15-316: Software Foundations of Security and Privacy

Lecture Notes on Information Flow Types II

Matt Fredrikson

Carnegie Mellon University
Lecture 12

1 Introduction

In the previous lecture, we began our study of a type system that enforces the information flow property non-interference [VIS96]. Recall that the relation $\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2$ denotes the fact that states ω_1 and ω_2 agree on the values of variables typed $\mathtt{L}$ by type context Γ. Then Definition 1 says that a program α satisfies non-interference if and only if it does not allow the values of $\mathtt{H}$-typed variables in the initial state to influence $\mathtt{L}$-typed variables in the final state.

Definition 1 (Non-interference)**.** Let α be a program and Γ a type environment associating security labels to all of the variables in α. Then α satisfies non-interference under Γ if and only if executing α under $\mathtt{L}$-equivalent states leads to final states that are also $\mathtt{L}$-equivalent. More precisely,

$$\forall \omega_1, \omega_2 . \omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2 \land \langle \omega_1, \alpha \rangle \Downarrow \omega_1' \land \langle \omega_2, \alpha \rangle \Downarrow \omega_2' \to \omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2' \tag{1}$$

where ω_1 and ω_2 range over the set of possible program states.

We then began looking at a set of typing rules that would allow us to decide whether a given program satisfies this definition under a given typing context: if we can use the rules to prove that the program is well-typed according to this system, then the program satisfies the property. We developed the rules for expressions, which form judgements of the form $\Gamma \vdash e : \ell$, meaning that under context Γ expression e is typed ℓ.

$$\text{(ConstL)}\ \frac{}{\Gamma \vdash c : \mathtt{L}} \quad \text{(TrueL)}\ \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{L}} \quad \text{(FalseL)}\ \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{L}}$$

$$\text{(Var)}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \quad \text{(UnOp)}\ \frac{\Gamma \vdash e : \ell}{\Gamma \vdash \odot\, e : \ell} \quad \text{(BinOp)}\ \frac{\Gamma \vdash e : \ell_1 \quad \Gamma \vdash \tilde{e} : \ell_2}{\Gamma \vdash e \odot \tilde{e} : \ell_1 \sqcup \ell_2}$$

Recall that when we developed the rule BinOp, we introduced a partial order $\sqsubseteq$ on security types. The partial order is reflexive and transitive, so $\ell \sqsubseteq \ell$ and if $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_3$ then $\ell_1 \sqsubseteq \ell_3$. For the types $\texttt{L}, \texttt{H}$, we formalized our intuition about these labels corresponding to "low" and "high" by specifying the partial ordering $\texttt{L} \sqsubseteq \texttt{H}, \texttt{H} \not\sqsubseteq \texttt{L}$. Finally, we attached the operator $\ell_1 \sqcup \ell_2$ to our partial order on security types, which returns the smallest label that is as least as large as ℓ_1 and ℓ_2, where "smallest" and "largest" come from the order relation $\sqsubseteq$.

Today we will continue developing this type system, covering a set of rules that let us judge the type-correctness of commands. But first we will introduce a lemma about the rules for expressions called the *simple security lemma* which says that if an expression can be typed with label ℓ by the rules, then the expression does not depend on any information given a higher label. This lemma will be crucial later on when we prove that the full type system is sound, meaning that only programs satisfying non-interference have derivations of their typing judgement in this system.

2 Simple Security

So far we have introduced the typing rules for expressions in our imperative language, and for each one we tried to build an intuition for why the rule appropriately capture the intent of the type system to prevent information flows from H to L variables. But we haven't formalized this, and before pressing on with the typing rules for commands it will be helpful to understand what the system gives us from judgements on expressions.

Expressions in our language are only capable of reading sensitive information, and possibly carrying that information in their value. So we would like to formalize a property that says something like, "an expression never reads what it isn't supposed to read". What does "supposed to" mean? Recall that judgements on expressions take the form $\Gamma \vdash e : \ell$. So for example if we were able to use the rules to derive $\Gamma \vdash e : \texttt{L}$, then we would hope that e doesn't read any data typed H by Γ. The simple security lemma says exactly this, which is that expressions typed by the system never read data from sources with higher types according to $\sqsubseteq$.

