boolean operations: we do not want their running times to be data-dependent. This means that evaluating $\top \wedge P$ and $\bot \wedge P$ should take the same amount of time. In other words, the Boolean operations should not be "short-circuiting". Recall that in a previous lecture we determined that it didn't matter whether they were short-circuiting or not since all expressions and formulas are safe and terminating. In this context it does matter, and steps may need to be taken to ensure data independence.

Timing considerations raise another point about abstractions. So far we have said that variables can hold arbitrary integers. However, this means that the running times of many operations like addition cannot be data-independent. So we assume instead that integers are implemented with a fixed range, like 64 bits or 256 bits and that arithmetic is modular. This creates a number of complications when reasoning about the correctness of code, but it actually makes reasoning about certain security properties (especially those that are timing-sensitive) both more realistic and simpler.

5 A Type System for Constant-Time Computation

Next comes the task to develop a type system that enforces time-sensitive noninterference, which is usually called "*constant-time*". However, it isn't actually constant time, it is just that the running time can only depend on low-security inputs. In order to formalize that, we once again proceed construct by construct. We write $\Sigma \vdash \alpha \ \mathsf{secure}^t$ for time-sensitive security with respect to signature Σ.

Assignment. For now, there doesn't seem to be any reason to change our rule for assignment. The point is that the time for the evaluation of an expression e is some n, regardless of the state ω. So it just once again comes down to the basic information flow properties.

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \ \mathsf{secure}^t} :=\!F^t, \text{preliminary version}$$

See the end of this section for further thoughts on assignment.

Sequential Composition and skip. If α and β are indistinguishable even via a timing channel, then their composition should not be, either. The proof is straightforward and follows familiar patterns, so we omit it here. **skip** is trivial, as usual.

$$\frac{\Sigma \vdash \alpha \ \mathsf{secure}^t \quad \Sigma \vdash \beta \ \mathsf{secure}^t}{\Sigma \vdash \alpha \ ; \ \beta \ \mathsf{secure}^t} \ ;F^t \qquad \frac{}{\Sigma \vdash \mathbf{skip} \ \mathsf{secure}^t} \ \mathbf{skip}F^t$$

Conditionals. The first instinct might be that in a conditional, both branches need to take the same amount of time. Under such a discipline, we could rewrite our motivating example as follows:

$auth := 1$;
$i := 0$;
while $(i < len)$
  if $pin[i] = guess[i]$
  then $auth := auth$
  else $auth := 0$;
  $i := i + 1$

The good part here is that we go around the loop the same number of times, regardless whether the guess is correct or not. The bad part is that even though the two branches, abstractly, take the same amount of time that is unlikely to be case in practice. Any reasonable compiler would optimize $auth := auth$ into a no-op and the loop would leak some timing information.

The problem actually goes deeper: in order to have a *type system* for time-sensitive information flow that allows such a program, the type system would have to capture exactly how long an expression or command takes to evaluate. Not only would this be inaccurate with respect to the actually running code, it would also require our type-checker to perform arithmetic reasoning. This would just be too brittle a design.

Instead, we take a more drastic step: for any conditional, we require that the formula P depends only on low-security variables. This rules out our sample program above, because the comparison $pin[i] = guess[i]$ depends on pin which is of high security.

Intuitively, we are repeating the solution for termination-sensitive information flow from the last lecture, but with conditionals instead of loops. Here is the first formulation: we just require ℓ' to be $\bot$, the least element of the security lattice.

$$\frac{\Sigma \vdash P : \ell \quad \bot = \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \ \mathsf{secure}^t \quad \Sigma' \vdash \beta \ \mathsf{secure}^t}{\Sigma \vdash \mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta \ \mathsf{secure}^t}\ \mathbf{if} F^t$$

We notice that this requires $\ell = \Sigma(pc) = \bot$. And if $\ell' = \bot = \ell$, there is no need to update the pc, so we can use $\Sigma' = \Sigma$.

$$\frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha \ \mathsf{secure}^t \quad \Sigma \vdash \beta \ \mathsf{secure}^t}{\Sigma \vdash \mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta \ \mathsf{secure}^t}\ \mathbf{if} F^t$$

At this point we realize we don't need pc at all, since we only introduced it to track the implicit flow from the condition into the branches of an if-then-else.

With this simplification, we have drastically reduced the range of programs that are considered secure. So how do we rewrite our example? We have to "inline" the conditional as an ordinary operation. Note that the number of bit (or maximal length of the string) would have to be a low-security variable.

$auth := 1$;
$i := 0$;
while $(i < len)$
  $auth := auth \wedge (pin[i] = guess[i])$;
  $i := i + 1$

This is just outside the range of what our language permits in that equality produces a Boolean, and conjunction takes two Booleans, but here they are integers. This could be solved any number of ways. Perhaps the simplest is that the Boolean operators are overloaded to also work on integers, for example, with $\bot$ represented by 0 and $\top$ by 1, or by anything nonzero.

Also, we go back and think of pin and $guess$ as strings rather than nonnegative integers, this code still makes sense and is not subject to timing attacks.

Let's prove the soundness of this rule. We set up:

$\Sigma \models P : \bot$ (1, first premise)
$\Sigma \models \alpha$ secure^t (2, second premise)
$\Sigma \models \beta$ secure^t (3, third premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$ (4, assumption)
eval ω_1 (**if** P **then** α **else** β) $= (n_1, \nu_1)$ (5, assumption)
eval ω_2 (**if** P **then** α **else** β) $= (n_2, \nu_2)$ (6, assumption)
...
$\Sigma \vdash \nu_1 \approx_\ell \nu_2$ and (to show)
$n_1 = n_2$ (to show)

Because $\Sigma \vdash \omega_1 \approx_\ell \omega_2$ and $\bot \sqsubseteq \ell$, we have that eval ω_1 $P =$ eval ω_2 $P = (k, b)$ for some nonnegative time k and Boolean b. We consider the case where $b = \top$; the other is symmetric. Then

$$\text{eval } \omega_1 \text{ (}\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) = (k + m_1 + 1, \nu_1)$$

where eval ω_1 $\alpha = (m_1, \nu_1)$ and $n_1 = k + m_1 + 1$.

Also

$$\text{eval } \omega_2 \text{ (}\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) = (k + m_2 + 1, \nu_2)$$

where eval ω_2 $\alpha = (m_2, \nu_2)$ and $n_2 = k + m_2 + 1$.

Since α is secure (that is, $\Sigma \models \alpha$ secure^t, which comes from the second premise) we get $\Sigma \vdash \nu_1 \approx_\ell \nu_2$ (which is one of the two properties we needed to prove) and $m_1 = m_2$.

Therefore, also $n_1 = k + m_1 + 1 = k + m_2 + 1 = n_2$, which is the second property we needed to prove.

Loops. For the same reason as conditionals, we require the loop guard to be of low security.

$$\frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha \text{ secure}^t}{\Sigma \vdash \mathbf{while}\ P\ \alpha \text{ secure}^t}\ \mathbf{while}F^t$$

Tests. If the test fails and the program aborts we have a different kind of channel. Again, the simplest condition that avoid information flow here (even if outside of the definition) is to require the test to be of low security.

$$\frac{\Sigma \vdash P : \bot}{\Sigma \vdash \mathbf{test}\ P \text{ secure}^t}\ \mathbf{test}F$$

Assignment Revisited. Since we eliminated the ghost variable pc, we simplify the rule for assignment.

$$\frac{\Sigma \vdash e : \ell \quad \ell \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \text{ secure}^t}\ :=F^t$$

A summary of the rules can be found in Figure 2. It is remarkably simple because it is not afraid to rule out many programs. The soundness of all the rules taken together gives us the soundness of the whole type system.

Theorem 1 (Soundness of timing-sensitive information flow typing)
If $\Sigma \vdash \alpha$ secure^t *then* $\Sigma \models \alpha$ secure^t

Proof: All the rules are sound in the sense that they preserve semantic validity of the premises. By a trivial induction over the derivation of $\Sigma \vdash \alpha$ secure^t we obtain $\Sigma \models \alpha$ secure^t. □

Comparing the type system against the one from the last lecture for termination-sensitive noninterference, we see that it actually enforces that as well. One can update the definition of noninterference and statement of the above theorem to account for that.

6 Randomization

Another defense against timing attacks is to introduce randomization into the program. Even though an observer might still be able to observe timing, this information will then be insufficient to determine the secret. In our example (using the interpretation of pins and guesses as large integers) we could write something like:

$r := \mathsf{random}()$;
$pin := pin + r$;
$guess := guess + r$;
$auth := 1$;
$i := 0$;
while $(i < len)$
  if $pin[i] = guess[i]$
  then $i := i + 1$
  else $(auth := 0 \,;\, i = len)$

Every time this program is run, it is like to use a different random number, essentially making it impossible to draw useful conclusions from the running times. Some care must be taken for the randomization and its result to be sufficiently uniform.

References

David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, August 2005.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (SP 2019)*, pages 1–19, San Francisco, California, May 2019. IEEE.

$$
\begin{array}{lcl}
\mathsf{eval}_{\mathbb{Z}}\ \omega\ c & = & (0, c) \\
\mathsf{eval}_{\mathbb{Z}}\ \omega\ x & = & (1, \omega(x)) \\
\mathsf{eval}_{\mathbb{Z}}\ \omega\ (e_1 + e_2) & = & (n_1 + n_2 + 1, c_1 + c_2) \\
 & & \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_1 = (n_1, c_1) \\
 & & \text{and } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_2 = (n_2, c_2) \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (e_1 \leq e_2) & = & (n_1 + n_2 + 1, c_1 \leq c_2) \\
 & & \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_1 = (n_1, c_1) \\
 & & \text{and } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_2 = (n_2, c_2) \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (\top) & = & (0, \top) \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (\bot) & = & (0, \bot) \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (P \wedge Q) & = & (n_1 + n_2 + 1, b_1 \wedge b_2) \\
 & & \text{where } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = (n_1, b_1) \\
 & & \text{and } \wedge\mathsf{eval}_{\mathbb{B}}\ \omega\ Q = (n_2, b_2) \\
\mathsf{eval}\ \omega\ (x := e) & = & (n + 1, \omega[x \mapsto c]) \\
 & & \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e = (n, c) \\
\mathsf{eval}\ \omega\ (\alpha\ ;\ \beta) & = & (n_1 + n_2 + 1, \nu) \\
 & & \text{where } \mathsf{eval}\ \omega\ \alpha = (n_1, \mu) \\
 & & \text{and } \mathsf{eval}\ \mu\ \beta = (n + 1, \nu) \\
\mathsf{eval}\ \omega\ (\mathbf{skip}) & = & (1, \omega) \\
\mathsf{eval}\ \omega\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) & = & (k + n + 1, \nu) \\
 & & \text{where } \mathsf{eval}\ \omega\ P = (k, \top) \text{ and } \mathsf{eval}\ \omega\ \alpha = (n, \nu) \\
 & & \text{or } \mathsf{eval}\ \omega\ P = (k, \bot) \text{ and } \mathsf{eval}\ \omega\ \beta = (n, \nu) \\
\mathsf{eval}\ \omega\ (\mathbf{while}\ P\ \alpha) & = & (k + n + 1, \nu) \\
 & & \text{where } \mathsf{eval}\ \omega\ P = (k, \bot) \text{ and } n = 0 \text{ and } \nu = \omega \\
 & & \text{or } \mathsf{eval}\ \omega\ P = (k, \top) \text{ and } \mathsf{eval}\ \omega\ (\alpha\ ;\ \mathbf{while}\ P\ \alpha) = (n, \nu)
\end{array}
$$

Figure 1: Timed Evaluation

$$\frac{\Sigma \vdash e : \ell \quad \ell \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \ \mathsf{secure}^t} :=F^t$$

$$\frac{\Sigma \vdash \alpha \ \mathsf{secure}^t \quad \Sigma \vdash \beta \ \mathsf{secure}^t}{\Sigma \vdash \alpha \ ; \ \beta \ \mathsf{secure}^t} ;F^t \qquad \frac{}{\Sigma \vdash \mathbf{skip} \ \mathsf{secure}^t} \ \mathbf{skip}F^t$$

$$\frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha \ \mathsf{secure}^t \quad \Sigma \vdash \beta \ \mathsf{secure}^t}{\Sigma \vdash \mathbf{if} \ P \ \mathbf{then} \ \alpha \ \mathbf{else} \ \beta \ \mathsf{secure}^t} \ \mathbf{if}F^t \qquad \frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha \ \mathsf{secure}^t}{\Sigma \vdash \mathbf{while} \ P \ \alpha \ \mathsf{secure}^t} \ \mathbf{while}F^t$$

$$\frac{\Sigma \vdash P : \bot}{\Sigma \vdash \mathbf{test} \ P \ \mathsf{secure}^t} \ \mathbf{test}F$$

Figure 2: Timing Sensitive Information Flow Typing

Lecture Notes on Authorization Logic

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

Lecture 15
October 29, 2024

1 Introduction

We started the course with an analysis of *safety properties* of computations, and how to account for them statically (via verification conditions in dynamic logic) or dynamically (via sandboxing). Then we moved on to the more complex *information flow properties*. An attacker may discover secrets via particular programs or inputs in a variety of ways, with side-channel attacks (for example, timing attacks) being the most sophisticated. Countermeasures are mostly via information-flow type systems.

With this lecture we are starting a new section of the course, considering *authentication* (you are who you say you are) and *authorization* (you are allowed to perform the actions you are trying to perform). Authentication and authorization are pervasive in today's computing environment, from shared file systems like AFS and cloud services like Github, to shopping and banking services. Today's and the next lecture will focus on *authorization*, followed in later lectures by authentication. In many cases, authorization is just embedded in code. Once authorization becomes complex, this can easily be compromised by bugs since flow of the authorization checks through a program can be difficult to audit. It can also be difficult to even understand what the authorization policy actually is, which then means it is hard to compare it against the code.

In this lecture we take a very general approach, expressing policy in an *authorization logic*. Like in other applications we have seen, the logic is the interface between policy and implementation. It expresses, at a high level of abstraction, who is allowed to do what in a complex system. On one side, this serves as a specification one can reason about rigorously, divorced from an implementation. On the other side, we can use it directly to enforce authorization policies in an implementation by using formal proofs of authorization. Abadi [2003] gives a general overview of

authorization logics. The architecture where formal proofs are used was pioneered by Bauer [2003] under the name of *proof-carrying authorization* (PCA).

Among the direct applications of PCA are the Grey systems [Bauer et al., 2005] for access to offices in Cylab at CMU, and a *proof-carrying file system* that particularly explored issues of efficiency [Garg, 2009, Garg and Pfenning, 2010]. We ignore here a number of practically relevant issues, such as revocation and temporal aspects of authorization. Once added [DeYoung et al., 2007], authorization logic is expressive enough, for example, to express the rules governing access to classified information in the American intelligence community [Garg et al., 2009].

2 Affirmations

Early on this class, we introduced and reasoning in Boolean logic, with connectives such as conjunction, disjunction, implication, etc. Then we introduced so-called *modal operators*, $[\alpha]Q$, $\langle\alpha\rangle Q$ that speak about programs, and $\Box P$ that guarantees validity of P (that is, truth in all possible states).

In order to express access control policies *logically*, we need a new kind of modal operator that expresses an affirmation, A **says** P (*principal* A *says proposition* P). Principals A, B, C, etc. can stand for user ids like $admin$, fp, or $hemant$ Propositions include atomic propositions, conjunction, implication, quantifiers, and other connectives as we need them.

As an example, we use the Grey [Bauer et al., 2005] system that is used to control access to offices in Cylab at CMU. There is an administrator (principal $admin$) that sets policies, and individual professors and students identified by their Andrew id. There are also resources, like offices and conference rooms. Among the atomic propositions are the following:

- $\mathsf{mayOpen}(A, R)$. Principal A may open room R.
- $\mathsf{owns}(A, R)$. Principal A owns office R.
- $\mathsf{studentOf}(B, A)$. Principal B is a student of A.

Here are some examples. The administrator may say that fp owns $ghc6017$.

$$admin \ \textbf{says} \ \mathsf{owns}(fp, ghc6017)$$

This is a basic affirmation. A general rule could state that the owner of a room may open it.

$$admin \ \textbf{says} \ (\forall A.\, \forall R.\, \mathsf{owns}(A, R) \rightarrow \mathsf{mayOpen}(A, R))$$

An even more complex policy component would be that any student of the owner of an office, may also open the office. The twist here is that the studentOf relationship should be affirmed by the owner, rather than the administrator.

$$\mathit{admin}\ \mathbf{says}\ (\forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \land \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R))$$

The scope of fp **says** here is intended to be only the studentOf proposition. When the scope is larger, we enclose it in parentheses. We can combine this policy with the affirmation by fp that hemant is his student:

$$\mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp})$$

Under this policy hemant should be able to access $\mathit{ghc6017}$. We can formulate the question whether this is allowed as a sequent in authorization logic, with the policy as antecedents (assumptions) and the query as the succedent. In this particular situation, we would try to prove the sequent

$$\begin{array}{l}(1) : \mathit{admin}\ \mathbf{says}\ (\forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R)),\\ (2) : \mathit{admin}\ \mathbf{says}\ (\forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \land \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R)),\\ (3) : \mathit{admin}\ \mathbf{says}\ \mathsf{owns}(\mathit{fp}, \mathit{ghc6017}),\\ (4) : \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp})\\ \vdash\\ \mathit{admin}\ \mathbf{says}\ \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})\end{array}$$

Here, we have labeled the assumptions so we can reference them in the proof. Reasoning entirely intuitively (to be formalized later in the lecture) we can deduce:

$$\begin{array}{ll}(5) : \mathit{admin}\ \mathbf{says}\ \mathsf{mayOpen}(\mathit{fp}, \mathit{ghc6017}) & \text{from } (1) \text{ and } (3)\\ (6) : \mathit{admin}\ \mathbf{says}\ \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017}) & \text{from } (2), (3), \text{ and } (4)\end{array}$$

The last line (6) is exactly what we are trying to prove. So we have confirmed that hemant may open my office, according to the policy.

3 Constructive Logic

The Boolean logic of the earlier lectures (including propositional logic and dynamic logic) are based on a semantics where every formula has one of two truth values: $\top$ or $\bot$. This is not a good match for authorization logic, as remarked by Abadi [2003]. The first issue that we may be "agnostic" about a proposition. Any proposition that is affirmed should be seen as *extending* what we can deduce, but not be in conflict with what we know so far. Another example is given by

$$A\ \mathbf{says}\ P \to (P \lor A\ \mathbf{says}\ Q)$$

We can read this: if A affirms P then either P must be true, or A affirms any proposition (including $\bot$). If we worked with just two truth values, why would this be valid? Let's assume A **says** P. Now we distinguish two cases for P. If P is true,

then $P \vee A$ **says** Q. If P is false, then $\neg P$ is true. A would affirm any true proposition, so A **says** $\neg P$. But if A **says** P and A **says** $\neg P$, A is mired in a state of internal contradiction and would have to affirm anything. Not good.

In order to avoid these kind of paradoxes, we use an *intutionistic logic* as the basis for reasoning. Intuitionistic logic does not allow us to reason with the law of excluded middle, or to prove P by assuming $\neg P$ and deriving a contradiction (using an indirect proof). Instead, we want the reason access to a resource is granted to be as clear and direct as possible, which is what intuitionistic logic provides. Actually, we go a step further and rule out negation $\neg P$ and falsehood $\bot$ entirely, because it is easy to make intuitively meaningful statements that are wrong. For example, principal *fp* does **not** want $hemant$ to access his office. Stating *fp* **says** $\neg\mathsf{mayOpen}(hemant, ghc6017)$ turns out to be the wrong way to say this. Because if the rest of the policy says that $\mathsf{mayOpen}(hemant, ghc6017)$ then suddenly *fp* affirms *everything*, including that every principal in the system may access his office—clearly not the desired effect.

This change in perspective, from two-valued Boolean logic (also called *classical logic*) to a richer intuitionstic logic has two consequences, one semantic and one syntactic. Semantically, we need to generalize to a so-called *Kripke semantics* where we consider multiple *worlds* in which different propositions may be true. This is a good match for a logic of authorization since at the very least each principal defines a world, and different principals will affirm different propositions. For such a semantics, see, for example, Garg [2008]. Syntactically, it is surprisingly simple: we restrict the succedent of a sequent to be exactly one formula. We therefore write $\Gamma \vdash \delta$. That this actually works was one of Gentzen's [1935] profound insights.

We actually make the lack of a mathematical semantics a philosophical principle. We think of the meanings of formulas as given by their possible derivations in the sequent calculus. In other words, the right and left rules of the sequent calculus themselves define the meaning of each logical constant, connective, and modality. In the context of authorization, this can be justified since it is ultimately a formal proof that is used to claim and check authorization—we don't appeal to an external semantics. For a fuller development of this viewpoint, see, for example, Dummett [1991] and Martin-Löf [1983].

As we will in particular see in the next lecture, the properties of the logic change in fundamental ways when we restrict the succedents to be just a single formula.