Lemma 2 (Simple Security)**.** *If $\Gamma \vdash e : \ell$, then for every variable subexpression x appearing in e, $\Gamma(x) \sqsubseteq \ell$. In other words, expressions never read variables above their type.*

Proof. This proof is done most appropriately by induction on the structure of e. We will treat arithmetic and Boolean expressions at once, so the inductively-defined syntax that we are working with is as follows:

$$e, \tilde{e} \ ::= \ c \mid \texttt{true} \mid \texttt{false} \mid x \mid \ \odot\, e \mid e \odot \tilde{e}$$

As before, we use $\odot$ to represent unary and binary operators.

The base cases in this proof correspond to the expressions with no subexpressions, which are the numeric and Boolean constants $c, \texttt{true}, \texttt{false}$, as well as variable expressions x. We proceed with the inductive argument by cases.

Base case: constants. The rules ConstL,TrueL,FalseL type all expressions ℓ. The proof for these cases is indeed trivial, because constants do not contain any variable subexpressions.

Base case: variables. The rule Var gives us the judgement $\Gamma \vdash x : \Gamma(x)$. The only variable appearing in the expression is x, and $\sqsubseteq$ is reflexive so $\Gamma(x) \sqsubseteq \Gamma(x)$.

Inductive case: unary operators. The rule UnOp says that if $\Gamma \vdash e : \ell$, then $\Gamma \vdash \odot e : \ell$. The only variables appearing in $\odot e$ are those that also appear in e, and the inductive hypothesis tells us that for all variables x appearing in e, if $\Gamma \vdash x : \ell'$ then $\ell' \sqsubseteq \ell$.

Inductive case: binary operators. The rule BinOp says that if $\Gamma \vdash e : \ell_1$ and $\Gamma \vdash \tilde{e} : \ell_2$, then $\Gamma \vdash e \odot \tilde{e} : \ell_1 \sqcup \ell_2$. The only variables appearing in $e \odot \tilde{e}$ are those that also appear in either e or $\tilde{e}$, and the inductive hypothesis tells us that for all variables x appearing in e, if $\Gamma \vdash x : \ell_1'$ then $\ell_1' \sqsubseteq \ell_1$, and likewise for any variable appearing in $\tilde{e}$, $\Gamma \vdash x : \ell_2'$ then $\ell_2' \sqsubseteq \ell_2$. But $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$, and $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$, so because $\sqsubseteq$ is transitive it must be that for any variable x appearing in *either* e or $\tilde{e}$, if $\Gamma \vdash x : \ell'$ then $\ell' \sqsubseteq \ell_1 \sqcup \ell_2$. This completes the proof.

$\square$

The simple security lemma gives us a straightforward guarantee to work with as we develop typing rules for commands that contain expressions. Namely, we can assume that whatever label the type system gives us for an expression will be an upper-bound of all of the labels assigned to variables within the expression, and thus an upper bound on the security type of information that may influence an expression.

2.1 Type-checking commands

Now that we know how to assign types to expressions, we can move on to the more interesting question of how to check that a given command satisfies non-interference under a type context Γ. We will proceed as before, developing a set of rules that we can later prove give us this property. The judgements that the rules derive is slightly different than with expressions, though. Keeping in mind that the security types L, H denote the fact that a given data element, e.g. an expression in our language, carries information of a particular security level, it does not make sense to write something like the following:

$$\Gamma \vdash \texttt{if}(x \le 0)\, y := 1\, \texttt{else}\, y := 0 : \texttt{H}$$

We have not thought of programs as any sort of data element capable of carrying information by themselves, and however intriguing such an idea may seem, doing so would distract us from our goal of enforcing non-interference at the moment.

Instead, we have thought of programs as objects that compute, i.e., by applying operations to data and moving it between variables. Our goal is to make it impossible to use the typing rules to constructa proof that a program satisfies non-interference when it doesn't. So the judgements that our rules for commands will use takes the form shown in Equation 2, and should be read as "under type context Γ, program α is well-typed". We will set up the rules so that "well-typed" implies "satisfies non-interference", but we'll say more on this later.

$$\Gamma \vdash \alpha \tag{2}$$

Now we proceed to design typing rules for each of the commands in our language.

Assignments. To design a typing rule for assignment commands, we must ask ourselves what conditions might result in a violation of Definition 1. More intuitively, what conditions on Γ and the command $x := e$ would result in a flow of information from variables labeled `H` to those labeled `L`?

Certainly, if the target of the assignment x is labeled `L` and the expression on the right-hand side is labeled `H`, then such a flow will occur. More generally, if $\Gamma(x) = \ell_1$ and $\Gamma \vdash e : \ell_2$, where $\ell_2 \not\sqsubseteq \ell_2$, then an information flow will occur when the value of e is assigned to x. Perhaps we can write the following rule, shown in Equation 3.

$$\frac{\Gamma \vdash e : \ell_1 \quad \ell_1 \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e} \tag{3}$$

This rule says that if the label of e as no larger (i.e., "more secret") than that of x, then the assignment is well-typed. Does this work? What about the "implicit" information flows we discussed last lecture, such as that in the following judgement?

$$x : \mathtt{H}, y : \mathtt{L} \vdash \mathtt{if}(x)\, y := 1\, \mathtt{else}\, y := 0 \tag{4}$$

We know that this doesn't satisfy non-interference, but it seems as though we might not reject it using the rule shown in (3).

In order to reject programs with implicit flows, we will keep track of the security label of a distinguished "program counter" variable `pc` in Γ. Later when we design rules for conditionals and loops, we will make sure to account for this part of the type context so that whenever the control flow is currently influenced by `H`-labeled data, then $\Gamma(\mathtt{pc}) = \mathtt{H}$. But for now we will just assume that this has been properly accounted for in the context, and use it to define our rule for assignment.

$$(\text{Asgn}) \ \frac{\Gamma \vdash e : \ell_1 \quad \ell_1 \sqcup \Gamma(\mathtt{pc}) \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e}$$

The rule Asgn says that if the label of x is at least as large as the larger of the label for the right-hand side and `pc`, then the assignment is well-typed. Thinking through a few cases, this means that whenever $\Gamma(\mathtt{pc}) = \mathtt{H}$, then $\Gamma(x)$ must be `H` because $\mathtt{H} \sqcup \ell = \mathtt{H}$ and $\mathtt{H} \not\sqsubseteq \mathtt{L}$. Really, if either of e or `pc` is `H`, then in order for the assignment to be well-typed then x must also be `H`. This seems like what we want, so we move to the next command.

Composition. Compared to the surprisingly nuanced reasoning we had to do for assignment commands, composition is relatively easy to think about. If we want to reason that a program $\alpha; \beta$ is well-typed, then the only rule that makes any sense is to require that both α and β also be well-typed on their own.

$$\text{(Comp)}\ \frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha; \beta}$$

The rule Comp above says exactly this.

Conditionals. Now we come to conditionals. Unlike the previous cases, conditionals raise the possibility of influencing control flow (represented in our type system by $\Gamma(\texttt{pc})$) on H-labeled data. We need to account for this in the typing rule, so that when we reason about whether the branches are well-typed we properly account for whether the pc might carry H data.

Recall that we treat type environments similar to updateable maps, so that if Γ is an environment, possibly containing a mapping for x, then $(\Gamma, x : \texttt{L})$ is the environment that maps x to L and everything else y to $\Gamma(y)$. Then to carry the type of the Boolean expression Q in the guard of a conditional to the type contexts of its branches, we want to use a context that maps pc to the least upper bound of the current pc, and the type of e.

$$\text{(If)}\ \frac{\Gamma \vdash Q : \ell \quad \ell' = \ell \sqcup \Gamma(\texttt{pc}) \quad \Gamma, \texttt{pc} : \ell' \vdash \alpha \quad \Gamma, \texttt{pc} : \ell' \vdash \beta}{\Gamma \vdash \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta}$$

The rule If above does this. Looking at the antecedent, we first type the guard expression Q as ℓ, and then compute the least upper bound ℓ' of ℓ and the current label of pc. We then check that both branches are well-typed under the environment where $\Gamma(pc) = \ell'$. If this is the case, then the conditional is well-typed.