The rules we get otherwise just reflect the earlier ones.

$$\frac{}{\Gamma, P \vdash P}\ \mathsf{id}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ {\to}R \qquad \frac{\Gamma \vdash P \quad \Gamma, Q \vdash \delta}{\Gamma, P \to Q \vdash \delta}\ {\to}L$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ {\wedge}R \qquad \frac{\Gamma, P, Q \vdash \delta}{\Gamma, P \wedge Q \vdash \delta}\ {\wedge}L$$

$$\frac{\Gamma \vdash P(y) \quad y \notin (\Gamma, \forall x.\, P(x))}{\Gamma \vdash \forall x.\, P(x)}\ \forall R^y \qquad \frac{\Gamma, P(c) \vdash \delta}{\Gamma, \forall x.\, P(x) \vdash \delta}\ \forall L$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ {\vee}R_1 \quad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ {\vee}R_2 \qquad \frac{\Gamma, P \vdash \delta \quad \Gamma, Q \vdash \delta}{\Gamma, P \vee Q \vdash \delta}\ {\vee}L$$

For the quantifiers, recall our convention of writing $P(x)$ and then $P(c)$ for the result of substituting c for x in $P(x)$. Quantification was previously over integers, but here we think of principals, rooms, etc. c could also be a variable y that was introduced by a $\forall R$ rule.

The most immediate effect of reasoning intuitionistically is perhaps on disjunction. We can no longer prove, for example $p \vee (p \to q)$ because we have to decide between one of the disjuncts instead of carrying both as succedents.

In these rules there is a main formula among the antecedents to which a left rule is applied. This formula may be needed again so it can be kept among the antecedents if so desired. This is particularly useful for $\forall L$. To anticipate a bit the next lecture, it is no longer the case that all rules are invertible, the way it is in Boolean propositional logic. So proof search is significantly more difficult than in Boolean logic, and not just because of the quantifiers.

4 Affirmations

We haven't yet discussed A **says** P, the central modality of authorization logic. The key insight is that when we try to prove $\Gamma \vdash A$ **says** P we need to proceed with our reasoning from the perspective of principal A. But what does this mean? First, principal A should be willing to affirm any proposition that is *true*. Second, anything that A says is also available to reason with. On the other hand, if B **says** Q is in Γ for some $B \neq A$, the proposition Q is not available to A: this is something that B affirms, but not necessarily A.

Formalizing this reasoning is not entirely straightforward, and there are different approaches. The one we choose is due to Garg and Pfenning [2006], primarily due to it simplicity. We distinguish a succedent expressing that a proposition is

true (written P true) and another that A affirms a proposition P (written A **aff** P). We often omit the "true" notation, but always write A **aff** P to use an affirmation. Formally:

$$\text{Succedent} \quad \delta \quad ::= \quad P \text{ true} \mid A \textbf{ aff } P$$

The first rule says that in order to prove that A **says** P is true, we need to prove that A affirms P.

$$\frac{\Gamma \vdash A \textbf{ aff } P}{\Gamma \vdash (A \textbf{ says } P) \text{ true}} \textbf{ says}R$$

This may seem redundant, but it exposes the principal A and the proposition P. It is similar to the rule $\to R$, where the implication $P \to Q$ is turned into $P \vdash Q$, opening up P and Q to further inferences.

The next rule expresses that if P is true, then A is willing to affirm that.

$$\frac{\Gamma \vdash P \text{ true}}{\Gamma \vdash A \textbf{ aff } P} \textbf{ aff}$$

This rule is like "peeling an onion", moving entirely now into A's head, reasoning from their perspective.

The left rule for A **says** P jumps directly to P, both of which are propositions and therefore can appear among the antecedents. But we can make this transition only if we are currently reasoning from A's perspective, that is, if the succedent is A **aff** Q for some Q.

$$\frac{\Gamma, P \vdash A \textbf{ aff } Q}{\Gamma, A \textbf{ says } P \vdash A \textbf{ aff } Q} \textbf{ says}L$$

5 Some Axioms

Before we go back to our motivating example, we can analyze some of the properties of affirmations.

$$\vdash P \to A \textbf{ says } P$$

As we have said, any principal (here A) is willing to affirm any true proposition (here P), so we should be able to prove this. As usual, we build the proof bottom-up; we show here only the result.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\,\text{id}}{P \vdash A \textbf{ aff } P}\,\textbf{aff}}{P \vdash A \textbf{ says } P}\,\textbf{says}R}{\cdot \vdash P \to A \textbf{ says } P}\,\to R$$

The implication in the other direction should not be valid: just because A says P that doesn't mean P is actually true. We show that it is not derivable.

$\not\vdash (A \textbf{ says } p) \to p$

$$\dfrac{\dfrac{\textsf{XXX}}{A \textbf{ says } p \vdash p}}{\cdot \vdash (A \textbf{ says } p) \to p}\ {\to}R$$

As a first step, only ${\to}R$ is applicable; for the second no rule is.

We can also "distribute" A **says** over an implication. This captures that if A affirms $P \to Q$ and P, then it also affirms Q.

$\vdash A \textbf{ says } (P \to Q) \to (A \textbf{ says } P \to A \textbf{ says } Q)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\ \textsf{id} \quad \dfrac{}{Q \vdash Q}\ \textsf{id}}{P \to Q, P \vdash Q}\ {\to}L}{P \to Q, P \vdash A \textbf{ aff } Q}\ \textbf{aff}}{A \textbf{ says } (P \to Q), A \textbf{ says } P \vdash A \textbf{ aff } Q}\ \textbf{says}L \times 2}{A \textbf{ says } (P \to Q), A \textbf{ says } P \vdash A \textbf{ says } Q}\ \textbf{says}R}{\vdash A \textbf{ says } (P \to Q) \to (A \textbf{ says } P \to A \textbf{ says } Q)}\ {\to}R \times 2$$

Iterating A **says** twice is the same as just affirming just once.

$\vdash A \textbf{ says } (A \textbf{ says } P) \to A \textbf{ says } P$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\ \textsf{id}}{P \vdash A \textbf{ aff } P}\ \textbf{aff}}{A \textbf{ says } P \vdash A \textbf{ aff } P}\ \textbf{says}L}{A \textbf{ says } (A \textbf{ says } P) \vdash A \textbf{ aff } P}\ \textbf{says}L}{A \textbf{ says } (A \textbf{ says } P) \vdash A \textbf{ says } P}\ \textbf{says}R}{\vdash A \textbf{ says } (A \textbf{ says } P) \to A \textbf{ says } P}\ {\to}R$$

The other direction of this implication is an instance of the first axiom. On the other hand, for different principals A and B we cannot prove

$$\not\vdash A \textbf{ says } p \to B \textbf{ says } p$$

These axioms (together with those for intuitionistic logic) are complete for implication and affirmation and identify this as a generalization of *lax logic* [Fairtlough and Mendler, 1997]. Instead of a single modality $\bigcirc P$ we have a whole family of

such modalities, indexed by principals. From the perspective of functional programming, each modality "A **says** $-$" is a strong monad [Moggi, 1989, Wadler, 1992]. Here, it relativizes reasoning to each principal; in functional programming monads can isolate the pure part of the language from effects. The connection to modal logics is further explored by Pfenning and Davies [2001]. A summary of the axioms is in Figure 1.

$$\vdash P \to A \textbf{ says } P$$
$$\vdash A \textbf{ says } (P \to Q) \to (A \textbf{ says } P \to A \textbf{ says } Q)$$
$$\vdash A \textbf{ says } (A \textbf{ says } P) \to A \textbf{ says } P$$

$$\not\vdash (A \textbf{ says } p) \to p$$

Figure 1: Axioms for Authorization Logic

6 An Example of Authorization

We return to our motivating example, where we have labeled each of the antecedents as before.

$(1) : \mathit{admin}$ **says** $(\forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R))$,
$(2) : \mathit{admin}$ **says** $(\forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \wedge \mathit{fp}$ **says** $\mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R))$,
$(3) : \mathit{admin}$ **says** $\mathsf{owns}(\mathit{fp}, \mathit{ghc6017})$,
$(4) : \mathit{fp}$ **says** $\mathsf{studentOf}(\mathit{hemant}, \mathit{fp})$
$\vdash$
admin **says** $\mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$

We highlighted in blue the focus of the next inference. Using **says**R, we reduce this in one step to proving the sequent

$(1) : \mathit{admin}$ **says** $(\forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R))$,
$(2) : \mathit{admin}$ **says** $(\forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \wedge \mathit{fp}$ **says** $\mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R))$,
$(3) : \mathit{admin}$ **says** $\mathsf{owns}(\mathit{fp}, \mathit{ghc6017})$,
$(4) : \mathit{fp}$ **says** $\mathsf{studentOf}(\mathit{hemant}, \mathit{fp})$
$\vdash$
admin **aff** $\mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$

This unlocks the affirmations by admin among the antecedents, so applying **says**L three times we get

$(1)'$: $\forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R)$,
$(2)'$: $\forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \wedge \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R)$,

$(3)'$: $\mathsf{owns}(\mathit{fp}, \mathit{ghc6017})$,
(4) : $\mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp})$

$\vdash$

$\mathit{admin}\ \mathbf{aff}\ \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$

Note that fp's affirmation cannot be unlocked, since $\mathit{admin} \neq \mathit{fp}$. Now we can apply $\forall L$ three times, instantiating A, B, and C with fp, hemant, and $\mathit{ghc6017}$, respectively.

$(1)'$: $\forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R)$,
$(2)''$: $\mathsf{owns}(\mathit{fp}, \mathit{ghc6017}) \wedge \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp}) \to \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$,

$(3)'$: $\mathsf{owns}(\mathit{fp}, \mathit{ghc6017})$,
(4) : $\mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp})$

$\vdash$

$\mathit{admin}\ \mathbf{aff}\ \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$

At this point we can apply $\to L$ to the antecedent $(2)''$, proving the conjunction by $(3)'$ and (4) and obtaining the new line (5).

$(1)'$: $\forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R)$,
$(2)''$: $\mathsf{owns}(\mathit{fp}, \mathit{ghc6017}) \wedge \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp}) \to \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$,

$(3)'$: $\mathsf{owns}(\mathit{fp}, \mathit{ghc6017})$,
(4) : $\mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp})$
(5) : $\mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$

$\vdash$

$\mathit{admin}\ \mathbf{aff}\ \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$

Now we can apply the rule of affirmation, followed by the identity to complete the proof.

7 Summary

Principals	A, B, C		
Formulas	P, Q	$::=$	$p \mid P \wedge Q \mid P \to Q \mid P \vee Q \mid \forall x.\, P(x) \mid A\ \mathbf{says}\ P$
Succedents	δ	$::=$	$P\ \mathsf{true} \mid A\ \mathbf{aff}\ P$

$$\frac{}{\Gamma, P \vdash P \text{ true}}\ \mathsf{id}$$

$$\frac{\Gamma, P \vdash Q \text{ true}}{\Gamma \vdash P \to Q \text{ true}}\ {\to}R \qquad \frac{\Gamma \vdash P \quad \Gamma, Q \vdash \delta}{\Gamma, P \to Q \vdash \delta}\ {\to}L$$

$$\frac{\Gamma \vdash P \text{ true} \quad \Gamma \vdash Q \text{ true}}{\Gamma \vdash P \wedge Q \text{ true}}\ {\wedge}R \qquad \frac{\Gamma, P, Q \vdash \delta}{\Gamma, P \wedge Q \vdash \delta}\ {\wedge}L$$

$$\frac{\Gamma \vdash P(y) \text{ true} \quad y \notin (\Gamma, \forall x.\, P(x))}{\Gamma \vdash \forall x.\, P(x) \text{ true}}\ \forall R^y \qquad \frac{\Gamma, P(c) \vdash \delta}{\Gamma, \forall x.\, P(x) \vdash \delta}\ \forall L$$

$$\frac{\Gamma \vdash P \text{ true}}{\Gamma \vdash P \vee Q \text{ true}}\ {\vee}R_1 \quad \frac{\Gamma \vdash Q \text{ true}}{\Gamma \vdash P \vee Q \text{ true}}\ {\vee}R_2 \qquad \frac{\Gamma, P \vdash \delta \quad \Gamma, Q \vdash \delta}{\Gamma, P \vee Q \vdash \delta}\ {\vee}L$$

$$\frac{\Gamma \vdash A\ \mathbf{aff}\ P}{\Gamma \vdash (A\ \mathbf{says}\ P) \text{ true}}\ \mathbf{says}R \qquad \frac{\Gamma, P \vdash A\ \mathbf{aff}\ Q}{\Gamma, A\ \mathbf{says}\ P \vdash A\ \mathbf{aff}\ Q}\ \mathbf{says}L$$

$$\frac{\Gamma \vdash P \text{ true}}{\Gamma \vdash A\ \mathbf{aff}\ P}\ \mathbf{aff}$$

Figure 2: Affirmation Logic

References

Martín Abadi. Logic in access control. In *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, pages 228–233, Ottawa, Canada, June 2003. IEEE Computer Society Press.

Lujo Bauer. *Access Control for the Web via Proof-Carrying Authorization*. PhD thesis, Princeton University, November 2003.

Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, pages 431–445, Singapore, September 2005. Springer Verlag LNCS 3650.

Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. Technical Report CMU-CS-07-166, Carnegie Mellon University, Department of Computer Science, December 2007. Revised February 2008.

Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

M. Fairtlough and M.V. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, August 1997.

Deepak Garg. Principal-centric reasoning in constructive authorization logic. In *Workshop on Intuitionistic Modal Logic and Applications (IMLA'08)*, July 2008. Extended and revised version available as Technical Report CMU-CS-09-120, April 2009.

Deepak Garg. *Proof Theory for Authorization Logic and Its Application to a Practical File System*. PhD thesis, Carnegie Mellon University, December 2009. Available as Technical Report CMU-CS-09-168.

Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In J. Guttman, editor, *Proceedings of the 19th Computer Security Foundations Workshop (CSFW'06)*, pages 283–293, Venice, Italy, July 2006. IEEE Computer Society Press.

Deepak Garg and Frank Pfenning. A proof-carrying file system. In D.Evans and G.Vigna, editors, *Proceedings of the 31st Symposium on Security and Privacy (Oakland 2010)*, pages 349–364, Berkeley, California, May 2010. IEEE. Extended version available as Technical Report CMU-CS-09-123, June 2009.

Deepak Garg, Frank Pfenning, Denis Serenyi, and Brian Witten. A logical representation of common rule for controlling access to classified information. Technical Report CMU-CS-09-139, Carnegie Mellon University, June 2009.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983. URL http://www.hf.uio.no/ifikk/forskning/publikasjoner/tidsskrifter/njpl/vol1no1/meaning.pdf.

Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

Philip Wadler. The essence of functional programming. In *Conference Record of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, January 1992. ACM Press.

Lecture Notes on
Proof Search in Authorization Logic

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

Lecture 16
October 31, 2024

1 Introduction

In Boolean logic, the sequent calculus was a good basis for proof search (at least on the small scale) because all rules were sound and invertible. This is no longer the case for intuitionistic logic, whether extended with affirmation or not. In fact, Boolean propositional logic is just about the only logic where we can arrange for all rules to be invertible. Our task then is to find out which rules *are* invertible and which are not. This gives rise to a classification of logical connectives based on the invertibility of their right and left rules. This can be done generally for logics that admit a sequent calculus formulation. It was first discovered for *linear logic* [Girard, 1987] by Andreoli [1992] but since been applied to many other ones, including those with a lax modality [Liang and Miller, 2009, Watkins et al., 2002].

Our first order of business, then, is to identify invertibility properties, followed by proof strategies based on them.

2 Inversion

Since it is a pervasively used connective, we start with implication.

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ {\to}R \qquad\qquad \frac{\Gamma \vdash P \quad \Gamma, Q \vdash \delta}{\Gamma, P \to Q \vdash \delta}\ {\to}L$$

invertible not invertible

It turns out the right rule is invertible, while the left rule is not. To see that the left rule is *not* invertible, we only need a counterexample. Consider

$$p \to q, q \to r \vdash p \to r$$

Trying to apply the left rule for implication to the first antecedent results in a sequent that cannot be derived.

$$\dfrac{\dfrac{\mathsf{XXX}}{\cdot \vdash q} \quad \dfrac{\mathsf{XXX}}{r \vdash p}}{q \to r \vdash p}\,{\to}L \qquad \vdots}{p \to q, q \to r \vdash p \to r}\,{\to}L$$

We suspect that the right rule for implication is invertible, but how do we prove it? Previously, we used a *semantic argument*, using the validity of a sequent. Here, such a direct argument is not available—we would have to introduce a semantics which is not straightforward. But there is also a syntactic technique, using the *admissibility of cut*. So we make a brief detour to introduce it.

When presented as a rule

$$\dfrac{\Gamma \vdash P \quad \Gamma, P \vdash \delta}{\Gamma \vdash \delta}\ \mathsf{cut}$$

its meaning is clear: if we can prove P we are allowed to assume it in further reasoning. For bottom-up proof search it is somewhat problematic because we have to determine a useful P, which could be *any* formula. So we use it only under controlled circumstances.

Gentzen [1935] showed that the rule of cut is redundant for both classical (Boolean) and intuitionistic logic. What do we mean by "redundant"? More technically, we call a rule *admissible* if it is both sound and whenever the premises can be derived, so can the conclusion *without using the rule*. We write an admissible rule with a dashed line:

$$\dfrac{\Gamma \vdash P \quad \Gamma, P \vdash \delta}{\Gamma \vdash \delta}\ \mathsf{cut}$$

We won't go into detail how to show that cut is admissible. This is covered in many articles (including Gentzen's original one) and also in *15-317 Constructive Logic* in somewhat more modern notation (see Lecture 8).

Here we concentrate on how to use the admissibility of cut to obtain other properties. First, it is convenient to have an alternative version that is also admissible and more suitable to top-down reasoning.

$$\dfrac{\Gamma_1 \vdash P \quad \Gamma_2, P \vdash \delta}{\Gamma_1, \Gamma_2 \vdash \delta}\ \mathsf{cut}'$$

We want to prove that $\to R$ is invertible. That is, we want to show that

$$\dfrac{\Gamma \vdash P \to Q}{\Gamma, P \vdash Q}\ {\to}R^{-1}$$

is admissible. Step-by-step: first, we have a succedent $P \to Q$ in the premise, so we should try to cut it with a sequent with antecedent $P \to Q$ with some Γ' and δ.

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash P \to Q \quad \Gamma', P \to Q \vdash \delta\end{array}}{\Gamma, P \vdash Q}\ \mathsf{cut}'$$

What could we choose for Γ' and δ? It is pretty obvious from the conclusion: $\Gamma' = P$ and $\delta = Q$. We can then complete the derivation of the second premise.

$$\dfrac{\Gamma \vdash P \to Q \qquad \dfrac{\dfrac{}{P \vdash P}\ \mathsf{id} \quad \dfrac{}{Q \vdash Q}\ \mathsf{id}}{P, P \to Q \vdash Q}\ {\to}L}{\Gamma, P \vdash Q}\ \mathsf{cut}'$$

We see in the second premise everything is proved, so this derivation shows that ${\to}R^{-1}$ is admissible.

We can prove other rules invertible as well, with clever uses of cut′. On our fragment (excluding affirmation for now), the invertible rules are $\wedge R$, $\wedge L$, $\vee L$, $\forall R$. The noninvertible rules are ${\to}L$, $\vee R_1$, $\vee R_2$, and $\forall L$.

A first simple strategy emerges: apply all the invertible rules in some arbitrary, unspecified order until we reach a sequent where either identity applies, or we have to make a choice between noninvertible rules. This choice typically then requires backtracking, in case we make a wrong choice.

We can express such a strategy with three judgment forms, two that force only invertible rules to be used on the left or right, and one that requires a choice. Since it is not our ultimate destination, we elide this here and refer the interest reader to Lecture 15 of the course on constructive logic.

3 Inversion for Affirmation

An empirical observation is that if the right rule for a connective is invertible then the left rule is not and vice versa. This is apparently violated by conjunction which is invertible on both sides. This is because there are actually two forms of conjunction hiding under the symbol "$\wedge$" that are indistinguishable with respect to provability.

We call connectives whose right rule is invertible *negative connectives* and the ones whole left rule is invertible are *positive connectives*. The inversion phase of proof search then applies negative right rules and positive left rules.

But what about affirmation? There is something rather strange going on because in the right rule we go from $(A\ \mathbf{says}\ P)\ \mathsf{true}$ to $A\ \mathbf{aff}\ P$ and in the left rule we jump directly from $A\ \mathbf{says}\ P$ to P. It turns out that the affirmation modality

signifies a change in polarity. One way to state that is that A **says** P is negative, but the judgment underneath, A **aff** P is positive. That means, the right rule is invertible, but the rule of affirmation is not.

$$\frac{\Gamma \vdash A \text{ } \mathbf{aff} \text{ } P}{\Gamma \vdash (A \text{ } \mathbf{says} \text{ } P) \text{ true}} \text{ } \mathbf{says}R \qquad \frac{\Gamma \vdash P \text{ true}}{\Gamma \vdash A \text{ } \mathbf{aff} \text{ } P} \text{ } \mathbf{aff}$$

invertible not invertible

As for the left rules, because A **says** P is negative, it's left rules is not invertible. However, if the succedent has the form A **aff** Q we can always apply the rule since P is a stronger assumption than A **says** P. So we put "invertible" in quotation: we can apply the rule when possible, but we cannot always apply it when A **says** P is an antecedent.

$$\frac{\Gamma, P \vdash A \text{ } \mathbf{aff} \text{ } Q}{\Gamma, A \text{ } \mathbf{says} \text{ } P \vdash A \text{ } \mathbf{aff} \text{ } Q} \text{ } \mathbf{says}L$$

"invertible"

Now we also see why A **aff** P does not appear as a judgment among the antecedents: it is positive, and therefore the corresponding rule can be applied immediately and the stepping stone be omitted.

$$\frac{\dfrac{\Gamma, P \vdash A \text{ } \mathbf{aff} \text{ } Q}{\Gamma, \text{“}A \text{ } \mathbf{aff} \text{ } P\text{”} \vdash A \text{ } \mathbf{aff} \text{ } Q}}{\Gamma, A \text{ } \mathbf{says} \text{ } P \vdash A \text{ } \mathbf{aff} \text{ } Q} \text{ } \mathbf{says}L$$

4 Focusing

If we look at the example from the last lecture we see that the only invertible step is the right rule for **says**, followed by stripping the "$admin$ **says** " prefix from (1), (2), and (3) using **says**L.