While Loops. The last command that we need to derive a rule for is our looping construct `while`. Much like in the case of conditionals, while loops can leak information from H data to the program counter through their conditional test. Not surprisingly, the rule for while loops can use the same approach as that for conditionals, as shown in While below.

$$\text{(While)}\ \frac{\Gamma \vdash Q : \ell \quad \ell' = \ell \sqcup \Gamma(\texttt{pc}) \quad \Gamma, \texttt{pc} : \ell' \vdash \alpha}{\Gamma \vdash \texttt{while}(Q)\, \alpha}$$

It may seem strange that the rule for `while` is in some ways simpler than the one for `if` statements, as this was certainly not the case when we discussed axioms for proving safety. However, the type of reasoning that this type system does about program behaviors is significantly more "coarse" than what we did when proving arbitrary postconditions, and in this case the only special consideration that we need to account for is whether the loop flows H data to L state through the program counter.

2.2 Confinement

Before moving on, let's return to the type system's treatment of implicit information flows resulting from `H`-typed influence on control flow. To type a branching command, the rules first obtain the label ℓ of the condition, and then attempt to type the subcommands under a `pc` label that is the least upper bound of the current `pc` label $\Gamma(\texttt{pc})$ and ℓ. The point of this is to ensure that if a subcommand writes to a variable typed with a label that is lower than the current `pc` label, or lower than the label given to the condition, then the system will be unable to derive a judgement for the subcommand and thus its enclosing branching command.

The confinement lemma below formalizes this property, and as with the simple security lemma, will be a useful building block when we attempt to prove the soundness of the entire type system.

Lemma 3 (Confinement)**.** *Well-typed commands never write to variables below the* `pc` *label. More precisely, if* $\Gamma \vdash \alpha$*, then for every variable* x *assigned in* α*,* $\Gamma(\texttt{pc}) \sqsubseteq \Gamma(x)$*.*

Proof. As we did with simple security, we can prove this lemma by induction on the structure of α. The inductive definition of command syntax is:

$$\alpha, \beta \ ::= \ x := e \mid \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta \mid \alpha;\beta \mid \texttt{while}(Q)\,\alpha$$

So the only base case containing no sub-commands is assignment, and the rest will make use of the inductive hypothesis which says that confinement holds for any sub-command appearing in α. We proceed with the cases below.

Base case: assignment. If we can derive the judgement $\Gamma \vdash x := e$, then the rule Asgn tells us that $\Gamma(\texttt{pc}) \sqsubseteq \Gamma(x)$ because in fact $\ell_1 \sqcup \Gamma(\texttt{pc}) \sqsubseteq \Gamma(x)$, where $\Gamma \vdash e : \ell_1$, and we know that $\Gamma(pc) \sqsubseteq \ell_1 \sqcup \Gamma(\texttt{pc})$.

Inductive case: conditionals. If $\Gamma \vdash \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$, then If tells us that both $\Gamma, \texttt{pc} : \ell' \vdash \alpha$ and $\Gamma, \texttt{pc} : \ell' \vdash \beta$, where $\ell' = \ell \sqcup \Gamma(pc)$. But $\Gamma(\texttt{pc}) \sqsubseteq \ell'$, and the inductive hypothesis gives us that for any variable x assigned in either α or β, $\ell' \sqsubseteq \Gamma(x)$. So because $\sqsubseteq$ is transitive, $\Gamma(\texttt{pc}) \sqsubseteq \Gamma(x)$.

Inductive case: composition. This case follows immediately from the inductive hypothesis. If $\Gamma \vdash \alpha;\beta$ then by Comp we have $\Gamma \vdash \alpha$ and $\Gamma \vdash \beta$. The inductive hypothesis tells us that for all x assigned in α or β, $\Gamma(\texttt{pc}) \sqsubseteq \Gamma(x)$.