$(1) : admin \text{ } \mathbf{says} \text{ } (\forall A.\, \forall R.\, \mathsf{owns}(A, R) \rightarrow \mathsf{mayOpen}(A, R))$,
$(2) : admin \text{ } \mathbf{says} \text{ } (\forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \wedge fp \text{ } \mathbf{says} \text{ } \mathsf{studentOf}(B, A) \rightarrow \mathsf{mayOpen}(B, R))$,
$(3) : admin \text{ } \mathbf{says} \text{ } \mathsf{owns}(fp, ghc6017)$,
$(4) : fp \text{ } \mathbf{says} \text{ } \mathsf{studentOf}(hemant, fp)$

$\vdash$

$admin \text{ } \mathbf{says} \text{ } \mathsf{mayOpen}(hemant, ghc6017)$

The resulting sequent below has no invertible rule we can blindly apply.

$(1)' : \forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R)$,
$(2)' : \forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \wedge \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R)$,
$(3)' : \mathsf{owns}(\mathit{fp}, \mathit{ghc6017})$,
$(4) : \mathit{fp}\ \mathbf{says}\ \mathsf{studentOf}(\mathit{hemant}, \mathit{fp})$
$\vdash$
$\mathit{admin}\ \mathbf{aff}\ \mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$

For example, we could instantiate the quantifier $\forall A$ in $(1)'$. But still, everything remains negative and we could instantiate either the $\forall R$ we have uncovered or the $\forall A$ quantifier in (2). We see that at every step we have to choose between a number of alternatives. This can lead to a lot of backtracking if the choices are incorrect.

Focusing [Andreoli, 1992] is the idea that we can pick a negative antecedent or a positive succedent and continue to apply noninvertible rules to this particular formula until we reach either an atomic formula or the polarity switches. For example, if we guess $(2)'$ we would apply $\forall L$ three times, followed by $\to L$. Let's treat conjunction as positive, which we continue with that as well and succeed when we find the needed assumptions.[1] All that reasoning adds $\mathsf{mayOpen}(\mathit{hemant}, \mathit{ghc6017})$ to the assumptions and we have to make another choice. At this point we want to choose the rule of affirmation followed by the identity to complete the proof.

We don't write out the sequent calculus for focusing in its most general form, but you may refer to Lecture 17 of the constructive logic course notes for details. We write $\Gamma, [P] \vdash \delta$ for a formula P in left focus and $\Gamma \vdash [Q]$ for a formula in right focus. In this kind of sequent just one formula can be in focus, and rules can be applied only to the formula in focus. The rules with no focus are the invertible rules.

This gives us the following rules. We work with following polarities below. Except for atoms and conjunction (which we choose to be positive), they are determined by the inversion properties of the connectives.

$$\begin{array}{lllll} \text{Negative} & P^-, Q^- & ::= & P \to Q \mid \forall x.\, P(x) \mid A\ \mathbf{says}\ P \\ \text{Positive} & P^+, Q^+ & ::= & p \mid P \wedge Q \mid P \vee Q \end{array}$$

First the rules to *initiate* a phase of focusing.

$$\frac{\Gamma \vdash [Q^+]}{\Gamma \vdash Q^+}\ \mathbf{focus}R \qquad \frac{P^- \in \Gamma \quad \Gamma, [P^-] \vdash \delta}{\Gamma \vdash \delta}\ \mathbf{focus}L$$

In the left rule we focus on a *copy* of P^- because we may need this formula again. In the examples we sometimes drop the extra copy if we anticipate (or know) it will not be needed again.

[1] Actually, not quite. The affirmation "*fp* **says**" forces us to stop and make another explicit choice.

The rule for affirmation is an additional transition rule. Next, the logical rules. For brevity, we mostly omit "true" in the succedent if it is just a formula. The judgment $\Gamma \vdash \delta$, $\Gamma, [P] \vdash \delta$ and $\Gamma \vdash [Q]$ are mutually exclusive, that is, there may be no focused formula in Γ or δ.

$$\frac{}{\Gamma, p \vdash [p]}\ \mathsf{id}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ {\to}R \qquad \frac{\Gamma \vdash [P] \quad \Gamma, [Q] \vdash \delta}{\Gamma, [P \to Q] \vdash \delta}\ {\to}L$$

$$\frac{\Gamma \vdash [P] \quad \Gamma \vdash [Q]}{\Gamma \vdash [P \wedge Q]}\ {\wedge}R \qquad \frac{\Gamma, P, Q \vdash \delta}{\Gamma, P \wedge Q \vdash \delta}\ {\wedge}L$$

$$\frac{\Gamma \vdash P(y) \quad y \notin \Gamma, P(x)}{\Gamma \vdash \forall x.\, P(x)}\ {\forall}R^y \qquad \frac{\Gamma, [P(c)] \vdash \delta}{\Gamma, [\forall x.\, P(x)] \vdash \delta}\ {\forall}L$$

$$\frac{\Gamma \vdash [P]}{\Gamma \vdash [P \vee Q]}\ {\vee}R_1 \quad \frac{\Gamma \vdash [Q]}{\Gamma \vdash [P \vee Q]}\ {\vee}R_2 \qquad \frac{\Gamma, P \vdash \delta \quad \Gamma, Q \vdash \delta}{\Gamma, P \vee Q \vdash \delta}\ {\vee}L$$

Next, the rules to complete a focusing phase (still postponing affirmations). We say that we *blur the focus*.

$$\frac{\Gamma, P^+ \vdash \delta}{\Gamma, [P^+] \vdash \delta}\ \mathsf{blur}L \qquad \frac{\Gamma \vdash Q^-}{\Gamma \vdash [Q^-]}\ \mathsf{blur}R$$

Finally, the rules for affirmation. They incorporate some phase transitions because of the polarity shift intrinsic to the modality.

$$\frac{\Gamma \vdash A\ \mathbf{aff}\ P}{\Gamma \vdash (A\ \mathbf{says}\ P)\ \mathsf{true}}\ \mathbf{says}R \qquad \frac{\Gamma, P \vdash A\ \mathbf{aff}\ Q}{\Gamma, [A\ \mathbf{says}\ P] \vdash A\ \mathbf{aff}\ Q}\ \mathbf{says}L$$

$$\frac{\Gamma \vdash [P]\ \mathsf{true}}{\Gamma \vdash A\ \mathbf{aff}\ P}\ \mathbf{aff}$$

Our motivating example is too lengthy to write out formally with these rules for focusing, but we can try a small example and see how focusing reduces nondeterminism. Consider

$$p \to q, p \to (q \to r) \vdash p \to r$$

After the obligatory right inversion, we arrive at

$$p, p \to q, p \to (q \to r) \vdash r$$

In this situation, we could in principle try to focus on $p \to q$, on $p \to (q \to r)$ or r. Let's try right focus first. We immediately fail because r is not among the antecedents.

$$\dfrac{\dfrac{\mathsf{XXX}}{p, p \to q, p \to (q \to r) \vdash [r]}}{p, p \to q, p \to (q \to r) \vdash r}\ \mathbf{focus}R$$

Next we try to focus on $p \to (q \to r)$.

$$\dfrac{\dfrac{}{p, p \to q \vdash [p]}\ \mathsf{id} \quad \dfrac{\dfrac{\mathsf{XXX}}{p, p \to q \vdash [q]} \quad \dfrac{\vdots}{p, p \to q, [r] \vdash r}}{p, p \to q, [q \to r] \vdash r}\ {\to}L}{\dfrac{p, p \to q, [p \to (q \to r)] \vdash r}{p, p \to q, p \to (q \to r) \vdash r}\ \mathbf{focus}L}\ {\to}L$$

Note that after we decided which antecedent to focus on, everything was determined. We fail, because q is not among the antecedents.

We cannot focus on p on the left because it is positive, but we can try one final option, namely to focus on $p \to q$.

$$\dfrac{\dfrac{}{p, p \to (q \to r) \vdash [p]}\ \mathsf{id} \quad \dfrac{\dfrac{\vdots}{p, q, p \to (q \to r) \vdash r}}{p, [q], p \to (q \to r) \vdash r}\ \mathbf{blur}L}{\dfrac{p, [p \to q], p \to (q \to r) \vdash r}{p, p \to q, p \to (q \to r) \vdash r}\ \mathbf{focus}L}\ {\to}L$$

Now it is possible to focus on $p \to (q \to r)$ because both p and q are among the antecedents. This will add r to the antecedents and we can finally successfully focus on the succedent.

Remarkably, with focusing there is just a single proof (assuming we don't copy the formula in focus).

Using inversion and focusing is sound and complete with respect to the sequent calculus. Soundness is important

Theorem 1 (Soundness and Completeness of Inversion and Focusing) *$\Gamma \vdash \delta$ in the sequent calculus if and only if $\Gamma \vdash \delta$ in the calculus with inversion and focusing.*

Proof: Soundness is straightforward, since we only restrict the application of certain rules.

The proof of completeness is quite complex, and not just because of affirmations. See Liang and Miller [2009] for a blueprint that can be adapted to this authorization logic. □

In the architecture of proof-carrying authorization we have to contend with the fact that more complex policies engender more difficult theorem proving problems. In the next lecture we will identify a fragment of authorization logic that represents a reasonable compromise between expressiveness and difficulty of proving. It is inspired by Horn clauses, but goes beyond it due the presence of affirmations.

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

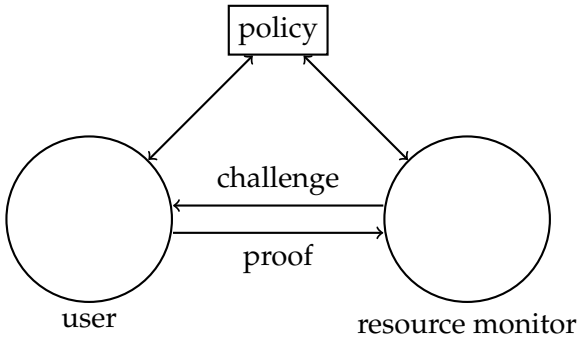
Lecture Notes on
Proof Representation

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

Lecture 17
November 5, 2024

1 Introduction

The picture below illustrates the general proof-carrying architecture we have considered in the last two lectures.



In this particular use of authorization logic, the user bears the burden of proof. If the policy is not particularly complex, the resource monitor itself could construct a proof instead, either explicitly or implicitly via an implementation that has been proved correct against the policy.

One advantage of this architecture is that new affirmations can enter the picture dynamically. For example, when *myra* stands in front of my office she might contact me to obtain an affirmation from me that she is my student and can therefore enter it. Such an affirmation would come in the form of a *signed certificate*, which we will discuss in the next lecture.

If we stick to the architecture as depicted, it is the user's responsibility to produce a proof of the challenge formula and the resource monitor's responsibility to check the proof it received. In the last lecture we talked about some strategies

for finding proofs; today we'll talk about how to represent them so they can be communicated to the resource monitor and then checked.

The mainstay of the whole idea is that the policy expresses the intended authorization policy in a straightforward and understandable way.

2 Proof Terms for Intuitionistic Propositional Logic

We start with the proof terms for propositional logic. The basic idea is to annotate a sequent

$$P_1, \ldots, P_n \vdash Q$$

with *proof terms* M_i and N such that

$$M_1 : P_1, \ldots, M_n : P_n \vdash N : Q$$

The initial sequent we try to prove has the form

$$c_1 : P_1, \ldots, c_n : P_n \vdash ? : Q$$

where c_i are *signed certificates* that serve as justifications for the antecedents. As we proceed with the proof, we may create additional antecedents $M : P$ and eventually justify the conclusion with a proof term $N : Q$. The term N should have enough information to check that the initial sequent has a proof.

For the remainder of this lecture, we will use Γ to also stand for a sequent where each antecedent has a suitable justification.

Conjunction. We start with conjunction:

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \wedge R \qquad \frac{\Gamma, P, Q \vdash \delta}{\Gamma, P \wedge Q \vdash \delta}\ \wedge L$$

We see that the justification for $P \wedge Q$ should be a pair, consisting of a justification for P and one for Q.

$$\frac{\Gamma \vdash M : P \quad \Gamma \vdash N : Q}{\Gamma \vdash \langle M, N \rangle : P \wedge Q}\ \wedge R$$

To check that $\langle M, N \rangle$ is a proof of $P \wedge Q$ we just need to check that M is a proof of P and N is a proof of Q.

How does the left rule work? Assume that M is a justification for $P \wedge Q$. That means that M represents a pair, and its first component should be a proof of P and its second component a proof of Q.

$$\frac{\Gamma, M.\pi_1 : P, M.\pi_2 : Q \vdash N : \delta}{\Gamma, M : P \wedge Q \vdash N : \delta}\ \wedge L$$

Identity. Let's complete this first analysis with the identity rule. Because all antecedents have a justification, we just use that as our justification of the (identical) succedent.

$$\frac{}{\Gamma, P \vdash P}\ \mathsf{id} \qquad \frac{}{\Gamma, M : P \vdash M : P}\ \mathsf{id}$$

At this point we can already write out a small example.

$$\dfrac{\dfrac{\dfrac{}{P, Q \vdash Q}\ \mathsf{id} \quad \dfrac{}{P, Q \vdash P}\ \mathsf{id}}{P, Q \vdash Q \wedge P}\ \wedge R}{P \wedge Q \vdash Q \wedge P}\ \wedge L$$

We now annotate this in several steps, leaving question marks where information is still to be filled in

$$\dfrac{\dfrac{\dfrac{}{? : P, ? : Q \vdash ? : Q}\ \mathsf{id} \quad \dfrac{}{? : P, ? : Q \vdash ? : P}\ \mathsf{id}}{? : P, ? : Q \vdash ? : Q \wedge P}\ \wedge R}{x : P \wedge Q \vdash ? : Q \wedge P}\ \wedge L$$

Here, x is the initial justification for the antecedent $P \wedge Q$ which could be a signed certificate, or perhaps a variable that can come from somewhere else. From it, we can construct the justifications for P and Q by projection. These then also flow upward in the derivation.

$$\dfrac{\dfrac{\dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash ? : Q}\ \mathsf{id} \quad \dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash ? : P}\ \mathsf{id}}{x.\pi_1 : P, x.\pi_2 : Q \vdash ? : Q \wedge P}\ \wedge R}{x : P \wedge Q \vdash ? : Q \wedge P}\ \wedge L$$

Now we can copy over the justifications for P and Q in the two applications of identity.

$$\dfrac{\dfrac{\dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash x.\pi_2 : Q}\ \mathsf{id} \quad \dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash x.\pi_1 : P}\ \mathsf{id}}{x.\pi_1 : P, x.\pi_2 : Q \vdash ? : Q \wedge P}\ \wedge R}{x : P \wedge Q \vdash ? : Q \wedge P}\ \wedge L$$

Now we can fill in the proof term for $Q \wedge P$ and propagate it down the derivation.

$$\dfrac{\dfrac{\dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash x.\pi_2 : Q}\ \mathsf{id} \quad \dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash x.\pi_1 : P}\ \mathsf{id}}{x.\pi_1 : P, x.\pi_2 : Q \vdash \langle x.\pi_2, x.\pi_1 \rangle : Q \wedge P}\ \wedge R}{x : P \wedge Q \vdash \langle x.\pi_2, x.\pi_1 \rangle : Q \wedge P}\ \wedge L$$

Already here we might anticipate something that holds on a larger scale. Namely, the proof looks like a piece of code that reverses the elements of a pair. So the term representing a proof is like a program, and the proposition is like its type. We can only scratch the surface of that connection, common called the *Curry-Howard Isomorphism* [Curry, 1934, Howard, 1969]. The CMU course on Constructive Logic investigates this connection in depth.

Implication. The intuitionistic reading of $P \to Q$ is as a function from proofs of P to proofs of Q.

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q} \to R \qquad \frac{\Gamma \vdash P \quad \Gamma, Q \vdash \delta}{\Gamma, P \to Q \vdash \delta} \to L$$

Then the proof term for a right rule is a function.

$$\frac{\Gamma, x : P \vdash N : Q}{\Gamma \vdash (\lambda x.\, N) : P \to Q} \to R$$

This notation goes back to Church's λ-calculus [Church and Rosser, 1936] and may be written in concrete syntax as `fn x => N` (Standard ML) or `fun x -> N` (OCaml) or `\x -> N` (Haskell).

The left rule represents function application, written as juxtaposition $N\, M$.

$$\frac{\Gamma \vdash M : P \quad \Gamma, N\, M : Q \vdash O : \delta}{\Gamma, N : P \to Q \vdash O : \delta} \to L$$

Just like for $\wedge L$, the succedent does not change in this rule.

To resume our example: we can finish with an $\to R$ rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash x.\pi_2 : Q}\,\mathsf{id} \quad \dfrac{}{x.\pi_1 : P, x.\pi_2 : Q \vdash x.\pi_1 : P}\,\mathsf{id}}{x.\pi_1 : P, x.\pi_2 : Q \vdash \langle x.\pi_2, x.\pi_1 \rangle : Q \wedge P} \wedge R}{x : P \wedge Q \vdash \langle x.\pi_2, x.\pi_1 \rangle : Q \wedge P} \wedge L}{\vdash \lambda x.\, \langle x.\pi_2, x.\pi_1 \rangle : P \wedge Q \to Q \wedge P} \to R$$

Recall that the left rule for implication is *not invertible*. This means we actually may need the antecedent $P \to Q$ (or, annotated, $N : P \to Q$) again in the proof. We don't explicitly reflect this in $\to L$, or in left rules in general. Instead, by convention, the antecedent we apply a left rule to may be kept or discarded in the premises. If one wants to build a theorem prover that is complete, a detailed analysis of whether antecedents may be discarded would be indicated.

Universal Quantification. Intuitionistically, a proof of $\forall x.\, P(x)$ also represents a function. It takes as argument an element c from the domain of quantification and returns a proof of $P(c)$. Inside a proof, this element c could also be a variable denoting a constant.

$$\frac{\Gamma \vdash P(y) \quad y \notin (\Gamma, \forall x.\, P(x))}{\Gamma \vdash \forall x.\, P(x)}\ \forall R^y \qquad \frac{\Gamma, P(c) \vdash \delta}{\Gamma, \forall x.\, P(x) \vdash \delta}\ \forall L$$

Since it also is a function, we reuse the same notation as for implication. The context of use will provide enough information to disambiguate.

$$\frac{\Gamma \vdash M(y) : P(y) \quad y \notin (\Gamma, \forall x.\, P(x))}{\Gamma \vdash (\lambda x.\, M(x)) : \forall x.\, P(x)}\ \forall R^y \qquad \frac{\Gamma, M\, c : P(c) \vdash N : \delta}{\Gamma, M : \forall x.\, P(x) \vdash N : \delta}\ \forall L$$

Cut. The cut rule introduces a lemma into a proof. With terms, this means we introduce a name for a possibly complex term.

$$\frac{\Gamma \vdash P \quad \Gamma, P \vdash \delta}{\Gamma \vdash \delta}\ \mathsf{cut} \qquad \frac{\Gamma \vdash M : P \quad \Gamma, x : P \vdash N : Q}{\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : Q}\ \mathsf{cut}$$

There is a potential issue with checking applications of cut since the conclusion (and the term $\mathbf{let}\ x = M\ \mathbf{in}\ N$) does not contain P. So we might need to add this to the term. Note that there potentially is another rule for a succedent $A\ \mathbf{aff}\ Q$.

Disjunction. We omit the proof terms for disjunction since our application ultimately does not use disjunction. In brief, the right rule tags members of a sum while the left rule represents a program that distinguishes cases.

3 Proof Terms for Affirmations

The complication for the rules of affirmation is that the left rules for principle A are unlocked for a limited section of the proof. This section needs to be represented explicitly and we use $\{M\}_A$ to represent this scope.

$$\frac{\Gamma \vdash A\ \mathbf{aff}\ P}{\Gamma \vdash (A\ \mathbf{says}\ P)\ \mathsf{true}}\ \mathbf{says}R \qquad \frac{\Gamma \vdash M : A\ \mathbf{aff}\ P}{\Gamma \vdash \{M\}_A : (A\ \mathbf{says}\ P)\ \mathsf{true}}\ \mathbf{says}R$$

The left rule "strips off" the scoping that may be present in the term M and binds a fresh variable x within the current scope.

$$\frac{\Gamma, P \vdash A\ \mathbf{aff}\ Q}{\Gamma, A\ \mathbf{says}\ P \vdash A\ \mathbf{aff}\ Q}\ \mathbf{says}L \qquad \frac{\Gamma, x : P \vdash N : A\ \mathbf{aff}\ Q}{\Gamma, M : A\ \mathbf{says}\ P \vdash \mathbf{let}\ \{x\}_A = M\ \mathbf{in}\ N : A\ \mathbf{aff}\ Q}\ \mathbf{says}L$$

Finally, the rule of affirmation is a judgmental transition (rather than being connected to a particular proposition), so the proof term remains the same.

$$\frac{\Gamma \vdash M : P \ \mathsf{true}}{\Gamma \vdash M : A \ \mathbf{aff} \ P} \ \mathbf{aff}$$

Before we go to our motivating example, we work out a relatively simple one.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\ \mathsf{id} \quad \dfrac{}{Q \vdash Q}\ \mathsf{id}}{P \to Q, P \vdash Q}\ {\to}L}{P \to Q, P \vdash A \ \mathbf{aff} \ Q}\ \mathbf{aff}}{A \ \mathbf{says} \ (P \to Q), A \ \mathbf{says} \ P \vdash A \ \mathbf{aff} \ Q}\ \mathbf{says}L \times 2}{A \ \mathbf{says} \ (P \to Q), A \ \mathbf{says} \ P \vdash A \ \mathbf{says} \ Q}\ \mathbf{says}R}{\vdash A \ \mathbf{says} \ (P \to Q) \to (A \ \mathbf{says} \ P \to A \ \mathbf{says} \ Q)}\ {\to}R \times 2$$

We start by annotating bottom-up, leaving "unknowns" as question marks.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash ? : P}\ \mathsf{id} \quad \dfrac{}{Q \vdash ? : Q}\ \mathsf{id}}{P \to Q, P \vdash ? : Q}\ {\to}L}{P \to Q, P \vdash ? : A \ \mathbf{aff} \ Q}\ \mathbf{aff}}{x : A \ \mathbf{says} \ (P \to Q), y : A \ \mathbf{says} \ P \vdash ? : A \ \mathbf{aff} \ Q}\ \mathbf{says}L \times 2}{x : A \ \mathbf{says} \ (P \to Q), y : A \ \mathbf{says} \ P \vdash ? : A \ \mathbf{says} \ Q}\ \mathbf{says}R}{\vdash ? : A \ \mathbf{says} \ (P \to Q) \to (A \ \mathbf{says} \ P \to A \ \mathbf{says} \ Q)}\ {\to}R \times 2$$

Now we apply the rule of affirmation, which introduces two new antecedents derived from x and y, which we call x' and y'.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y' : P \vdash ? : P}\ \mathsf{id} \quad \dfrac{}{? : Q \vdash ? : Q}\ \mathsf{id}}{x' : P \to Q, y' : P \vdash ? : Q}\ {\to}L}{x' : P \to Q, y' : P \vdash ? : A \ \mathbf{aff} \ Q}\ \mathbf{aff}}{x : A \ \mathbf{says} \ (P \to Q), y : A \ \mathbf{says} \ P \vdash ? : A \ \mathbf{aff} \ Q}\ \mathbf{says}L \times 2}{x : A \ \mathbf{says} \ (P \to Q), y : A \ \mathbf{says} \ P \vdash ? : A \ \mathbf{says} \ Q}\ \mathbf{says}R}{\vdash ? : A \ \mathbf{says} \ (P \to Q) \to (A \ \mathbf{says} \ P \to A \ \mathbf{says} \ Q)}\ {\to}R \times 2$$

Now we can apply $\to L$ and then copy over x' and y' in applications of the identity

and then work our way down, annotating the succedent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y' : P \vdash y' : P}\,\mathsf{id} \quad \dfrac{}{x'\,y' : Q \vdash x'\,y' : Q}\,\mathsf{id}}{x' : P \to Q, y' : P \vdash x'\,y' : Q}\,{\to}L}{x' : P \to Q, y' : P \vdash x'\,y' : A \;\mathbf{aff}\; Q}\,\mathbf{aff}}{x : A \;\mathbf{says}\; (P \to Q), y : A \;\mathbf{says}\; P \vdash ? : A \;\mathbf{aff}\; Q}\,\mathbf{says}L \times 2}{x : A \;\mathbf{says}\; (P \to Q), y : A \;\mathbf{says}\; P \vdash ? : A \;\mathbf{says}\; Q}\,\mathbf{says}R}{\vdash ? : A \;\mathbf{says}\; (P \to Q) \to (A \;\mathbf{says}\; P \to A \;\mathbf{says}\; Q)}\,{\to}R \times 2$$

The $\mathbf{says}L$ rules wrap lets around the proof term for the affirmation judgment.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y' : P \vdash y' : P}\,\mathsf{id} \quad \dfrac{}{x'\,y' : Q \vdash x'\,y' : Q}\,\mathsf{id}}{x' : P \to Q, y' : P \vdash x'\,y' : Q}\,{\to}L}{x' : P \to Q, y' : P \vdash x'\,y' : A \;\mathbf{aff}\; Q}\,\mathbf{aff}}{x : A \;\mathbf{says}\; (P \to Q), y : A \;\mathbf{says}\; P \vdash \mathbf{let}\;\{x'\}_A = x\;\mathbf{in}\;\mathbf{let}\;\{y'\}_A = y\;\mathbf{in}\;x'\,y' : A \;\mathbf{aff}\; Q}\,\mathbf{says}L \times 2}{x : A \;\mathbf{says}\; (P \to Q), y : A \;\mathbf{says}\; P \vdash ? : A \;\mathbf{says}\; Q}\,\mathbf{says}R}{\vdash ? : A \;\mathbf{says}\; (P \to Q) \to (A \;\mathbf{says}\; P \to A \;\mathbf{says}\; Q)}\,{\to}R \times 2$$

It remains to wrap the proof to indicate A's perspective ($\mathbf{says}R$) and then introduce two λ-abstractions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y' : P \vdash y' : P}\,\mathsf{id} \quad \dfrac{}{x'\,y' : Q \vdash x'\,y' : Q}\,\mathsf{id}}{x' : P \to Q, y' : P \vdash x'\,y' : Q}\,{\to}L}{x' : P \to Q, y' : P \vdash x'\,y' : A \;\mathbf{aff}\; Q}\,\mathbf{aff}}{x : A \;\mathbf{says}\; (P \to Q), y : A \;\mathbf{says}\; P \vdash \mathbf{let}\;\{x'\}_A = x\;\mathbf{in}\;\mathbf{let}\;\{y'\}_A = y\;\mathbf{in}\;x'\,y' : A \;\mathbf{aff}\; Q}\,\mathbf{says}L \times 2}{x : A \;\mathbf{says}\; (P \to Q), y : A \;\mathbf{says}\; P \vdash \{\mathbf{let}\;\{x'\}_A = x\;\mathbf{in}\;\mathbf{let}\;\{y'\}_A = y\;\mathbf{in}\;x'\,y'\}_A : A \;\mathbf{says}\; Q}\,\mathbf{says}R}{\vdash \lambda x.\,\lambda y.\,\{\mathbf{let}\;\{x'\}_A = x\;\mathbf{in}\;\mathbf{let}\;\{y'\}_A = y\;\mathbf{in}\;x'\,y'\}_A : A \;\mathbf{says}\; (P \to Q) \to (A \;\mathbf{says}\; P \to A \;\mathbf{says}\; Q)}\,{\to}R \times 2$$

4 Example Revisited

Recall the motivating example, where we have labeled the antecedents with c_i. We imagine that in an implementation, they would be signed certificates.

$c_1 : admin$ **says** $(\forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R))$,
$c_2 : admin$ **says** $(\forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \wedge fp$ **says** $\mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R))$,

$c_3 : admin$ **says** $\mathsf{owns}(fp, ghc6017)$,
$c_4 : fp$ **says** $\mathsf{studentOf}(hemant, fp)$

$\vdash$

$?Q_0 : admin$ **says** $\mathsf{mayOpen}(hemant, ghc6017)$

We have also named the resulting (as yet to be determined) proof term so we can reference it. First, we apply **says**R and then unlock c_1, c_2, and c_3. We arrive at the sequent

$x_1 : \forall A.\, \forall R.\, \mathsf{owns}(A, R) \to \mathsf{mayOpen}(A, R)$,
$x_2 : \forall A.\, \forall B.\, \forall R.\, \mathsf{owns}(A, R) \wedge fp$ **says** $\mathsf{studentOf}(B, A) \to \mathsf{mayOpen}(B, R)$,

$x_3 : \mathsf{owns}(fp, ghc6017)$,
$c_4 : fp$ **says** $\mathsf{studentOf}(hemant, fp)$

$\vdash$

$?Q_1 : admin$ **aff** $\mathsf{mayOpen}(hemant, ghc6017)$

and

$$
\begin{aligned}
?Q_0 = \{\ & \textbf{let}\ \{x_1\}_{\mathsf{admin}} = c_1\ \textbf{in} \\
& \textbf{let}\ \{x_2\}_{\mathsf{admin}} = c_2\ \textbf{in} \\
& \textbf{let}\ \{x_3\}_{\mathsf{admin}} = c_3\ \textbf{in} \\
& ?Q_1\ \}_{\mathsf{admin}}
\end{aligned}
$$

Now x_2 together with the pair $\langle x_3, c_4 \rangle$ should complete the proof. But we also need to instantiate A, B, and R, with fp, $hemant$, and $ghc6017$, respectively, which is a form of function application.

$$?Q_1 = x_2\ fp\ hemant\ ghc6017\ \langle x_3, c_4 \rangle$$

Substituting this out in the original sequent, we get

$$
\begin{aligned}
?Q_0 = \{\ & \textbf{let}\ \{x_1\}_{\mathsf{admin}} = c_1\ \textbf{in} \\
& \textbf{let}\ \{x_2\}_{\mathsf{admin}} = c_2\ \textbf{in} \\
& \textbf{let}\ \{x_3\}_{\mathsf{admin}} = c_3\ \textbf{in} \\
& x_2\ fp\ hemant\ ghc6017\ \langle x_3, c_4 \rangle\ \}_{\mathsf{admin}}
\end{aligned}
$$

This proof term can now be communicated to and checked by the resource monitor.

References

Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

Lecture Notes on Bootstrapping Trust

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

Lecture 18
November 14, 2024

1 Introduction

When we discussed authorization logic, we briefly touched on the topic of trust. More specifically, authorization logic is distinguished from other logics that we have looked at by the A **says** P construct that represents the fact that principal A affirms proposition P. We saw how to encode *decentralized* security policies for systems using **says**, so that various entities can define their own access control rules. The system components that must ultimately either grant or deny access to resources can decide which principals to trust, and subsequently implement a policy by only accepting **says** proclamations from trusted principals.

We illustrated this using the Grey system Bauer et al. [2005], and now we will see a different example that emphasizes the importance of organized trust when making policy decisions. You may be familiar with the eduroam service, which provides members of participating academic institutions with wireless network access when they visit another institution. For example, because both CMU and Pitt are members of eduroam, when you visit the Pitt campus you can join the wireless SSID `eduroam`, and provide your Andrew ID and password to use the internet. The same is true at thousands of other institutions across the world that subscribe to this service.

If you stop to think about it, this is somewhat remarkable given the scale and disparity in geography and governance among the institutions. How does Pitt know that you have entered the valid credentials for your Andrew account, which is managed by CMU? A naive solution might be to distribute the credentials of *all* users at eduroam institutions to all of the institutions. Obviously this will not scale, so perhaps a decentralized authorization policy is called for. First of all, the service

*with edits by Giselle Reis, Ryan Riley, and Frank Pfenning

eduroam can delegate the responsibility of deciding who is currently a student to the various institutions, for example as shown below. We use er to denote the eduroam principal.

$$er \textbf{ says } \forall x.\, (cmu \textbf{ says } \mathsf{isStudent}(x)) \to \mathsf{isStudent}(x)$$
$$er \textbf{ says } \forall x.\, (pitt \textbf{ says } \mathsf{isStudent}(x)) \to \mathsf{isStudent}(x)$$

Then the main policy governing access to the eduroam wireless network allows every student to access the network.

$$er \textbf{ says } \forall x.\, \mathsf{isStudent}(x) \to \mathsf{canAccess}(x)$$

The wireless access points responsible for providing the service use this policy as assumptions Γ construct or check a proof of the sequent below when student $derek$ attempts to use the service.

$$\Gamma \vdash er \textbf{ says } \mathsf{canAccess}(derek)$$

For example, suppose that $derek$ attempts to do so. Rather than writing out the proof in the sequent calculus, we present it in the form of a proof term. In the proof-carrying authorization architecture, that is what would be communicated to the access point. More conventionally, it provides a logical justification for granting access but remains implicit in the code implementing access control for the network.

$$\begin{array}{lcl} c_1 & : & er \textbf{ says } \forall x.\, (cmu \textbf{ says } \mathsf{isStudent}(x)) \to \mathsf{isStudent}(x) \\ c_2 & : & er \textbf{ says } \forall x.\, \mathsf{isStudent}(x) \to \mathsf{canAccess}(x) \\ c_3 & : & cmu \textbf{ says } \mathsf{isStudent}(derek) \\ \vdash & & \\ ?M & : & er \textbf{ says } \mathsf{canAccess}(derek) \end{array}$$

$$\begin{array}{l} ?M = \{ \\ \quad \textbf{let } \{x_1\}_{er} = c_1 \textbf{ in} \\ \quad \textbf{let } \{x_2\}_{er} = c_2 \textbf{ in} \\ \quad \textbf{let } x_4 = x_1\ derek\ c_3 \textbf{ in} \\ \quad x_2\ derek\ x_4 \\ \}_{er} \end{array}$$

You may want to refresh your understanding of the details from the notes to Lecture 17.

Because the derivation above will be about the same for any user, except for the details of institution and user names, the endpoint need not recompute or check it for each login. Access boils down to a trusted institution's endorsement that the user is legitimate and eligible for the service, so instead perhaps users' devices can

just send evidence of the endorsement directly, or the endpoint can obtain it by some other means.

But how can the access point be sure that this evidence is authentic? We hinted in the last lecture that digital signatures utilizing cryptography are a common solution. But the cryptographic techniques that enable digital signatures require *keys*, which are either secrets distributed among trustworthy parties, or public objects that can be reliably associated with individuals or organizations.

For example, cmu could sign and date a certificate stating that $derek$ is currently a student, perhaps with a timeout to account for Bob's expected graduation date. It would do so using its *private key*, known only to CMU, and the access point would verify it by checking the signature against cmu's *public key*. How does the access point know that it is using the correct public key? What if someone tricked it into using a different public key, associated with an attacker's private key, so that it would trust statements signed by the attacker as coming from cmu?

We will address this topic today, looking more closely at digital certificates and *Public Key Infrastructure* (PKI), which is a distributed mechanism for managing the trust needed to solve the problems introduced in this example.

2 Digital Certificates & Certificate Authorities

For the rest of this lecture, we will assume that all principals A have a secret key sk/A and a public key pk/A. Suppose in the context of the running example from the previous section that the eduroam principal generates the public/private key pairs for all participating institutions.

Digital signatures. One of the main applications of public/secret key pairs is to *digital signatures*, which we have referenced informally before. A digital signature scheme consists of three algorithms, for generating keys, signing messages, and verifying signatures, respectively. We will always assume that public/secret key pairs have been generated correctly by some existing means, so we will not spend any time discussing the key generation algorithm. It is useful however to look a bit more closely at the latter two algorithms, $\mathsf{sign}_{sk/A}(m)$ and $\mathsf{verify}_{pk/A}(s, m)$, to understand how signatures are used to establish trust.

The signing algorithm $\mathsf{sign}_{sk/A}(m)$ takes as input a secret key sk/A for a principal A in addition to a message m, and outputs a signature s. The verification algorithm $\mathsf{verify}_{pk/A}(s, m)$ takes as input a public key pk/A for a principal A, a signature s, a message m, and outputs either true or false. It should be true *if and only if* the signature was produced by calling sign with A's secret key sk/A, i.e., $s = \mathsf{sign}_{sk/A}(m)$ for some m. Otherwise, $\mathsf{verify}_{pk/A}(s, m) = \mathsf{false}$. So in particular $\mathsf{verify}_{pk/A}(s, m)$ will return false if s is a signature created with the secret key of some other principal $B \neq A$, or more formally $\mathsf{verify}_{pk/A}(\mathsf{sign}_{sk/B}(m), m) = \mathsf{false}$ for all m. This is summarized below.

$$\mathsf{verify}_{pk/A}(s, m) = \begin{cases} \mathsf{true} & \text{if } s = \mathsf{sign}_{sk/A}(m) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Technically speaking, the behavior specified above is only required to hold with overwhelming probability over the keys produced by the generator [Katz and Lindell, 2014]. For our purposes however, it is fine to think of it as holding unconditionally.

The essential property established here is *unforgeability*. As long as sk/A remains a secret, and the only individual who knows the value of sk/A is A, then the only messages that $\mathsf{verify}_{pk/A}(\cdot)$ will return true on are those that A actually signed with sk/A. Of course, if one wanted to forge a message with A's signature, they could attempt to guess sk/A, which is why it is important that secret keys be chosen completely randomly from a very large space of possibilities. It is also important that the outputs of $\mathsf{sign}_{sk/A}(\cdot)$ reveal no information about sk/A that can help one guess the secret key with greater probability. We will assume that all of these facts hold for the secret keys and digital signatures used for the rest of the lecture, and we will also assume that if sk/A was generated by someone other than A (e.g., er in our running example), then they are trusted not to sign messages on A's behalf.

Certificates. Because er knows for a fact that CMU's public key pk/cmu is associated with the correct principal, it can generate a *certificate* that asserts this fact with its signature.

$$\mathsf{cert}_{er \to cmu}(pk/cmu) \equiv \mathsf{sign}_{sk/er}(\mathsf{isKey}(cmu, pk/cmu))$$

The predicate $\mathsf{isKey}(cmu, pk/cmu)$ denotes the fact that the public key pk/cmu belongs to, or is uniquely associated with, the principal cmu. er signs with its secret key sk/er to authenticate the certificate, as no other principals should have knowledge of sk/er and so only er itself could have produced the certificate.

Now if cmu wants to convince one of the access points that $derek$ is in fact a student, it can use $\mathsf{cert}_{er \to cmu}$ as part of a sequence of messages:

$$pk/cmu, \mathsf{cert}_{er \to cmu}(pk/cmu), \mathsf{sign}_{sk/cmu}(\mathsf{isStudent}(derek))$$

As long as the access point has eduroam's public key pk/er, then it will be able to verify that $\mathsf{cert}_{er \to cmu}(pk/cmu)$ is indeed signed by er, and so pk/cmu must really belong to cmu, and then use pk/cmu to verify that cmu signed $\mathsf{isStudent}(derek)$. cmu can send this information to the access point over an insecure channel, and the access point will still be able to trust the final conclusion.

Certificate authorities. Certificates enable the extension of trust to new principals from pre-existing trust relationships. In our running example, er is trusted by all access points to issue certificates for the public keys of other principals. In general, parties endowed with this sort of trust are called *certificate authorities* (CAs). The job of a CA is to issue digital certificates that associate principals with public keys, so in our example the CA is er.

The CA uses their own public/secret key pair to issue certificates, so those who wish to verify certificates issued by a particular CA need a reliable and secure way of obtaining the CA's public key. We will discuss several alternatives for achieving this in the next section, but for now it is fine to assume that all principals are in possession of the correct public key for the CA.

2.1 Formalizing certificates and trust

Now that we have seen how signatures and certificates are used to extend trust relationships, let us think about how to incorporate this into our reasoning about authorization. Specifically, we will formalize policies that utilize signatures, certificates, and trust in the CA so that these elements can be used with existing policies written in authorization logic. We will encapsulate this in a set of policies that can supplement the assumptions used in a proof, but one could alternatively incorporate these principles into axioms in the logic and devise corresponding proof rules [Bauer, 2003].

The first way in which we might want to use signatures is to introduce affirmations. Namely, if we are in possession of a proposition P signed with sk/A, and we know that sk/A is the secret key of A, then we can conclude that A **says** P. We will label this policy c_1 as formalized by the axiom c_1.

$$c_1 : \forall x.\, \forall pk.\, \mathsf{isKey}(x, pk) \to \mathsf{sign}_{sk/x}(P) \to x \textbf{ says } P$$

There is a small notational issue: previously, $\mathsf{sign}_{sk/x}(P)$ was defined as a function returning a signature (typically a string or a large number). Here we are using it as a proposition. This proposition should contain the signature, but also the proposition that was signed. So, more properly, it might be written as $\mathsf{signed}(x, s, P)$ where $s = \mathsf{sign}_{sk/x}(P)$. This is more difficult to read, so we will stick with the shorter $\mathsf{sign}_{sk/x}(P)$.

The assumption that makes c_1 reasonable is that if a principal is willing to sign something, then they are prepared to state it as well. In the base authorization logic, if our proof goal is an affirmation, then we had no choice but to apply **says**R and subsequently prove an affirmation judgement. It is not difficult to see that c_1 gives us an alternative way of proving such goals, namely that whenever we allow the axiom c_1 then we can prove A **says** P by proving $\mathsf{sign}_{sk/A}(P)$ and $\mathsf{isKey}(A, pk/A)$ from our assumptions.

There is a secondary issue here, namely that c_1 can not be a proper antecedent since it references an *arbitrary* proposition P. Quantifying over all propositions is

dicey (and in any case not allowed in our authorization logic). One way we could resolve that is to specialize the axioms to a certain class of propositions. Another is to turn it into an inference rule, the way we have with the axioms of dynamic logic. In that case we would obtain:

$$\frac{\Gamma \vdash \mathsf{isKey}(A, pk/A) \quad \Gamma \vdash \mathsf{sign}_{sk/A}(P)}{\Gamma \vdash A \textbf{ says } P} \textbf{ says/sign}$$

The second premise here uses the special sign predicate which we can prove by algorithmically verifying the signature.

$$\frac{\mathsf{verify}_{pk/A}(\mathsf{sign}_{sk/A}(P), P) = \mathsf{true}}{\Gamma \vdash \mathsf{sign}_{sk/A}(P)} \textbf{ verify}$$

This role can only be properly understood with the remark regarding our abuse of notation above: the formula $\mathsf{sign}_{\mathsf{sk/A}}(P)$ should actually contain the signature string s and either the public key pk/A or the principal A.