Inductive case: while loops. This case is very similar to that of conditionals. If $\Gamma \vdash \texttt{while}(Q)\,\alpha$, then While tells us $\Gamma, \texttt{pc} : \ell' \vdash \alpha$ where $\ell' = \ell \sqcup \Gamma(\texttt{pc})$ and $\Gamma \vdash e : \ell$. Then the inductive hypothesis says that any variables x assigned in α are typed such that $\Gamma(\ell') \sqsubseteq \Gamma(x)$. Then by transitivity of $\sqsubseteq$ and the fact that $\Gamma(pc) \sqsubseteq \ell$, we conclude that $\Gamma(\texttt{pc}) \sqsubseteq \Gamma(x)$. This completes the proof.

□

3 Soundness

Now we have introduced all of the typing rules, and proved two key lemmas that characterize what they accomplish. Our ultimate goal is to prove that the type system is *sound*, in the sense that if $\Gamma \vdash \alpha$, i.e. α is well-typed, then α satisfies non-interference under typing context Γ. More formally, we would like to prove Theorem 4.

Theorem 4 (Soundness of information flow type system)**.** *Let α be a program, and ω_1, ω_2 be states such that $\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2$. Moreover, let ω_1', ω_2' be states such that $\langle \omega_1, \alpha \rangle \Downarrow \omega_1'$ and $\langle \omega_2, \alpha \rangle \Downarrow \omega_2'$. Then for a type context Γ, if $\Gamma \vdash \alpha$ then $\omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2'$.*

How do we prove Theorem 4? Given the way things have gone so far with the lemmas, we might think to try an induction on the structure of α. This is a sensible initial guess, but we will run into problems when we attempt the case for `while` loops. To see why, first recall the big-step semantics for loops.

$$\frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \mathit{false}}{\langle \omega, \mathtt{while}(P)\, \alpha \rangle \Downarrow \omega} \qquad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \mathit{true} \quad \langle \omega, \alpha; \mathtt{while}(P)\, \alpha \rangle \Downarrow \omega'}{\langle \omega, \mathtt{while}(P)\, \alpha \rangle \Downarrow \omega'} \tag{5}$$

Were we to proceed with induction on the structure of α, the case where $\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \mathit{false}$ would not pose a problem because $\omega_1 = \omega_1'$ and $\omega_2 = \omega_2'$. But in the case where $\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \mathit{true}$, we would need to invoke the inductive hypothesis on $\langle \omega, \alpha; \mathtt{while}(P)\, \alpha \rangle \Downarrow \omega'$ in order to conclude that ω_1', ω_2' are $\mathtt{L}$-equivalent. But we should not even think about doing this, because it is not in any way induction! Notice that $\alpha; \mathtt{while}(P)\, \alpha$ is not structurally "smaller" than $\mathtt{while}(P)\, \alpha$, in fact it contains a copy of the loop. Invoking the inductive hypothesis in this way is tantamount to assuming what we are trying to prove, and is not sound reasoning.

3.1 Induction on the structure of the big-step derivation

To avoid the pitfall above, we need to find some kind of structure to do induction on that we know has a base case, i.e., that terminates. Notice that because we assume $\langle \omega_1, \alpha \rangle \Downarrow \omega_1'$ and $\langle \omega_2, \alpha \rangle \Downarrow \omega_2'$, we know that α must terminate when executed on ω_1 and ω_2, and that there is a derivation tree using the rules of the big-step semantics ending with $\langle \omega_1, \alpha \rangle \Downarrow \omega_1'$ (and another for ω_2, ω_2'). Perhaps we can do induction on the structure of that derivation. Specifically, perhaps we can reason that the property holds for all derivations of minimal size (i.e., one application of a semantic rule), and then break larger derivation trees into parts, assuming that the property holds for the smaller sub-derivations to prove that it holds for the entire tree.

This may seem like a new kind of inductive principle, but recall that we used it to prove the soundness of the propositional sequent calculus. The principle is the same as any other structural induction that we have done so far, but our cases come from the derivation rules of the big-step semantics $\Downarrow$. We will prove Theorem 4 by induction on $\langle \omega_1, \alpha \rangle \Downarrow \omega_1'$, giving an argument for each case of what the final derivation rule could be. The base cases correspond to derivation trees where only one rule was applied, which in this case is assignment commands. The inductive cases are those where the derivation tree may have more than one rule, and the inductive hypothesis that we use says that Theorem 4 holds for each immediate subtree.