Now that we have a way of converting signatures into affirmations, we want to apply this towards formalizing the trust extension principle behind certificate authorities and the certificates that they issue. Recall that the trust placed in CAs is that they will sign messages that reliably tell us which principals are bound to particular public keys. So if we know that A is a certificate authority, and we have a certificate $\mathsf{cert}_{A \to B}$, then this principle lets us conclude that pk/B belongs to B. We formalize this as follows:

$$c_2 : \forall x. \forall y. \forall pk. \mathsf{isCA}(x) \to x \textbf{ says } \mathsf{isKey}(y, pk) \to \mathsf{isKey}(y, pk)$$

Just as the **says/sign** rule simplifies the use of c_1 in proofs, the **cert** rule does so for c_2.

$$\frac{\Gamma \vdash \mathsf{isCA}(A) \quad \Gamma \vdash A \textbf{ says } \mathsf{isKey}(B, pk)}{\Gamma \vdash \mathsf{isKey}(B, pk)} \textbf{ cert}$$

2.2 Example Revisited

Now that we understand how certificate authorities, digital signatures, and certificates work, let us return to the example from the beginning and sketch how to use these elements to prove an affirmation using digital signatures. We pick out as a subgoal the proof that cmu **says** $\mathsf{isStudent}(derek)$. We previously claimed that the message sequence

$$pk/cmu, \mathsf{cert}_{er \to cmu}(pk/cmu), \mathsf{sign}_{sk/cmu}(\mathsf{isStudent}(derek))$$

should convince the verifier that cmu **says** $\mathsf{isStudent}(derek)$. Logically, we express this message sequence, together with our policy and prior knowledge about the er as the following sequent:

$c_1 : \forall x.\, \forall pk.\, \mathsf{isKey}(x, pk) \to \mathsf{sign}_{sk/x}(P) \to x\ \mathbf{says}\ P$ (for all P)
$c_2 : \forall x.\, \forall y.\, \forall pk.\, \mathsf{isCA}(x) \to x\ \mathbf{says}\ \mathsf{isKey}(y, pk) \to \mathsf{isKey}(y, pk)$

$x_3 : \mathsf{sign}_{sk/er}(\mathsf{isKey}(cmu, pk/cmu))$
$x_4 : \mathsf{sign}_{sk/cmu}(\mathsf{isStudent}(derek))$
$x_5 : \mathsf{isCA}(er)$
$x_6 : \mathsf{isKey}(er, pk/er)$
$\vdash$
$?N : cmu\ \mathbf{says}\ \mathsf{isStudent}(derek)$

We can define $?N$ as:

$\mathbf{let}\ x_7 = c_1\ er\ x_6\ x_3 : er\ \mathbf{says}\ \mathsf{isKey}(cmu, pk/cmu)\ \mathbf{in}$ (for $P = \mathsf{isKey}(cmu, pk/cmu)$)
$\mathbf{let}\ x_8 = c_2\ er\ cmu\ pk/cmu\ x_5\ x_7 : \mathsf{isKey}(cmu, pk/cmu)\ \mathbf{in}$
$\mathbf{let}\ x_9 = c_1\ cmu\ x_8\ x_4 : cmu\ \mathbf{says}\ \mathsf{isStudent}(derek)\ \mathbf{in}$ (for $P = \mathsf{isStudent}(derek)$)
x_9

This completes the proof. Now the access point can conclude that cmu vouches for $derek$ and so according to the original eduroam policy that he is allowed to use the network. Notice how the access point is able to draw this conclusion by trusting only the CA, er in the beginning, and not cmu or $derek$. From this initial seed of trust it was able to "bootstrap" the additional trust assumptions that it needed to apply the authorization policy. This idea of bootstrapping trust from a few entities to many through CA designations is a key takeaway of this lecture, and one that is widely used in practice to enforce authorization on large-scale distributed systems.

2.3 Failure modes

Can principals always rely on certificates and trust relationships to establish authenticity of messages? There are a few situations that the access point needs to worry about, and they have to do with the assumption that *private keys are only known to their respective principals*. If this assumption ever fails, then problems can crop up in a few places in our running example.

The first case where the assumption can fail is for cmu. Supposing that cmu's private key used for signing messages (e.g., $\mathsf{sign}_{sk/cmu}(\mathsf{isStudent}(derek))$) becomes compromised and leaks to an untrusted individual who is not authorized to make statements on behalf of cmu. Then this person can sign messages of their choosing and have others who believe that pk/cmu belongs to cmu believe them with reasonable evidence. In the context of the example, that individual could sign things that are patently false, such as $\mathsf{isStudent}(beyonce)$, and the access point would believe that the messages originated from cmu. Recalling that the assertion $\mathbf{says}$ $cmu(\mathsf{isStudent}(x))$ is the only thing one needs to establish to access the network, this obviously renders the access control ineffective.

The other case corresponding to compromise of eduroam's secret key has similar consequences when considered in the context of our example. If an attacker is

in possession of eduroam's secret key, then they gain the ability to generate certificates that look like they came authentically from eduroam. So rather than using cmu's secret key directly, this attacker would generate a separate public/secret key pair $\langle pk/\star, sk/\star \rangle$, and use er's key to certify that $pk/\star$ belongs to cmu.

$$\mathsf{cert}_{er \to cmu}(pk/\star) \equiv \mathsf{sign}_{sk/er}(\mathsf{isKey}(cmu, pk/\star))$$

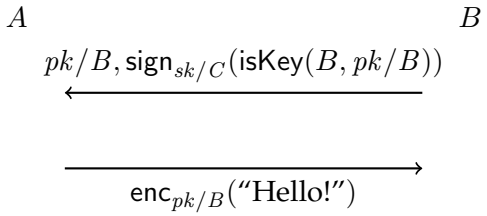
They could then convince the access point to allow any principal of their choosing to use the network, sending the messages such as

$$pk/\star, \mathsf{cert}_{er \to cmu}(pk/\star), \mathsf{sign}_{sk/\star}(\mathsf{isStudent}(beyonce))$$

As in the previous case, compromise of the secret key sk/er renders the access control system pointless.

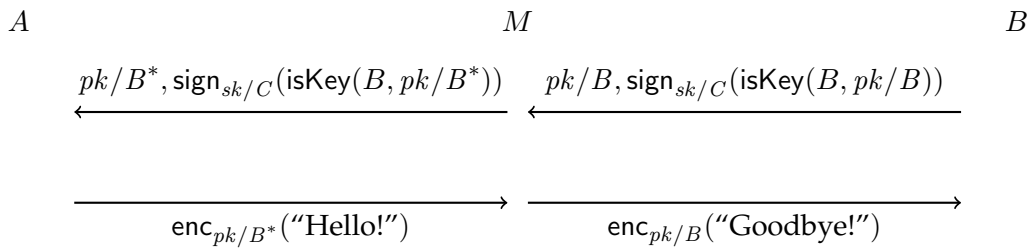
But outside the narrow context of our example, compromise of signing keys belonging to parties that are widely trusted to certify identities and establish policies is extremely serious. Without additional measures in place that we will discuss later, it gives one the ability to fabricate and steal the identities of arbitrary individuals. This can have dire consequences.

For example, suppose that A and B wish to communicate using their public and private keys. They trust certificates signed by C, and so if A wishes to send B an encrypted message, then B will first send A their public key pk/B and a certificate issued by C that attests to the validity of that public key. Then A can encrypt the message using B's public key, $\mathsf{enc}_{pk/B}(\ldots)$, and B will be able to decrypt with their secret key sk/B.



Now suppose that a malicious party M has obtained C's secret signing key. Then if M is able to intercept all messages passed between A and B, they can read the encrypted messages intended for B as well as make changes to them. When B sends A its public key and cert signed by C, then M uses C's signing key to certify a chosen public key $pk/B^\star$ (with corresponding secret key $sk/B^\star$ known to M), and forward $pk/B^\star$ to A with certification instead of pk/B. A will believe that $pk/B^\star$ is B's public key because it came with a certificated signed by trusted principal C,

and use it to encrypt messages to B as shown below.



Of course, because M knows the corresponding secret key $sk/B^{\star}$, it can decrypt and inspect the private messages A sent to B. It can then choose to either re-encrypt the original message with pk/B, or one of its choosing. This is called a *Man-in-the-Middle* (MitM) attack, as the attacker literally situates in between two parties who believe they are communicating over a secure channel.

3 Public key infrastructure

So far we have glossed over the details of how certificate authorities are assigned and managed. In the eduroam example, we assumed that access points know the correct pk/er because they are provisioned expressly for the service, and come preloaded with the necessary data. But certificates are used in all sorts of applications, and it may not always be possible to transmit the CA's key in such a way. How do principals come to trust a CA, and how does the CA know that pk/A actually belongs to A in cases where it does not generate the key? Answers to these questions entail defining a *Public Key Infrastructure* (PKI), and there are several alternatives for doing so.

3.1 Centralized CA

The most basic type of PKI consists of a single certificate authority who is trusted by all principals to issue certificates for everyone's public keys. Anyone who wants to use the PKI to establish trust in other principals must obtain a secure copy of the CA's public keys, and if they fail to accomplish this, then they will be unable to verify legitimate certificates issued by the true CA, and may instead end up "verifying" forged certificates issued by attackers. Protecting against this possibility is typically accomplished by obtaining a copy of the CA's key through physical contact, i.e., visiting the CA's offices and obtaining a file whose contents can be compared against a known checksum. Likewise, to obtain a certificate principals must usually present physical evidence of who they are, and that the keys they wish to have signed actually belong to them. Although the details of how this is done vary between CAs, the basic process must be transparent and rigorous enough so that others trust the CA's certs.

Another popular form of distribution for this model is to bundle public keys for widely-known CAs with popular software. This is done with browsers and operating systems, which typically implement a key store that is pre-loaded with CA keys that can be automatically verified as needed for validation. However, this approach is not without its risks, as users are often tricked into downloading corrupted versions of software that may have additional keys not associated with real CAs pre-loaded into the store. In this event, all of the failure modes discussed in the previous section are possible and likely, which is why it is important to always verify checksums for software that needs to interact with PKI.

This type of CA is usually a company that charges a fee to issue certificates, or department within an organization tasked with overseeing security. Because issuing certificates is a lucrative business model, in practice there are many "centralized" CAs that exist, and principals are free to choose whichever one they like when purchasing certificates for their keys. In one important sense this makes the overall PKI, in which users can choose which CAs to use and trust, less brittle to compromise of any one CA's signing key. In fact, it is considered good practice by some to obtain multiple certificates for the same key, so that if one CA is corrupted then others still have reason to trust the authenticity of the key. However, most browsers and operating systems that come pre-loaded with CA keys are configured to trust all of them equally, so really the entire PKI is only as trustworthy as the least-trustworthy CA. Ultimately, the responsibility is placed on end-users to configure their settings in response to corrupt or compromised CAs as such information becomes available. This is an unfortunate reality as most users are not equipped to make such decisions, and fixing it remains an open problem.

3.2 Delegated trust and hierarchical CAs

The reality of key compromise and the wide geographical reach of large certificate authorities has led to the emergence of an alternative hierarchical PKI. This model extends the centralized approach, and still makes use of the key distribution and principal verification strategies used by the centralized model. But now, the primary "root" CA *delegates* the ability to issue certificates to a number of subsidiary or "second-level " CAs. Certificates issued by second-level CAs then come with root-issued CAs themselves, thus forming a "certificate chain" that can be verified in sequence until reaching a trusted root CA with a known public key.

We can formalize this delegation policy using authorization logic. All that the CA needs to do is sign propositions that denote which principals they trust to sign on their behalf, e.g. with a predicate $\mathsf{trusts}(\ldots)$. Then the subsidiary CA can attach the policy signed by the root CA to any certificate that it issues using its delegation privilege.

$$\begin{aligned} CA\ \mathbf{says}\ (\forall x.\, \forall y.\, \forall pk.\, CA\ \mathbf{says}\ \mathsf{trusts}(x) \\ \rightarrow x\ \mathbf{says}\ \mathsf{isKey}(y, pk) \rightarrow \mathsf{isKey}(y, pk) \end{aligned}$$

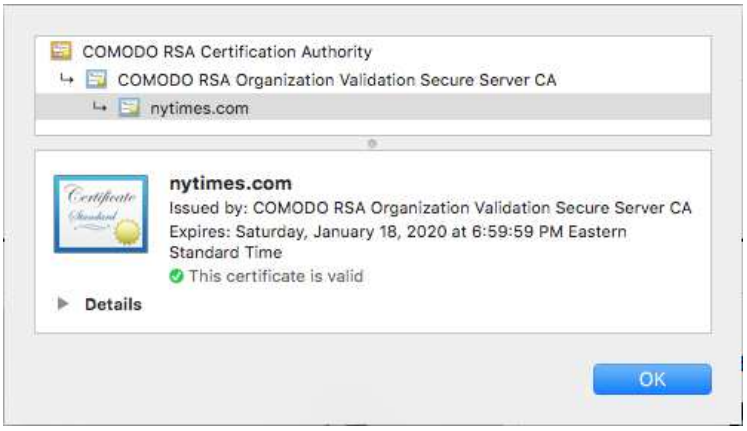


Figure 1: Example certificate chain used to authenticate the secure `nytimes.com` website, as displayed in the Chrome browser.

Checking that the rules described earlier allow others to make effective use of subsidiary-issued certificates is left as an exercise.

An example of this model in action is shown in Figure 1, which is the current certificate provided when visiting `https://nytimes.com`. The root CA in this case is *COMODO RSA Certification Authority* (we will call this A), and the second-level CA is *COMODO RSA Organization Validation Secure Server CA* (we will call this B), which is the principal who signed the public key of `nytimes.com`. The browser verifies that the certificate for `nytimes.com` was signed by B, and that the certificate for B was signed by the root CA A. In addition, the browser will check that the root-signed certificate for B's public key is authorized to sign certificates itself; this is a special "extension" field supported by the standard (X.509 [IETF, a]) for digital certificates. This special designation essentially says that B is trusted by A to issue additional certificates on behalf of A, and that those certificates should be treated as though they were issued by A itself.

The ability to delegate certification authority addresses many of the practical hurdles in the centralized CA model. Certificate authorities need to shoulder several burdens: ensuring the secrecy of the signing key, ensuring that the public key is readily available for verification, and vetting clients who wish to obtain certificates. Splitting these responsibilities among several subsidiaries makes good logistical sense. However, it also means that instead of just one signing key, there are now several that must be kept secret. The root CA must also ensure the integrity of subsidiary CAs, as they have the ability to issue certificates on behalf of the root CA, and so the trustworthiness of all related CAs is defined by the least trustworthy subsidiary. In short, while the hierarchical model solves some problems, it introduces several others.

Given that this model is in use on the Internet, exactly how many different

organizations can function as CAs and sign keys that your web browser will trust? It is difficult to be sure, because the distributed nature of delegation means that there is no central repository listing all CAs and who has delegated to whom. A study done by the Electronic Frontier Foundation's SSL Observatory in 2010 found 651 distinct organizations functioning as CAs. That number is likely even higher today.

3.3 Web of trust

An alternative PKI model to the hierarchical trust model is known as *Web of Trust* (WOT). While the hierarchical model is widely deployed in operating systems and web browsers, WOT has been in use for several decades particularly in the context of the *Pretty Good Privacy* (PGP) [Garfinkel, 1995] project. In WOT, trust is completely decentralized and users are responsible for making their own decisions about which certificates to trust. Likewise, every user is able to issue certificates as they wish, and distribute them at-will to others.

To get an idea of how this might work, consider a scenario where $giselle$ wishes to send $matt$ an email encrypted with her secret key. She sends her public key, along with certificates $\mathsf{cert}_{mike \to giselle}(pk_1)$ signed by the CMUQ dean Michael Trick and $\mathsf{cert}_{ryan \to giselle}(pk_2)$ signed by $ryan$. Suppose that $matt$ does not know $mike$ (because $matt$ has never visited CMUQ). Suppose that he does know $ryan$, because they have corresponded previously and so $matt$ has already established the authenticity of $ryan$'s public key. He can thus verify the first cert $\mathsf{cert}_{ryan \to giselle}(pk_1)$, and authenticate $giselle$'s public key prior to decrypting.

The main advantage of WOT over the prior two models is the distribution of potential failure. There are no concentrated points of failure in the event of compromise, and everyone is incentivized to proactively authenticate the public keys of anyone they communicate with. Over time, this tends to build redundancy into the system so that if any one user's signing key becomes compromised then anyone who may need to use a cert issued by them will still have several options available.

The main drawback is scalability and usability. While WOT remains in use in the context of encrypted email, it has not become an established alternative for other applications as it is difficult and time-consuming to develop a robust network of trust relationships. Additionally, users who are not familiar with public key cryptography face hurdles in being tasked with maintaining a secure and extensive set of trust relationships and certificates, and it is not at all clear that this approach is usable outside of the relatively homogeneous group of PGP devotees.

3.4 Dealing with certificate compromise

So far we have discussed the possible consequences of key compromise, and weighed the potential ramifications of several PKI models on this outcome. But what happens when a signing key becomes compromised? This poses a significant chal-

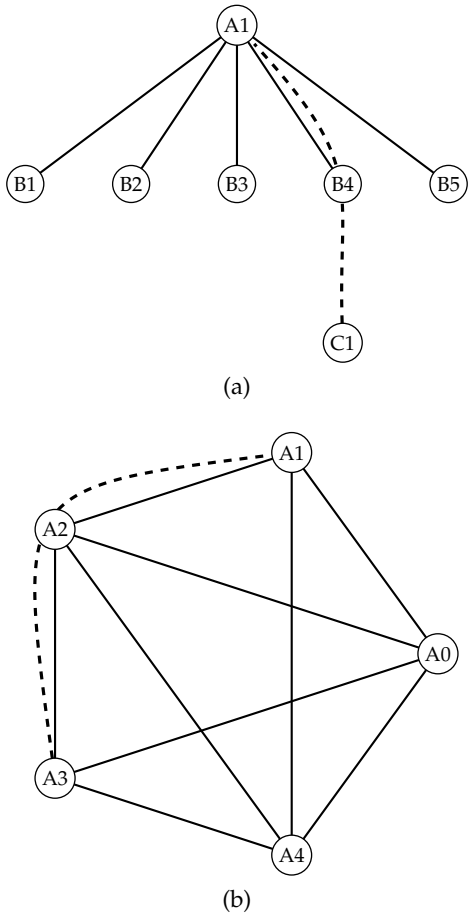


Figure 2: Hierarchical (a) and Web of Trust (b) PKI models, where solid lines correspond to existing trust relationships and dashed curves to the certificate chains that must be verified to build new ones. In the hierarchical model, A1 is the root CA, and B1-B5 are the second-level subsidiary CAs. The second-level CAs issue most certificates, so if one wants to verify C1's certificate then they need to first check that B4 signed C1's key, and then verify that A1 signed B4's key. In Web of Trust, all parties occupy a flat hierarchy, and verify certificates using previously-verified keys. If A3 wishes to authenticate A1's key, A3 can ask for a certificate signed by A3, who is a common point of trust between the two parties.

lenge, as continued trust in signatures issued with that key cripples the security of applications that rely on it. The CA needs to disavow, or *revoke*, the compromised key immediately while ensuring that users are aware that they should no longer trust the old one. There are several approaches for doing this, none of them entirely satisfactory. At present, this remains an open and active research question.

Expiration. Nearly all certificates in use today were issued with an expiration date, as shown in Figure 1. This facilitates a "default" mode of protection against key compromise, as once the expiration date passes verifiers will no longer trust the certificate. However, expiration alone is not sufficient to fully address the problem, as there is an untenable conflict between scalability and the burden on CAs to continually issue and distribute new certificates, and the "window of vulnerability" between compromise and the certificate's expiration date. In other words, it is not considered feasible to set short certificate lifetimes of, say, one day to one week, because if this were common practice then there would be no way for CAs to keep up with the logistical requirements needed to constantly re-issue new certificates. Typically, CA-issued certificates have lifetimes that last several years, and this leaves the parties in question with a potentially large time span in which their operations are affected by compromise. One notable exception to this is Let's Encrypt, which has automated the entire process of issuing certificates and so is logistically able to support three month lifetimes for certificates.

Certificate revocation lists. The most common way of handling this problem is for the CA to maintain a *certificate revocation list* (CRL). Each certificate is given a unique serial number, and if the key becomes compromised then the CA is notified that the certificate with the corresponding serial number should be revoked. The CA distributes an updated CRL each day, and verifiers are responsible for cross-referencing the list when checking a certificate.

Because new CRLs must be obtained by users regularly, this solution imposes a significant burden on the PKI. Whereas before communications could take place "offline" without needing to communicate with services exclusive to PKI, this is no longer the case. If the CRL server goes offline, either incidentally or as the result of an explicit attack, then users can no longer verify certificates without running the risk of accepting one signed by a compromised key. Additionally, CRLs tend to grow quite large over time, and this leads to non-trivial bandwidth costs for ISPs and end-users, particularly those who operate mobile devices. While proposals for incremental CRLs exist ([IETF, a], Section 5.2.4), they are not widely implemented.

Online Certificate Status Protocol (OCSP). An alternative to certificate revocation lists is OCSP, which is an active protocol in which parties "pull" information about certificate status rather than having CAs "push" the information routinely [IETF, b]. The details of the protocol are not immediately relevant to our

discussion, but it does offer some interesting tradeoffs to CRLs. The problem that OCSP alleviates is the transmission of large CRLs to end users. Because typical OCSP data transfers occur in response to specific certificate transactions, the amount of data in them is significantly smaller and therefore easier to parse and manage on resource-constrained devices. However, where connectivity or latency are issues, OCSP may become more burdensome than if an up-to-date CRL were stored on the device.

OCSP has also raised concern from privacy advocates. The on-demand "pull" nature of the protocol essentially requires users to tell a central third party whose certificates they would like to validate. Because much of the web now supports HTTPS, which requires certificate validation, this means that visiting a secure website from a browser that uses OCSP (this includes Internet Explorer, Mozilla, Safari, and Opera, but not Chrome) results in the OCSP server learning that the user visited that website. To make matters worse, the OCSP proposed standard does not mandate encryption by default, so third parties sitting between the user and the OCSP server may also be able to snoop on these requests. For these reasons, the support for OCSP seems to be waning [Aas, 2024].

Certificate pinning. A fairly recent practice called *certificate pinning* addresses the possibility of CA key compromise. Because there are dozens of root CAs that browsers are configured to trust by default, if any one of these CAs becomes compromised then the attacker can issue certificates as that CA for *any* user or domain. So suppose that `cmu.edu` contracts with only one CA for certificate issuance, $trustedCA$. Now a new CA, $discountCA$, enters the marketplace and quickly becomes compromised thanks to their lax security standards. If browsers and operating systems are already configured to trust $discountCA$, then the party who compromised their key can now issue certificates for `cmu.edu`, even though `cmu.edu` never used the services of $discountCA$!

Certificate pinning addresses this by allowing parties to "pin" a set of trusted CAs, so that verifiers will only trust the public keys of pinned CAs chosen by `cmu.edu`; in this case, `cmu.edu` would only pin $trustedCA$. Certificate pinning is now common practice when configuring HTTPS websites, and is supported by all major browsers. One drawback to certificate pinning is that it can obviate legitimate network security tools that essentially use man-in-the-middle attacks to scan encrypted network traffic for malicious content. Another drawback is that if the CA becomes compromised, then nobody will be able to verify certificates pinned to that CA until the pin expiration date arrives.

References

Josh Aas. Intent to end OCSP service. https://letsencrypt.org/2024/07/23/replacing-ocsp-with-crls/, July 2024.

Lujo Bauer. *Access Control for the Web via Proof-Carrying Authorization*. PhD thesis, Princeton University, November 2003.

Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, pages 431–445, Singapore, September 2005. Springer Verlag LNCS 3650.

Simson L. Garfinkel. *PGP - Pretty Good Privacy: Encryption for Everyone*. O'Reilly, 2nd edition edition, 1995.

IETF. RFC 6960: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) profile. https://tools.ietf.org/html/rfc6960, a.

IETF. RFC 5280: Internet X.509 Public Key Infrastructure Online Certificate Status Protocol – OCSP. https://tools.ietf.org/html/rfc5280, b.

Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2nd edition, 2014.

Lecture Notes on
Functional and Higher-Order Information Flow

15-316: Software Foundations of Security & Privacy
Hemant Gouni

Lecture 20
November 21st, 2024

1 Introduction

So far, we have looked at information flow in a simplified imperative setting. We looked at how to handle constructs like assignments, loops, and memory access for termination-insensitive information flow, then sprinkled on additional restrictions to account for termination and timing sensitivity. However, the language we've been using so far lacks even functions! Undoubtedly, you wouldn't like to work in such a language. In this lecture, we will show that the foundations of information flow we've developed generalize well beyond our simple imperative setting to handle the vastly different case of higher-order functional languages like Standard ML, OCaml, Haskell, or Lean. Of course, modern languages like Swift, Scala, and Rust combine both imperative and functional elements. The techniques introduced in this lecture can be used to develop a relatively complete account of information flow for them.

2 Functional Programs are Expressions

Recall from Lecture 11 that we defined the security level of expressions and formulas (reproduced in Figure 1) by finding the highest variable among them. For instance, $+F$ defines the security level of $e_1 + e_2$ as simply the highest variable contained between both (represented by taking their least upper bound $\sqcup$). This machinery is quite different from that for programs, which did not have a security level at all: we had to check them against a policy consisting of assignments from variables to security levels. The essential difference between expressions and programs in TinyScript is that expressions evaluate to a value, but programs are evaluated for their side effects. Because expressions evaluate to a value, they (semantically, the value they return) can be assigned a security level.

$\boxed{\Sigma \vdash e : \ell}$

$$\frac{\Sigma(x) = \ell}{\Sigma \vdash x : \ell}\ \mathsf{var}E \qquad \frac{}{\Sigma \vdash c : \bot}\ \mathsf{const}E \qquad \frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash e_1 + e_2 : \ell_1 \sqcup \ell_2}\ +E$$

$\boxed{\Sigma \vdash P : \ell}$

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash e_1 \leq e_2 : \ell_1 \sqcup \ell_2}\ \leq F \qquad \frac{}{\Sigma \vdash \top : \bot}\ \top F \qquad \frac{\Sigma \vdash P : \ell_1 \quad \Sigma \vdash Q : \ell_2}{\Sigma \vdash P \wedge Q : \ell_1 \sqcup \ell_2}\ \wedge F$$

Figure 1: Information Flow for Expressions and Formulas

For instance, the expression $1 + 1$ evaluates to 2, but the program $x := 1 + 1$ merely produces a poststate $[x \mapsto 2]$ when evaluated. All functional programs are of the former variety: they do not modify variables and therefore produce a poststate, but exclusively compute and return values. For the same reason that the information flow rules for TinyScript expressions are much simpler than those for programs, we can significantly simplify information flow in the functional setting! Gone is the complexity of checking assignments and of carefully setting up constructs to account for them— in **if**, **while**, and (as you saw on Assignment 4) **try**/**catch**. To check information flow for functional programs, we simply have to extend our existing label propagation approach for expressions and formulas. In other words, because information flow is a matter of reasoning about the inputs and outputs of a computation, we need only worry about the data passed to expressions and the data they return— because these are the only possible inputs and outputs in a pure functional setting. Throughout this lecture, we will take advantage of this simplicity and demonstrate the extra power it affords us.

3 Parametric Polymorphism is Information Flow

We first take a detour into more familiar territory. Consider the function fst in Figure 2. Type variables would typically have ticks before them, but they are italicized here instead. fst takes two arguments, and returns the first. Its type signature, $a \to b \to a$, expresses exactly this fact! That is, the type of fst captures its information flows. The intuition is that if we view a and b as security levels, then the type tells us the label of the return value: it is the same as the label of the first argument.

The function both is similar, taking two arguments and returning a pair containing them. Again, the return type $a * b$ tells us exactly which elements of the pair are dependent on which argument to both. In other words, the label of the first element is a, and the label of the second is b.

Now turn to add, which again takes two arguments but now adds them to-

$$\textbf{val}\ \text{fst} : a \to b \to a$$
$$\textbf{let}\ \text{fst}\ x\ y = x$$

$$\textbf{val}\ \text{both} : a \to b \to a * b$$
$$\textbf{let}\ \text{both}\ x\ y = (x, y)$$

$$\textbf{val}\ \text{add} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$$
$$\textbf{let}\ \text{add}\ x\ y = x + y$$

$$\textbf{type}\ a\ b\ \mathsf{sum} = \mathsf{Left\ of}\ b \mid \mathsf{Right\ of}\ a$$
$$\textbf{val}\ \text{branch} : \mathsf{bool} \to a \to b \to a\ b\ \mathsf{sum}$$
$$\textbf{let}\ \text{branch}\ b\ x\ y = \textbf{if}\ b\ \textbf{then}\ \mathsf{Left}(x)\ \textbf{else}\ \mathsf{Right}(y)$$

Figure 2: A couple ML programs

gether. From an information flow perspective, we'd like to know that the return value is dependent on both arguments. However, the ML type system will not allow us to straightforwardly express this: as soon as we do interesting (if you consider adding integers interesting) computation with our data, we lose the ability to talk about information flow. branch also witnesses this fact: it uses a sum type to express that the output is dependent on its latter two inputs. However, we miss the indirect flow from the first input b— which is not polymorphic because we need to compute with/branch on it— to the return value. Doing information flow this way is convenient, but appears to be quite brittle... surely there is a better way? We'll work informally first, before introducing the typing rules and discussing soundness.

3.1 A Second Attempt: Tagging Types

The key is to recognize that information flow of the variety shown above piggybacks on ML's ability to express machinery that is generic over the structure of its inputs. In other words, the same mechanism— parametric polymorphism— is deployed for both writing reusable machinery and specifying information flow properties. That seems to be the core of our troubles. What if we separate these two? Instead of conflating polymorphic types, which are intended to describe the structure of some underlying data, and information flow labels, which describe the structure of the computation, we introduce a special new type for describing information flow constraints. For simplicity, we won't use type variables within the data portion of the type going forward— we'll just specialize everything to base types bool and int.

Figure 3 shows the new types for the terms from Figure 2. $M : [\ a\ ]\ \mathsf{int}$ can be read as "the expression M has type int with dependencies a." The types for fst

$$\textbf{val}\ \text{fst} : [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ ]\ \mathsf{int}$$
$$\textbf{let}\ \text{fst}\ x\ y = x$$

$$\textbf{val}\ \text{both} : [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ ]\ \mathsf{int} * [\ b\ ]\ \mathsf{int}$$
$$\textbf{let}\ \text{both}\ x\ y = (x, y)$$

$$\textbf{val}\ \text{add} : [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ b\ ]\ \mathsf{int}$$
$$\textbf{let}\ \text{add}\ x\ y = x + y$$

$$\textbf{val}\ \text{branch} : [\ c\ ]\ \mathsf{bool} \to [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ b\ c\ ]\ \mathsf{int}$$
$$\textbf{let}\ \text{branch}\ b\ x\ y = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y$$

Figure 3: Adding information flow labels

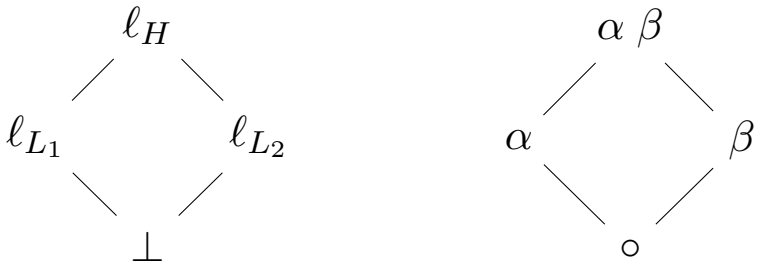
and both are nearly identical, now specialized to work on int and with the information flow labels appearing separately in brackets $[\ a\ ]$. add shows the first signs of departure, now able to be equipped with an information flow type signature expressing the dependency of its output on both of its inputs. Finally, the term for branch has changed: it no longer needs to use a sum type in order to capture its flows. Over the prior typing, the indirect flow from the conditional guard is now expressed with the dependency of the output on c.

3.2 Syntax and Typing

$$\begin{aligned}
\text{Dependencies}\ \phi &::= \circ \mid \phi\ \alpha \\
\text{Types}\ \tau &::= \mathsf{bool} \mid \mathsf{int} \mid [\tau \cdot \phi] \mid \tau_1 \to \tau_2 \mid \forall\alpha.\,\tau \\
\text{Expressions}\ M, N &::= \mathsf{true} \mid \mathsf{false} \mid n \mid x \mid M + N \mid \lambda x.\,M \mid M\ N \mid \Lambda\alpha.\,M \mid M\ [\phi] \\
&\quad \mid\ \textbf{if}\ N\ \textbf{then}\ M_1\ \textbf{else}\ M_2
\end{aligned}$$

With some intuition in hand, we can look at our information flow system more formally. A grammar is given above; we have security labels ϕ, types τ, expressions M, N, dependencies α, integers n, and variables x. The form of our typing judgment— for now— is $\Gamma \vdash M : \tau \mid \phi$. Our antecedents $\Gamma = x_1 : \tau_1, x_2 : \tau_2, \ldots$ consist of variables mentioned in M and their types. τ is the type of M, and ϕ is its set of dependencies. The latter corresponds to the bracketed dependency sets from Figure 3 and plays the same role as the labels ℓ from Figure 1.

Our dependency sets correspond to mathematical sets, and operations on them can be thought of that way: $\sqsubset$ is $\subset$, $\sqcup$ is $\cup$, and $\circ$ is $\varnothing$. This also means that the order and number of dependencies within a dependency set does not matter. As a matter of notation, we will elide $\circ$ when dependency sets are non-empty.



The above diagrams illustrate the difference between labels ℓ as we have previously worked with them, on the left, and labels ϕ in our system, on the right. We previously worked with abstract labels ℓ drawn from a lattice, with a partial ordering $\bot \sqsubset \ell_{L_1}, \ell_{L_2} \sqsubset \ell_H$ between them. The situation here is similar, but now the internal structure of the labels is exposed via the set of *dependencies* they represent. The partial ordering is expressed by taking subsets of those dependencies, as shown in the diagram on the right. The empty set of dependencies $\circ$ corresponds to the $\bot$ label. Each dependency represents some particular input to the computation; for instance, *password* or *gradebook* might appear inside labels. Dependencies won't always be so concrete; functions will generally introduce generic dependencies corresponding to their arguments, as seen in Figure 3.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \mid \circ}\ \text{T-TRUE} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \mid \circ}\ \text{T-FALSE} \qquad \frac{}{\Gamma \vdash n : \mathsf{int} \mid \circ}\ \text{T-INT}$$

$$\frac{\Gamma \vdash M : \mathsf{int} \mid \phi_1 \quad \Gamma \vdash N : \mathsf{int} \mid \phi_2}{\Gamma \vdash M + N : \mathsf{int} \mid \phi_1 \sqcup \phi_2}\ \text{T-ADD}$$

Starting with integers and booleans, we have four rules. T-TRUE, T-FALSE, and T-INT judge any boolean false/true or integer literal $n \in \mathbb{N}$ to be a bool or int with no dependencies. T-ADD computes the label of the addition of two integers to be the join (or set union) of their labels. We see that these are strikingly similar to the rules const$E$ and $+E$ presented in Figure 1— in fact, they are essentially identical! This is no mistake, and the intuition for the earlier rules carries over straightforwardly. For the program $x + y$, where x has dependencies α and y has dependencies β, the addition expression will have dependencies $\alpha\ \beta$.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \mid \circ}\ \text{T-VAR}$$

The variable rule is also nearly identical to the prior variant var$E$, requiring an $x : \tau$ in the antecedents Γ in order to conclude that x has type τ with no dependencies (represented by $\circ$). The last part is slightly odd, though: variables appear to be constrained to be at the $\circ$ (or $\bot$) label! This is not the case for const$E$, which permits

variables to be at whichever label is prescribed by the environment Σ (which we previously referred to as the security policy). What gives? The secret is in the next two rules, which permit security labels ϕ to be captured in types τ.

$$\frac{\Gamma \vdash M : \tau \mid \phi}{\Gamma \vdash M : [\tau \cdot \phi] \mid \circ}\ \text{T-CONSUME} \qquad \frac{\Gamma \vdash M : [\tau \cdot \phi_1] \mid \phi_2}{\Gamma \vdash M : \tau \mid \phi_1 \sqcup \phi_2}\ \text{T-PRODUCE}$$

Reading from top-to-bottom, T-CONSUME permits an expression M with some dependencies ϕ to pull (or 'consume') those dependencies into its type, turning its type τ into $[\tau \cdot \phi]$. T-PRODUCE reverses this operation, ejecting dependencies from the type of M back into the typing judgment. Now we see why T-VAR isn't very restrictive at all: it's perfectly valid to have $x : [\mathsf{int} \cdot \phi]$, which can be applied to T-PRODUCE to get $\Gamma \vdash x : \mathsf{int} \mid \phi$.

Let's work through a derivation of the program $x + y$ from before with the rules we've introduced so far. We complete the branch for x; the branch for y is analogous.

$$\dfrac{\dfrac{\dfrac{}{x : [\mathsf{int} \cdot \alpha], y : [\mathsf{int} \cdot \beta] \vdash x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-VAR}}{x : [\mathsf{int} \cdot \alpha], y : [\mathsf{int} \cdot \beta] \vdash x : \mathsf{int} \mid \alpha}\ \text{T-PRODUCE} \qquad \ldots\ y : \mathsf{int} \mid \beta\ \ldots}{x : [\mathsf{int} \cdot \alpha], y : [\mathsf{int} \cdot \beta] \vdash x + y : \mathsf{int} \mid \alpha\ \beta}\ \text{T-ADD}$$

It appears the rules are tracking information flow faithfully, as expected: the labels of both inputs to the addition are forced to be represented in the dependency set of the output. In our setting, a bad flow is one where the dependencies of the source of some flow (here, the inputs to addition) are not expressed in the type or dependency set of the destination (the output of addition).

$$\frac{\Gamma \vdash N : \mathsf{bool} \mid \phi_b \quad \Gamma \vdash M_1 : \tau \mid \phi \quad \Gamma \vdash M_2 : \tau \mid \phi}{\Gamma \vdash \textbf{if}\ N\ \textbf{then}\ M_1\ \textbf{else}\ M_2 : \tau \mid \phi_b \sqcup \phi}\ \text{T-IF}$$

T-IF is simpler than in the imperative setting. Since we're working in a functional language, **if** now returns the value of its succeeding branch rather than executing it for its side effects. This is reflected in the type τ of a conditional expression being the same as the type of its branches. T-IF requires that the security level of the whole expression $\phi_b \sqcup \phi$ depends on the security level of the conditional guard ϕ_b, which accounts for indirect flows. Previously, we set pc to the security level ϕ_b of the branch, but our language lacks assignment, memory access, or any other kind of side effecting operation, so we're absolved of that requirement.

A choice that may seem slightly odd here is that both of the branches are constrained to return the same dependency set ϕ. This seems unnecessarily prohibitive! Let's try relaxing this restriction.

$$\frac{\Gamma \vdash N : \mathsf{bool} \mid \phi_b \quad \Gamma \vdash M_1 : \tau \mid \phi_1 \quad \Gamma \vdash M_2 : \tau \mid \phi_2}{\Gamma \vdash \mathbf{if}\ N\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \tau \mid \phi_b \sqcup \phi_1 \sqcup \phi_2}\ \text{T-IF?}$$

The first rule turns out to be just as expressive as this one. To see why, consider the following program in ML:

$$\begin{array}{l}\mathbf{val}\ \mathrm{branch}' : [\ b\ ]\ \mathsf{bool} \to [\ a\ ]\ \mathsf{int} \to [\ a\ b\ ]\ \mathsf{int} \\ \mathbf{let}\ \mathrm{branch}'\ b\ x = \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0\end{array}$$

We might intuitively expect this would fail under the first rule because we have $x : \mathsf{int} \mid \alpha$ in the first branch and $0 : \mathsf{int} \mid \circ$ in the second. It turns out this is typable under T-IF because of another rule we haven't yet accounted for: *weakening*.

$$\frac{\Gamma \vdash M : \tau \mid \phi}{\Gamma \vdash M : \tau \mid \phi\ \alpha}\ \text{T-WEAKEN}$$

The rule of weakening allows us to add an arbitrary dependency to any expression. This may seem strange, but from an information flow perspective, it is intuitively sound: we may not lie *downwards* about our expression being of lower security than it actually is, but we may lie *upwards* and say that it is of higher security than it strictly needs to be. Think about it this way: it is fine to mark the boolean true with dependency *password*, because all this means is that we must now treat that boolean as though it contains password information— no password data can be leaked from this maneuver. Concretely, this means we can derive $\cdot \vdash 0 : \mathsf{int} \mid [\ \beta\ ]$ like so:

$$\dfrac{\dfrac{}{\cdot \vdash 0 : \mathsf{int} \mid \circ}\ \text{T-INT}}{\cdot \vdash 0 : \mathsf{int} \mid \beta}\ \text{T-WEAKEN}$$

In other words, whenever we have differing ϕs across branches, we can join them together and add dependencies on either side until they are equivalent. With this in our pocket, let's attempt a derivation for the body of branch' above. Premises in a $\boxed{\text{box}}$ are those yet to be solved.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash b : [\mathsf{bool} \cdot b] \mid \circ}\ \text{T-VAR}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash b : \mathsf{bool} \mid b}\ \text{T-PRODUCE} \quad \boxed{x} \quad \boxed{0}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0 : \mathsf{int} \mid a\ b}\ \text{T-IF}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0 : [\mathsf{int} \cdot a\ b] \mid \circ}\ \text{T-CONSUME}$$

$$\dfrac{\dots\ b\ \dots \quad \dfrac{\dfrac{}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash x : [\mathsf{int} \cdot a] \mid \circ}\ \text{T-VAR}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash x : \mathsf{int} \mid a}\ \text{T-PRODUCE} \quad \boxed{0}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0 : \mathsf{int} \mid a\ b}\ \text{T-IF}$$
$$\dfrac{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0 : \mathsf{int} \mid a\ b}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0 : [\mathsf{int} \cdot a\ b] \mid \circ}\ \text{T-CONSUME}$$

$$\dfrac{\dfrac{\dots\ b\ \dots \quad \dots\ x\ \dots \quad \dfrac{\dfrac{}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash 0 : \mathsf{int} \mid \circ}\ \text{T-INT}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash 0 : \mathsf{int} \mid a}\ \text{T-WEAKEN}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0 : \mathsf{int} \mid a\ b}\ \text{T-IF}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ 0 : [\mathsf{int} \cdot a\ b] \mid \circ}\ \text{T-CONSUME}$$

The invocation of T-WEAKEN in the last case shows our strategy for unifying dependencies across branches– we add a dependency a to 0 to satisfy T-IF. Returning to familiar territory, the rule for creating a lambda is nearly identical to what we have already seen. In Lecture 17, we introduced the 'proof term' versions of the $\to L$ and $\to R$ rules as:

$$\dfrac{\Gamma, x : P \vdash N : Q}{\Gamma \vdash (\lambda x.\, N) : P \to Q}\ \to R \qquad \dfrac{\Gamma \vdash N : P \quad \Gamma, M\ N : Q \vdash O : \delta}{\Gamma, M : P \to Q \vdash O : \delta}\ \to L$$

$$\dfrac{\Gamma, x : \tau_1 \vdash M : \tau_2 \mid \circ}{\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2 \mid \circ}\ \text{T-LAM} \qquad \dfrac{\Gamma \vdash M : \tau_1 \to \tau \mid \phi \quad \Gamma \vdash N : \tau_1 \mid \circ}{\Gamma \vdash M\ N : \tau \mid \phi}\ \text{T-AP}$$

T-LAM corresponds the right rule, and is nearly identical. T-AP and the left rule differ slightly: the left rule returns the result of application through its second premise, whereas T-AP presents the application form in its conclusion. As a slight aside, this captures the essential difference between *sequent calculus* and *natural deduction*-style presentations of programming language theory. In general, it will be the case that rules of creation will be identical between natural deduction and sequent calculus presentations, but rules for usage will return their result in the antecedent of a premise. In any case, the distinction isn't relevant here beyond gaining an understanding of the application rule.