So for example, when we do the case for `if` commands, there are two ways that the derivation could end.

$$\frac{\langle \omega, P\rangle \Downarrow_{\mathbb{B}} \mathit{true} \quad \langle \omega, \alpha\rangle \Downarrow \omega'}{\langle \omega, \mathtt{if}(P)\,\alpha\,\mathtt{else}\,\beta\rangle \Downarrow \omega'} \quad \frac{\langle \omega, P\rangle \Downarrow_{\mathbb{B}} \mathit{false} \quad \langle \omega, \beta\rangle \Downarrow \omega'}{\langle \omega, \mathtt{if}(P)\,\alpha\,\mathtt{else}\,\beta\rangle \Downarrow \omega'} \tag{6}$$

We will give a case for each rule, and our inductive hypothesis will let us assume that the theorem holds for $\langle \omega, \alpha\rangle \Downarrow \omega'$ (in the case where P evaluates to true), and $\langle \omega, \beta\rangle \Downarrow \omega'$ (in the case where P evaluates to false). More precisely, the inductive hypothesis in the *true* case would tell us that if $\Gamma \vdash \alpha$, then for any $\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2$ if $\langle \omega_1, \alpha\rangle \Downarrow \omega_1'$ and $\langle \omega_2, \alpha\rangle \Downarrow \omega_2'$ then $\omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2'$.

Before we continue with the proof, it is important to point out that our induction on the derivation structure will not include derivations for expressions that use $\Downarrow_{\mathbb{Z}}$ and $\Downarrow_{\mathbb{B}}$. These are different relations than $\Downarrow$, and while it is possible to include their derivation in this induction, it is not necessary. We proved Lemmas 2 and 3 for a reason, and as we will see, they give us everything that we need to know about expressions to complete the proof.

3.2 Soundness proof: `while` case

We will now complete the case for while loops, leaving the remaining cases as an exercise. So we would like to prove that if $\Gamma \vdash \mathtt{while}(Q)\,\alpha$, then for any ω_1, ω_2 where $\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2$ and ω_1', ω_2' where

$$\langle \omega_1, \mathtt{while}(Q)\,\alpha\rangle \Downarrow \omega_1'$$

Theorem 4 tells us that $\Gamma \vdash \mathtt{while}(Q)\,\alpha$, so the last rule used in the type derivation:

$$\text{(While)}\ \frac{\Gamma \vdash Q : \ell \quad \ell' = \ell \sqcup \Gamma(\mathtt{pc}) \quad \Gamma, \mathtt{pc} : \ell' \vdash \alpha}{\Gamma \vdash \mathtt{while}(Q)\,\alpha}$$

There are two cases for us to consider. Either the label ℓ' of `pc` used to type the subcommand α is `L`, or it is `H`.

Case $\ell' = \mathtt{L}$: Simple security (Lemma 2) tells us that for any variable x in e, $\Gamma(x) \sqsubseteq \ell'$. So by the assumption made in this case, and the fact that $\sqsubseteq$ is transitive, $\Gamma(x) = \mathtt{L}$. We can then apply the assumption from Theorem 4 that $\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2$ to reason that for some constant b, $\langle \omega_1, Q\rangle \Downarrow_{\mathbb{B}} b$ and $\langle \omega_2, Q\rangle \Downarrow_{\mathbb{B}} b$ (i.e., Q evaluates to the same thing in ω_1 and ω_2).

If $b = \mathit{false}$, then $\omega_1 = \omega_1'$ and $\omega_2 = \omega_2'$. Then the theorem holds because $\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2$, so $\omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2'$.