The notable part of T-LAM is that it requires the body of the lambda to have consumed its dependencies into its type. We may be tempted to write the rule instead as follows, with the ϕ propagating through the lambda:

$$\dfrac{\Gamma, x : \tau_1 \vdash M : \tau_2 \mid \phi}{\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2 \mid \phi}\ \text{T-LAM?}$$

This is possible, but semantically odd: from an information flow perspective, the dependencies of the function body are only expressed when it is *called*, not when it merely *appears* somewhere. Formally, this happens because a lambda is a negative type, and is therefore defined by how it is used— not by its passive structure. No information can be observed from a lambda without calling it, so we only track information flow on application. We also force function arguments to have consumed all their dependencies in the second premise of application. This forces programs to track information flow more precisely. Consider the function which takes an argument with a higher label than its result:

$$y : [\mathsf{int} \cdot \alpha] \vdash \lambda x.\, 1 : [\mathsf{int} \cdot \alpha] \to \mathsf{int} \mid \circ$$

When we apply this function, it'll have the empty set of dependencies, even though the argument did not:

$$y : [\mathsf{int} \cdot \alpha] \vdash (\lambda x.\, 1)\ y : \mathsf{int} \mid \circ$$

Beyond precision, there is a deeper semantic reason for the choice that function arguments must have no dependencies: it makes our system easy to extend to handle side effects and more advanced forms of information flow checking. We won't have time to talk about this more, though.

Finally, we have T-DEPLAM and T-DEPAP. The two rules below are similar to the left and right rules for universal quantifiers previously introduced. Just as T-LAM binds a variable, T-DEPLAM binds a *dependency*. We can then use T-DEPAP to instantiate that dependency to some *dependency set*, substituting it into the type under the quantifier $\forall$. This allows us to write functions which are polymorphic over the dependencies of their inputs, just as we can write functions in ML which are generic with respect to the structure of their arguments. Before we look at an example, a small omission must be revealed: beyond Γ for keeping track of term variables, we also need Δ in our typing judgment for tracking which dependency variables are currently in scope.

$$\text{T-DEPLAM}\ \frac{\Delta, \alpha; \Gamma \vdash M : \tau \mid \circ}{\Delta; \Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\, \tau \mid \circ} \qquad \text{T-DEPAP}\ \frac{\Delta; \Gamma \vdash M : \forall\alpha.\, \tau \mid \phi' \qquad \Delta \vdash \phi\ \mathsf{dep}}{\Delta; \Gamma \vdash M\ [\phi] : [\phi/\alpha]\tau \mid \phi'}$$

And we must update T-WEAKEN to scope check the weakened variable, because we want to preserve the property that all dependency sets are well-scoped. As a technical detail, we must also check in T-LAM that the function argument is well-scoped, because it's effectively pulling its argument type out of thin air. Briefly, if we assume inductively that M at type τ_2 is well-scoped, that tells us nothing about the scopedness of τ_1.

$$\frac{\Delta; \Gamma \vdash M : \tau \mid \phi \quad \Delta \vdash \alpha \ \mathsf{dep}}{\Delta; \Gamma \vdash M : \tau \mid \phi\ \alpha}\ \text{T-WEAKEN}$$

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash M : \tau_2 \mid \circ \quad \Delta \vdash \tau_1\ \mathsf{type}}{\Delta; \Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2 \mid \circ}\ \text{T-LAM}$$

Let's try to type the identity function in our system. First, what does this look like in ML? The following seems reasonable:

$$\begin{array}{l}\textbf{val}\ \mathrm{id} : [\ a\ ]\ \mathsf{int} \to [\ a\ ]\ \mathsf{int} \\ \textbf{let}\ \mathrm{id}\ x = \ x\end{array}$$

This corresponds to the following typing:

$$\cdot; \cdot \vdash \Lambda\alpha.\, \lambda x.\, x : [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \mid \circ$$

Which results in the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\cdot, \alpha; \cdot, x : [\mathsf{int} \cdot \alpha] \vdash x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-VAR} \quad \dfrac{\cdots}{\cdot, \alpha \vdash [\mathsf{int} \cdot \alpha]\ \mathsf{type}}}{\cdot, \alpha; \cdot \vdash \lambda x.\, x : [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-LAM}}{\cdot; \cdot \vdash \Lambda\alpha.\, \lambda x.\, x : \forall\alpha.\, [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-DEPLAM}$$

That was pretty painless! We omit the derivation of the scoping premise for T-LAM because it's straightforward: it simply checks that all dependency variables mentioned in $\tau_1 = [\mathsf{int} \cdot \alpha]$ are mentioned in $\Delta = \cdot, \alpha$. We can then instantiate α to some b, c, assuming those dependencies are in scope, by substituting away the former for the latter:

$$\dfrac{\dfrac{\dfrac{\cdots}{\cdot, b, c; \cdot \vdash \Lambda\alpha.\, \lambda x.\, x : \forall\alpha.\, [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \mid \circ} \quad \dfrac{\ldots}{\cdot, b, c \vdash b\ c\ \mathsf{dep}}}{\cdot, b, c; \cdot \vdash \Lambda\alpha.\, \lambda x.\, x\ [b\ c] : [\mathsf{int} \cdot b\ c] \to [\mathsf{int} \cdot b\ c] \mid \circ}\ \text{T-DEPAP}$$

We can easily handle higher-order functions, too! Consider the below ML program which executes some arbitrary function twice:

$$\begin{array}{l}\textbf{val}\ \mathrm{twice} : [\ a\ ]\ \mathsf{int} \to ([\ a\ ]\ \mathsf{int} \to [\ a\ ]\ \mathsf{int}) \to [\ a\ ]\ \mathsf{int} \\ \textbf{let}\ \mathrm{twice}\ x\ f = \ f\ (f\ x)\end{array}$$

We can witness its information flow security through the following derivation– no tricks here, just rote application of our existing rules. Let's start by doing the

derivation up to the first application form. We'll elide the scope checking premises for space reasons and because well-scopedness is straightforward here.

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{f} \quad \boxed{f\ x}}{\cdot, \alpha;\ \cdot, x : [\mathsf{int} \cdot \alpha], f : [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \vdash f\ f\ x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-AP}}{\cdot, \alpha;\ \cdot, x : [\mathsf{int} \cdot \alpha] \vdash \lambda f.\, f\ f\ x : ([\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha]) \to [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-LAM}}{\cdot, \alpha;\ \cdot \vdash \lambda x.\, \lambda f.\, f\ f\ x : [\mathsf{int} \cdot \alpha] \to ([\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha]) \to [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-LAM}}{\cdot;\ \cdot \vdash \Lambda\alpha.\, \lambda x.\, \lambda f.\, f\ f\ x : \forall \alpha.\, [\mathsf{int} \cdot \alpha] \to ([\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha]) \to [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-DEPLAM}$$

Then we type check f via T-VAR, eliding the typing environment because it is the same as the conclusion:

$$\dfrac{\dfrac{}{\cdot, \alpha;\ \ldots \vdash f : [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-VAR} \quad \boxed{f\ x}}{\cdot, \alpha;\ \cdot, x : [\mathsf{int} \cdot \alpha], f : [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \vdash f\ f\ x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-AP}$$

And finally we type check its argument, which contains another call to f, in much the same way:

$$\dfrac{\dfrac{\dfrac{}{\ldots \vdash f : [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-VAR} \quad \dfrac{}{\ldots \vdash x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-VAR}}{\cdot, \alpha;\ \ldots \vdash f\ x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-AP} \quad \ldots f \ldots}{\cdot, \alpha;\ \cdot, x : [\mathsf{int} \cdot \alpha], f : [\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha] \vdash f\ f\ x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \text{T-AP}$$

In summary, to deal with the higher-order function f, we simply introduce it as a standard variable into our typing environment and type check usages of it as usual. When we apply it, we use its type signature to determine the resulting information flows.

We don't have lists in our formal language, but we might wonder what a standard higher-order function like map looks like. Let's look at an example in ML:

$$\begin{array}{l}\textbf{val}\ \mathrm{map} : ([\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int}) \to [\ a\ ]\ \mathsf{int\ list} \to [\ b\ ]\ \mathsf{int\ list} \\ \textbf{let}\ \mathrm{map}\ f\ lst\ = \textbf{match}\ lst\ \textbf{with} \\ \qquad\qquad\qquad\qquad\quad \mid [\,] \to [\,] \\ \qquad\qquad\qquad\qquad\quad \mid hd :: tl \to f\ hd :: \mathrm{map}\ tl\end{array}$$

This bears striking similarity to the standard type for map, and it is tempting to stop here. However, it is not fully general: the length of the list betrays information dependencies! In reality, the type $[\ a\ ]\ \mathsf{int\ list}$ is hiding a second dependency environment, constrained to be empty– its true form is $[\ \ ]\ ([\ \mathsf{a}\ ]\ \mathsf{int})\ \mathsf{list}$. If we want to

allow our map function to work over lists which may have lists whose structure— not just contents— induce flows, then we need to introduce another flow variable:

$$\begin{array}{l}\textbf{val}\ \mathrm{map} : ([\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int}) \to [\ l\ ]\ ([\ a\ ]\ \mathsf{int})\ \mathsf{list} \to [\ l\ ]\ ([\ b\ ]\ \mathsf{int})\ \mathsf{list} \\ \textbf{let}\ \mathrm{map}\ f\ lst\ = \textbf{match}\ lst\ \textbf{with} \\ \qquad\qquad\qquad\quad \mid [\,] \to [\,] \\ \qquad\qquad\qquad\quad \mid hd :: tl \to f\ hd :: \mathrm{map}\ tl\end{array}$$

It is worth noting that f itself may have information flow dependencies, so we really could further add a dependency variable to the function type itself. However, due to the structure of the T-LAM rule this is rare enough that we consider the above signature to be general enough. Additionally, there exists a way to take any data at function type with dependencies of its own, and integrate it into the dependencies its return value.

3.3 Noninterference

Why is this information flow at all— or rather, what does it have to do with information flow as we've talked about it previously? All information flow systems are joined at the hip by noninterference. Recall the prior definition of noninterference, from the Lecture 11 notes.

> We define $\Sigma \models \alpha$ secure iff for all ω_1, ω_2, ν_1, ν_2, and ℓ
> $\Sigma \vdash \omega_1 \approx_\ell \omega_2$, eval $\omega_1\ \alpha = \nu_1$, and eval $\omega_2\ \alpha = \nu_2$ implies $\Sigma \vdash \nu_1 \approx_\ell \nu_2$.

We won't give a semantic definition of noninterference in our setting, because the soundness argument for this system is quite complicated due to the presence of quantification. However, boiling this definition down to its essence, it states that, holding all low data constant, evaluating the same program under two states which differ along high data should yield equivalent results. We can intuit a similar property in our setting. First, define the constant function:

$$\mathrm{const} \triangleq \lambda x.\, 1$$

A valid typing for this is $[\mathsf{int} \cdot \alpha] \to \mathsf{int}$, assuming that the argument is an integer dependent on some α (you might imagine this to be denoting a dependency on something sensitive, like a password). More generally, for our purposes the input type need only have more, or different, dependencies than the output.[1] The following should be true:

$$\mathrm{const}\ x \equiv \mathrm{const}\ y \text{ for all } x, y$$

[1] For technical reasons, each dependency in the type should be assumed to be quantified over exactly the shown type. The simplified view suffices for our informal approach here, though.

Where it may be the case that $x \not\equiv y$. In fact, it turns out that we can replace const here with *any* expression of type $[\mathsf{int} \cdot \alpha] \to \mathsf{int}$, and the above property should hold. That is, assume f is some such term. Then noninterference in our case guarantees that:

$$f\ x \equiv f\ y \text{ for all } x, y$$

Any function from *high* data to *low* data must only reveal the *low* data. Let's look at one more example, inspired from one we've seen in Lecture 12. (We haven't yet introduced an equality construct, but information flow-wise, it is analogous to T-ADD.)

$$\mathrm{check} \triangleq \lambda password.\ \lambda attempt.\ password = attempt$$

If we imagine that *password* has dependency p, then can the output of this function be something that isn't dependent on p? Let's rashly assume that the type of this function is:

$$[\mathsf{int} \cdot p] \to \mathsf{int} \to \mathsf{bool}$$

Of course, this seems wrong: there's an indirect flow from *password* to the return value! Can we use our intuition about non-interference to show that this typing is invalid? Remember that, parenthesizing, the above type is equivalent to the following, which fits our type schema from the constant function above.

$$[\mathsf{int} \cdot p] \to (\mathsf{int} \to \mathsf{bool})$$

Non-interference says any program of this type should satisfy the equation:

$$\mathrm{check}\ x \equiv \mathrm{check}\ y \text{ for all } x, y$$

Okay, so let's see if $\mathrm{check}\ 2\ 3 \equiv \mathrm{check}\ 3\ 3$ (with the same argument given for *attempt* on both sides, because it's 'low'). $\mathrm{check}\ 2\ 3$ returns false, but $\mathrm{check}\ 3\ 3$ returns true. So we've reduced the problem to showing $\mathsf{false} \equiv \mathsf{true}$, which is impossible! Noninterference tells us $[\mathsf{int} \cdot p] \to \mathsf{int} \to \mathsf{int}$ cannot possibly be a valid typing for check. Recall, however, that we originally introduced this example in the context of declassification— a password checker which is barred by the type system from returning a low-security boolean doesn't seem useful...

3.4 Bonus: Existential Quantification, or Declassification

...which is precisely why we introduced *declassification*! Remarkably, it turns out that the constructs we have introduced so far are all we need to implement a form of declassification in the functional setting, with the key being *higher-rank quantification*. Let's work backwards, starting from our types:

$$impl : (\forall p.\ [\mathsf{int} \cdot p] \to ([\mathsf{int} \cdot p] \to \mathsf{int} \to \mathsf{bool}) \to \mathsf{int}) \to \mathsf{int}$$

This is the type of a *declassifier* which offers certain *methods it controls* to a *client*. The methods here are the first two arguments of the outermost higher-order function, which are $[\mathsf{int} \cdot p]$ and $[\mathsf{int} \cdot p] \to \mathsf{int} \to \mathsf{bool}$. Note that the quantifier $\forall p$ is over this higher-order function's type, not the whole function type— this is what makes the quantification higher-rank. In order to demystify the situation, let's investigate this type from two perspectives, implementation and usage.

$$\mathit{impl} \triangleq \lambda \mathit{client}.\ \mathit{client}\ [\circ]\ 4\ (\lambda \mathit{password}.\ \lambda \mathit{attempt}.\ \mathit{password} = \mathit{attempt})$$

$$\mathit{client} \triangleq \mathit{impl}\ (\Lambda p.\ \lambda \mathit{password}.\ \lambda \mathit{check}.\ \mathbf{if}\ \mathit{check}\ \mathit{password}\ 4\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$$

Since *client* fully applies *impl*, it must be the case that its type is int, without any dependencies. How can this be? *client* obviously returns an int dependent on *password*, since it branches on the value of *password*. And *password* appears to have a depedency on p by its type $[\mathsf{int} \cdot p]$— but the eventual return type for the computation is int! It seems the prime offender is *check*, which takes in a $[\mathsf{int} \cdot p]$ and another int and returns just a bool. By our prior discussion, this would be fine if *check* was constant in its first argument, but it isn't! We can see that its output is dependent on its 'high-security' argument— comparing it and returning the result— despite returning a low-security value.

Our hat trick here leverages *higher-rank quantification*, particularly *existential quantification*, permitting one view of password data to the implementation of the type and another view to any clients. The key is the instantiation in *impl*: it sets the dependency p, which is bound inside *client*, to $\circ$. This allows it to arbitrarily manipulate p while implementing the *password* field and *check* method which will be provided to *client*. Meanwhile, *client* is oblivious to the fact that this trick has been pulled: the function it provides to *impl* is fully polymorphic in its dependency p (indicated by binding p with a Λ), so it must treat it as any other dependency.

The warning in footnote 1 from the prior section stems from the fact that we can do declassification by instantiating dependency variables to $\circ$. Non-interference must in reality operate on *fully quantified* functions, where the flows expressed in the type are not *for some* instantiation of dependency variables (possibly to $\circ$), but *for all* instantiations. Concretely, the first of the following evidently isn't true of the preceding *check* function (of type $[\mathsf{int} \cdot p] \to \mathsf{int} \to \mathsf{bool}$), but the second must be true of some check' of type $\forall p.\ [\mathsf{int} \cdot p] \to \mathsf{int} \to \mathsf{bool}$.

$$\mathit{check}\ x\ 1 \equiv^{?} \mathit{check}\ y\ 1 \text{ for all } x, y$$

$$\mathit{check}'\ [\phi]\ x\ 1 \equiv \mathit{check}'\ [\phi]\ y\ 1 \text{ for all } \phi, x, y$$

For more information about deploying existential quantification to elegantly address declassification, see Cruz and Tanter [2019]. The main benefit of such a system is that declassification is constrained: each *impl* may only declassify the existential variables *introduced into clients by it*. All others must behave as ordinary dependencies. This provides excellent local reasoning properties: we can be sure

that functionality intended to declassify password data, for instance, does not accidentally affect gradebook data.

4 Remarks

The system we have introduced here bears striking similarities to System F as introduced in Girard [1972] and Reynolds [1984]. System F provides the basis for parametric polymorphism as featured in many real-world programming languages, and enjoys a relational property called *parametricity* which turns out to be quite similar in flavor to noninterference. Another approach to information flow within functional languages can be found in Simonet [2003], which addresses information flow for (a subset of) OCaml in a more directly lattice-based manner.

The contents of this lecture is the subject of (my) ongoing research! In this note we've introduced the *fully structural* fragment. It turns out that we can remove T-WEAKEN from the system and retain a sensible notion of non-interference. We can also remove two further rules corresponding to contracting ($[\ \alpha\ \alpha\ ] = [\ \alpha\ ]$) and commuting ($[\ \alpha\ \beta\ ] = [\ \beta\ \alpha\ ]$) dependencies, which we assumed implicitly in this lecture, and these too have reasonable non-interference properties (in fact, the former case can be seen to correspond to timing-sensitive information flow). The soundness of the system relies on a powerful generalization of the typical non-interference theorem, called *substructural non-interference*.

References

Raimil Cruz and Éric Tanter. Existential types for relaxed noninterference. In Anthony Widjaja Lin, editor, *Programming Languages and Systems*, pages 73–92, Cham, 2019. Springer International Publishing. ISBN 978-3-030-34175-6.

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Université Paris VII, 1972.

John C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In R. E. A Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523, Amsterdam, 1984. Elsevier Science Publishers B. V. (North-Holland).

Vincent Simonet. Flow Caml in a Nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, 2003.

Lecture Notes on Differential Privacy

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

Lecture 22
December 6, 2024

1 Introduction

In Lecture 12 we looked at ways of relaxing noninterference so that we could reason about the information flow security of programs that leak some, but not "too much" information about their secrets. To address this problem in the special context of authorization checks, we discussed a type system due to Volpano and Smith [2000]. They introduced a **match** construct and a corresponding typing rule that we later generalized to an arbitrary declassification construct.

if declassify$(guess = pin)$
then $auth := 1$
else $auth := 0$

For this program, our security policy assigned pin high security level and $guess$ and $auth$ low security level.

We did not formally introduce the notion then, but we can define the *feasible set* as those initial states that lead to the observable outcome.

$$\Omega_\Sigma(\omega, \alpha) = \{\omega' \mid \Sigma \vdash \omega \approx_\mathsf{L} \omega' \text{ and } \Sigma \vdash \mathsf{eval}\ \omega\ \alpha \approx_\mathsf{L} \mathsf{eval}\ \omega'\ \alpha\}$$

The password checker above is acceptable because its feasible set is rather large, say, $2^{64} - 1$ if the outcome is $auth = 0$. Every time we run this program with a new guess we can reduce the size of the feasible set by 1 (even if this temporal evolution isn't part of the formal definition). By contrast, if we replaced equality by **declassify**$(guess \leq pin)$ then we can cut the size of the feasible set in half every

*Almost entirely based on notes by Matt Fredrikson including some edits by Giselle Reis and Ryan Riley

time we run the program. As you saw in Lab 2 this can easily be used for an attack on a server to discover the pin.

You can think of the size of feasible set as a quantitative measure of the uncertainty that a user (possibly an attacker) has regarding the high security part of the initial state, once they know the outcome.

In today's lecture, we will look more carefully at a different set of techniques for revealing some useful information about secret state while controlling the attacker's level of uncertainty about it. These techniques all use randomness to produce approximate results for computations, while providing some form of cover for the true secret. We will look at a property called *differential privacy* [Dwork, 2006] that formalizes the protections one might gain from this approach, and study some properties that make it useful for building computations that protect secret data. Differential privacy has been applied to a wide range of important computations to protect the privacy of source data [Dwork and Roth, 2014], from machine learning [Chaudhuri et al., 2011] to web browser data collection [Erlingsson et al., 2014]. We will not have time to cover these applications in any detail, but will instead focus on the core ideas behind the approach.

2 Quantifying Uncertainty

As discussed above, when the feasible set is large, it is an indication that the associated program did not reveal "too much" about its secret initial state. The reason for this is that when a large number of initial states remain consistent with an attacker's observations, then the attacker's uncertainty about which one was actually used is great. So perhaps we can reason about information flow security in terms of keeping the attacker's uncertainty about the secret high.

But what can we do if the program that we want to write is inherently "leaky" in that it results in small feasible sets? One way that we can make the attacker more uncertain about the secret initial state is to use randomness in our program. Consider for example a technique called *randomized response* [Warner, 1965], which is a privacy technique dating back to the 1960s with roots in the social sciences. Randomized response was motivated by survey collection, in situations where questions asked of respondents relate to sensitive issues. Randomized response gives these subjects *plausible deniability*, by providing a structured way of adding random "noise" to their answer.

In the following, assume that $\mathbf{flip}()$ is a random function that flips an unbiased coin. In other words,

$$\mathbf{flip}() = \begin{cases} 1 & \text{with probability } 1/2 \\ 0 & \text{with probability } 1/2 \end{cases}$$

Then suppose that F is a function that returns a value in $\{0, 1\}$, and that we wish to release $F(x)$ publicly while hiding the secret value x as much as possible. Then

the randomized response program RandResp, is as follows, where we assume that the variable out is publicly-observable and b is not (e.g., $\Gamma = x : \mathsf{H}, b : \mathsf{H}, out : \mathsf{L}$).

$$
\begin{array}{l}
b := \mathbf{flip}() \\
\mathbf{if}\ b = 1\ \mathbf{then} \\
\quad out := F(x) \\
\mathbf{else} \\
\quad out := \mathbf{flip}()
\end{array}
$$

In short, randomized response returns the true value of $F(x)$ with probability $1/2$, and a completely random answer with probability $1/2$. In terms of feasible sets, this appears to be an absolutely brilliant approach because now the attacker must be completely uncertain about the initial value of x. Why is this so? The adversary can only see out, and if $b = 0$ after being assigned, then out does not depend at all on x, so x could be anything as though the program satisfied non-interference.

But perhaps this does not seem quite right. Let us assume for a moment that $x \in \{0, 1\}$ and F is simply the identity function, and walk through the various possibilities. In the following, we will treat RandResp as though it were a function of x that returns the value in out after executing. If $x = 0$, then,

$$\Pr[\mathsf{RandResp}(0) = 0] = \Pr[b = 1] + \Pr[b = 0 \wedge \mathbf{flip}() = 0] = 1/2 + 1/4 = 3/4$$

We could use the exact same reasoning to conclude that $\Pr[\mathsf{RandResp}(1) = 1] = 3/4$. Likewise we could reason about the probability that randomized response outputs an incorrect answer,

$$\Pr[\mathsf{RandResp}(0) = 1] = 1 - \Pr[\mathsf{RandResp}(0) = 0] = \Pr[b = 0 \wedge \mathbf{flip}() = 1] = 1/4$$

So we see that RandResp outputs the *correct* value of $F(x)$ with fairly high probability of $3/4$, and an incorrect "random" value with probability $1/4$. In other words, most of the time the attacker is safe in assuming that RandResp outputs exactly the same value as $F(x)$, and so can go about inferring x by computing feasible sets as before.

This is not to say that randomized response does nothing to protect x, and indeed it may offer ample protection for many applications because the attacker still has more uncertainty than they would otherwise. But by reasoning about the probabilities of various outcomes and what the attacker is able to infer from them, we arrived at a much more nuanced view of the degree of security than was suggested by looking at the feasible set of RandResp alone.

2.1 Quantifying a Tradeoff

There are some arbitrary choices that have been made in this conception of randomized response, and they influence the degree of adversarial uncertainty of the

secret input x. In particular, we could generalize $\mathbf{flip}()$ by adding a parameter $0 \leq p \leq 1$ controlling the bias of the coin.

$$\mathbf{flip}(p) = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases}$$

We could use this in RandResp as follows, assuming p is chosen to be some constant in advance.

$b := \mathbf{flip}(p)$
if $b = 1$ **then**
  $out := F(x)$
else
  $out := \mathbf{flip}(p)$

Then updating the analysis we did before with this more general solution, we see that:

$$\begin{aligned} &\Pr[\mathsf{RandResp}(x) = F(x)] = \Pr[b = 1] + \Pr[b = 0 \wedge \mathbf{flip}(p) = F(x)] \\ &= p + (1-p)\Pr[F(x) = \mathbf{flip}(p)] \end{aligned}$$

When F is the identity function then we have,

$$\begin{aligned} \Pr[\mathsf{RandResp}(0) = 0] &= \Pr[b = 1] + \Pr[b = 0 \wedge \mathbf{flip}(p) = 0] = p + (1-p)^2 \\ \Pr[\mathsf{RandResp}(1) = 1] &= \Pr[b = 1] + \Pr[b = 0 \wedge \mathbf{flip}(p) = 1] = p + (1-p)p \end{aligned}$$

So if we set $p \geq 1/2$, then we would be sure to have a more accurate answer in the sense that RandResp returns $F(x)$ with greater likelihood. But this comes at a tradeoff in information flow security, as the attacker can also be more confident (less uncertain) about the feasible set. Likewise, smaller values of p lead to a less accurate solution, but increase the attacker's uncertainty and so afford greater security.

3 Differential Privacy

Now we will turn to a property that is useful in many cases for characterizing the adversarial uncertainty one obtains through the use of randomized computation. In this setting, we will assume that the program α makes use of memory operations, and wants to prevent too much information about the contents of any cell in from leaking through its output result. It is called *differential privacy*, and is an active area of study and application.

In the following, we will assume that all of the indices in memory m are secret and so typed H, and that all of the variables used by the program are typed L. So intuitively, think of the memory m as perhaps being an input where each cell holds the data of one individual that is to be used by α. The developer of α wishes to compute some useful aggregate fact about the individuals' data, and will store the

result in the variables of the final state $\mathsf{eval}\ (\omega, m)\ \alpha$. The goal is to make sure that the results do not reveal too much information about any single individual's data stored in m.

Definition 1 (ϵ-Differential Privacy) *Let $\epsilon \geq 0$. A program α satisfies ϵ-differential privacy if for all possible memory configurations m_1 and m_2 that differ in* exactly one index*, and all states ω and ν, the following inequality holds:*

$$\Pr[\mathsf{eval}\ (\omega, m_1)\ \alpha = \nu] \leq e^{\epsilon} \cdot \Pr[\mathsf{eval}\ (\omega, m_2)\ \alpha = \nu]$$

The probabilities in this expression are taken over the randomness of α's computation.

We write $m_1 \sim_1 m_2$ if m_1 and m_2 differ in one index. The fact that α is a program that does not explicitly "output" a single value is indeed irrelevant to the essence of this definition. It may be clearer for some to just think of α as a function f that takes a memory configuration m as input and returns a single discrete value rather than a state. This leads to the following equivalent definition.

Definition 2 (ϵ-Differential Privacy (functional form)) *Let $\epsilon \geq 0$. A function f satisfies ϵ-differential privacy if for all possible inputs m_1 and m_2 with $m_1 \sim_1 m_2$ and all return values s the following inequality holds:*

$$\Pr[f(m_1) = s] \leq e^{\epsilon} \cdot \Pr[f(m_2) = s]$$

The probabilities in this expression are taken over the randomness of f's outputs.

To keep notation as simple as possible, we will stick with the latter form of the definition for the remainder of the lecture.

First, notice that Definition 2 is a property of the function f, and *not* of the data being computed on or any particular output of f. In other words, when we speak of something as being differentially private, we are always referring to a process used to compute outputs from secret inputs. You may at times hear people refer to a piece of data as "differentially private", but do not get confused; when used correctly, this language means that the data was computed by a function that satisfies ϵ-differential privacy.

Second, the ϵ in Definition 2 is called the *privacy budget*, and controls the tradeoff between privacy and accuracy in much the same way that p did in our randomized response example before. We will get into some high-level intuitive interpretations of this definition in a little while, but first let us think about its various components and how they relate to f's behavior directly.

Privacy budget ϵ. We hinted earlier that the privacy budget has an influence on both the degree of privacy established by the function, as well as the degree of

approximation in the results. ϵ is our privacy budget, and it is a numeric real-valued quantity. To understand what it means, let us look at the behavior of an ϵ-differentially private f for extremal values of ϵ.

Suppose that we make $\epsilon = 0$. Then Definition 2 requires that for any $m_1 \sim_1 m_2$ and outputs s, the stated inequality holds. Notice that the definition is symmetric in the values that m_1 and m_2 take; there is nothing that distinguishes them from each other, so f must also satisfy:

$$\Pr[f(m_2) = s] \leq \Pr[f(m_1) = s]$$

Combining the two inequalities, it must be that $\Pr[f(m_1) = s] = \Pr[f(m_2) = s]$ for all $m_1 \sim_1 m_2$ What does this mean for the privacy of individuals in m_1 and m_2, and the utility of f?

- When it comes to privacy, we can conclude that $\epsilon = 0$ implies **no leakage** of information about the contents of *any* individual. Why does this hold for any index? Recall that Definition 2 needs to hold for *all* pairs $m_1 \sim_1 m_2$. So, if our actual input is m_1, then all inputs m_2 that we obtain by changing one index in m_1 must produce the same distribution of outputs in f.
- As for utility, you probably guessed that $\epsilon = 0$ is not great. In fact, because f's output distribution needs to remain the same for all adjacent inputs, we can observe that by transitivity f's output distribution needs to remain the same for *all* inputs. In other words, the results cannot contain any information about the input, which clearly means no utility is possible.

On the other hand, when ϵ tends to infinity, then the inequality is automatically satisfied because the right-hand side become arbitrarily large. Therefore, no constraint is imposed on the function and privacy drops off very quickly. This yields maximal utility, because we can have all available data without randomness.

3.1 Interpreting the Definition

Now that we have thought about the definition and some of its technical implications, let us think about what it means for privacy.

Inference and protection from harm. One view of privacy is that it is about protecting individuals from harm that may arise from the release of their data. By learning things about individuals, a party with corrupt intent might use that information to limit their opportunities (e.g., deny them a job or a loan), offer differentiated services (e.g., higher prices for customers from affluent areas), or otherwise discriminate against them in numerous ways that play against their advantage.

One question that we might ask is, why not strive for a definition that prevents such parties from learning *anything* new about an individual from a result

involving their data? If nothing new about the individual can be learned from the release, then no harm can follow. Researchers have contemplated this possibility before [Dwork, 2006], and not suprisingly it turns out that doing so is at fundamental odds with a simultaneous goal of extracting useful insights from personal information.

Differential privacy aims to protect individuals from such harm to the greatest extent possible. The key to this is the *relative* nature of the definition. Rather than trying to prevent users from learning *anything* about an individual, we can think of the definition as trying to prevent users from learning new things about an individual relative to what they *could have* learned had the individual not shared their data. This is where the idea of neighboring inputs comes from: a neighboring input is one in which a particular individual's data takes a different value, which we can view as being a input where everyone *except* that individual shared (i.e., some other individual took their place). Differential privacy requires that any output of f be approximately as likely in both cases: one where the individual shared their data, and one where they did not.

For example, suppose that you are given the opportunity to share your medical records with a researcher who will use them in a study intended to improve treatments. You may rightly be concerned that if the researcher publishes results based on your data, a data-savvy insurance provider might be able to infer something about your health status from these results in the future, and decide to raise your premiums or deny coverage. However, if the researcher applied differential privacy with an appropriately-chosen ϵ, then you might be reassured that no results that could come of the study would be that much more or less likely because of your decision to share. It follows that if an insurer were to base their decision on those differentially-private results, then they are similarly not much more or less likely to deny you coverage.

Plausible deniability. Another way of looking at the protection given by differential privacy is in terms of *plausible deniability*, or one's ability to make a believable claim that their data takes some value of their choosing, i.e., to "deny" a claim that their data took the value it did. Because Definition 2 requires that the likelihood of f responding with any value s is nearly identical regardless of what value the individual's data took, it would indeed be reasonable for the individual to claim that their data took another value; the probability of producing s would be about the same no matter what value they chose.

Indistinguishability and influence. Another way of viewing the definition, which brings us closer to the semantics of the computation done by f, is in terms of how much individuals' data can influence, or cause changes to, f's response. We have talked about influence before in the context of noninterference, which required that the H-typed parts of the initial state have no influence on the L-typed parts of the

final state:

For all ω_1, ω_2, $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ implies $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_{\mathsf{L}} \mathsf{eval}\ \omega_2\ \alpha$

We might rewrite Definition 2 more concisely as follows.

For all m_1, m_2, $m_1 \sim_1 m_2$ implies $\Pr[f(m_1) = s] \leq e^{\epsilon} \cdot \Pr[f(m_2) = s]$

Notice the similarities between these definitions:

- In both cases, the definitions quantify over all pairs of inputs (i.e., initial states) that are related in a way that reflects what we are trying to protect. For noninterference, the relation does this by only constraining the low-security variables, so that the final state is indistinguishable regardless of the initial high-security variables. For differential privacy, the neighbor relation works similarly by letting one individual's data take an arbitrary value, and fixing the rest of the input.
- The right-hand side of the implication in each case describes the sort of changes that inputs, and more precisely inputs described by the left-hand side, are allowed to cause. Noninterference rules out any changes to low-security variables, whereas differential privacy places limits on the probability of variation in the response.

Viewed this way, differential privacy is a property which states that the influence of individual indices on f's response should remain low, so that responses computed under neighboring inputs are "almost" indistinguishable. This is the essential property that allows for plausible deniability and protection from harm, and the core of differential privacy's strong guarantees.

Recall also that we were able to prove that programs satisfy noninterference, even to the point of designing type systems that simplify the task of writing noninterferent programs, and can be checked efficiently. Given the similarity between these definitions, it should not be too surprising that we can also prove program's adherence to differential privacy. This is part of the appeal of using the definition in practice: it provides a crisp mathematical formulation of what it means to be private, that can be proved on real computations.

3.2 Proving differential privacy: randomized response

Now let us go back to our example of randomized response. Does it satisfy differential privacy? Let us keep things simple and assume that F is the identity function that just returns the contents of $m[0]$, $p = 1/2$ and all variables and memory cells

hold values in the set $\{0, 1\}$. This corresponds to the following program:

$$
\begin{array}{l}
f(m) = \\
\quad b := \mathbf{flip}(p) \\
\quad \mathbf{if}\ b = 1\ \mathbf{then} \\
\quad\quad out := m[0] \\
\quad \mathbf{else} \\
\quad\quad out := \mathbf{flip}(p)
\end{array}
$$

It turns out that this does indeed satisfy ϵ-differential privacy. Normally, we would do our calculation and then find a tight ϵ, but let's anticipate it.

Theorem 3 *f satisfies $\ln(3)$-differential privacy when $p = 1/2$ and $m[0] \in \{0, 1\}$.*

Proof: Recall that we need to show that the following inequality holds over all pairs of neighboring inputs and all outputs s:

$$\Pr[f(m_1) = s] \leq e^{\epsilon} \cdot \Pr[f(m_2) = s]$$

Because this instantiation of randomized response only depends on the contents of a single memory cell, i.e. $m[0]$, There are two possible configurations of neighboring inputs: $m_1[0] = 1, m_2[0] = 0$ and $m_1[0] = 0, m_2[0] = 1$.

Case: $m_1[0] = 1, m_2[0] = 0$. The inequality has to be satisfied for all s, that is, for $s = 1$ and $s = 0$.

Subcase: $s = 1$. Then we calculate the left-hand side

$$\begin{aligned} \Pr[f(m_1) = 1] &= \Pr[b = 1] + \Pr[b = 0 \wedge \mathbf{flip}(p) = 1] \\ &= p + (1 - p)p = 3/4 \end{aligned}$$

and the right-hand side

$$\begin{aligned} \Pr[f(m_2) = 1] &= \Pr[b = 0 \wedge \mathbf{flip}(p) = 1] \\ &= (1 - p)p = 1/4 \end{aligned}$$

For the inequality to hold we have to have

$$\Pr[f(m_1) = 1] = 3/4 \leq e^{\epsilon} \cdot 1/4 = e^{\epsilon} \cdot \Pr[f(m_2) = 1]$$

We see $3 \leq e^{\epsilon}$ and therefore $\epsilon = \ln(3)$ is a tight bound.

Subcase: $s = 0$. Again, we calculate both sides:

$$\begin{aligned} \Pr[f(m_1) = 0] &= \Pr[b = 0 \wedge \mathbf{flip}(p) = 0] \\ &= (1 - p)^2 = 1/4 \end{aligned}$$

And for the other side:

$$\begin{aligned}\Pr[f(m_2)=0] &= \Pr[b=1]+\Pr[b=0\wedge \mathbf{flip}(p)=0]\\ &= p+(1-p)(1-p)=3/4\end{aligned}$$

Summarizing

$$\Pr[f(m_1)=0]=1/4\leq e^{\epsilon}\cdot 3/4=e^{\epsilon}\cdot\Pr[f(m_2)=0]$$

so $1/3\leq e^{\epsilon}$, which is satisfied for any $\epsilon\geq 0$.

Case: $m_1[0]=0, m_2[0]=1$. In the case where $p=1/2$ this turns out to be symmetric to the previous case.

Subcase: $s=1$.

$$\begin{aligned}\Pr[f(m_1)=1] &= \Pr[b=0\wedge \mathbf{flip}(p)=1]=(1-p)p=1/4\\ \Pr[f(m_2)=1] &= \Pr[b=1]+\Pr[b=0\wedge \mathbf{flip}(p)=1]\\ &= p+(1-p)p=3/4\end{aligned}$$

Subcase: $s=0$.

$$\begin{aligned}\Pr[f(m_1)=0] &= \Pr[b=1]+\Pr[b=0\wedge \mathbf{flip}(p)=0]\\ &= p+(1-p)p=3/4\\ \Pr[f(m_2)=1] &= \Pr[b=0\wedge \mathbf{flip}(p)=0]\\ &= (1-p)(1-p)=3/4\end{aligned}$$

So indeed the probabilities are simply inverted in this case

□

Now what would we expect for $p=1/4$? We go into the branch where we reveal $m[0]$ less frequently, instead returning a random (if biased) answer instead. This would indicate that the privacy budget could be less than $\ln(3)$. Conversely, if $p=3/4$ we go into the revealing branch with much higher probably, so we would need a higher privacy budget to be allowed to do that.

If we did our math right, there would be four inequalities that should be satisfied for general p:

$$\begin{aligned}\frac{p+(1-p)p}{(1-p)p} &\leq e^{\epsilon}\\ \frac{(1-p)(1-p)}{p+(1-p)(1-p)} &\leq e^{\epsilon}\\ \frac{(1-p)p}{p+(1-p)p} &\leq e^{\epsilon}\\ \frac{p+(1-p)p}{(1-p)(1-p)} &\leq e^{\epsilon}\end{aligned}$$

The middle two fractions are less than 1 and are therefore automatically satisfied.

Plugging in $p = 1/4$, we get $\epsilon \geq \ln(7/3)$, which is indeed less than $\ln(3)$. Plugging in $p = 3/4$, we get $\epsilon \geq \ln(15)$, which is indeed greater than $\ln(3)$. One could also imagine inverting the question: given a certain privacy budget, can we choose a suitable p to make the function comply?

4 Differentially Private Programming

In the preceding section we used an informal proof of adherence to ϵ-differential privacy even though the primary object of analysis in this theorem was a program. It is possible to prove this theorem more formally, but to do so we would need a formal semantics for the programming language with random elements (e.g., $\mathbf{flip}(p)$), and logic for expressing properties of this language like dynamic logic, and sound proof rules for that logic. Such things exist, and also remain an active area of research, but are beyond the scope of this class.

Alternatively, we could go the route of information flow and devise a *type system* so that type-checking a program would guarantee that it satisfies ϵ-differential privacy. Perhaps surprisingly, this is also possible! There is a sequence of two supremely elegant papers [Reed and Pierce, 2010, Gaboardi et al., 2013] that lay out such type systems, which also provide useful techniques for composing differentially private computations in various ways.

References

Kamalika Chaudhuri, Claire Monteleoni, and Anand D. Sarawate. Differentially private empirical risk minimization. *Journal of Machine Learning Research*, 12: 1069–1109, 2011.

Cynthia Dwork. Differential privacy. In *33rd International Colloquium on Automata, Languages, and Programming, II (ICALP 2006)*, pages 1–12, Venice, Italy, July 2006. Springer LNCS 4052.

Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, August 2014.

Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Conference on Computer and Communications Security (CCS 2014)*, pages 1054–1067, Scottsdale, Arizana, November 2014. ACM.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In Roberto Giacobazzi

and Radhia Cousot, editors, *40th Symposium on Principles of Programming Languages (POPL 2013)*, pages 357–370, Rome, Italy, January 2013. ACM.

Jason Reed and Benjamin C. Pierce. Distances makes the types grow stronger: A calculus for differential privacy. In P. Hudak and S. Weirich, editors, *15th International Conference on Functional Programming (ICFP 2010)*, pages 157–168, Baltimore, Maryland, September 2010. ACM.

Dennis M. Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In M. N. Wegman and T. W. Reps, editors, *Symposium on Principles of Programming Languages*, pages 268–276, Boston, Massachusetts, January 2000. ACM.

Stanley Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.