If $b = \mathit{true}$, then the case is a bit more nuanced. We know that the last steps of the big-step derivations of $\langle \omega_1, \alpha\rangle \Downarrow \omega_1'$ and $\langle \omega_2, \alpha\rangle \Downarrow \omega_2'$ were:

$$\frac{\langle \omega_1, P\rangle \Downarrow_{\mathbb{B}} \mathit{true} \quad \langle \omega_1, \alpha; \mathtt{while}(P)\,\alpha\rangle \Downarrow \omega_1'}{\langle \omega_1, \mathtt{while}(P)\,\alpha\rangle \Downarrow \omega_1'} \quad \frac{\langle \omega_2, P\rangle \Downarrow_{\mathbb{B}} \mathit{true} \quad \langle \omega_2, \alpha; \mathtt{while}(P)\,\alpha\rangle \Downarrow \omega_2'}{\langle \omega_2, \mathtt{while}(P)\,\alpha\rangle \Downarrow \omega_2'}$$

Moreover, the derivation of $\langle \omega_1, \alpha; \texttt{while}(P)\,\alpha \rangle \Downarrow \omega_1'$ and $\langle \omega_2, \alpha; \texttt{while}(P)\,\alpha \rangle \Downarrow \omega_2'$ ended with the step:

$$\frac{\langle \omega_1, \alpha \rangle \Downarrow \omega_1'' \quad \langle \omega_1'', \beta \rangle \Downarrow \omega_1'}{\langle \omega_1, \alpha; \beta \rangle \Downarrow \omega_1'} \quad \frac{\langle \omega_2, \alpha \rangle \Downarrow \omega_2'' \quad \langle \omega_2'', \beta \rangle \Downarrow \omega_2'}{\langle \omega_2, \alpha; \beta \rangle \Downarrow \omega_2'}$$

Then the inductive hypothesis tells us that $\omega_1'' \approx_{\Gamma,\mathtt{L}} \omega_2''$, and then using this fact that $\omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2'$. This proves the theorem for the case $\ell' = \mathtt{L}$.

Case $\ell' = \mathtt{H}$**:** In this case, the label of $\mathtt{pc}$ used to type α is $\mathtt{H}$, so we need to prove that there are no implicit flows. The confinement lemma (Lemma 3) tells us that for any variable x assigned in α, $\mathtt{H} \sqsubseteq \Gamma(x)$, and thus $\Gamma(x) \not\sqsubseteq \mathtt{L}$. So for every x where $\Gamma(x) \sqsubseteq \mathtt{L}$, $\omega_1(x) = \omega_1'(x)$ and $\omega_2(x) = \omega_2'(x)$. Using the assumption $\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2$, we can then conclude that $\omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2'$, completing the proof for this case.

The proofs for the remaining cases mirror the structure of the proof for `while` loops, splitting into cases where the label of `pc` is `L` and `H`, and apply the inductive hypothesis to assert `L`-equivalence of intermediate states corresponding to immediate subtrees in the big-step derivation. They are left as an exercise, and will help you gain familiarity with this type system, as well as the principle of induction on the structure of derivations, which is a powerful technique that is used widely to establish useful properties of deductive systems.

4 Completeness

Now that we have established the soundness of the information flow type system, it is natural to ask whether it is *complete* in the sense that if a program satisfies non-interference, then it is well-typed according to the rules.

Unfortunately, this is not the case, and the reason is the way that implicit flows are handled. Consider the following program, which satisfies the definition of non-interference (Definition 1) under type context $\Gamma = (x : \mathtt{H}, y : \mathtt{L}, \mathtt{pc} : \mathtt{L})$.

$$\texttt{if}(x = 0)\, y := 1\, \texttt{else}\, y := 1$$

The expression rule BinOp will only let us prove $\Gamma \vdash x = 0 : \mathtt{H}$, and so If requires that we derive $\Gamma, \mathtt{pc} : \mathtt{H} \vdash y := 1$. But Asgn then requires that $\mathtt{L} \sqcup \mathtt{H} \sqsubseteq \Gamma(y)$ and $\Gamma(y) = \mathtt{L}$, so we are unable to complete the type derivation. So in short, we are unable to type this command even though it satisfies non-interference.

Perhaps this should not come as a surprise, considering that non-interference is a semantic property and our typing rules are all based on the syntax of the program. If we want to prove that programs such as this one satisfy non-interference, then we must use the self-composition technique outlined in Lecture 10. Alternatively, we can try to find a way to write programs so that they always typecheck, avoiding control dependence on `H`-labeled data wherever it is not necessary. This is almost always the right approach to take: write the program in a way that makes it easy to prove the desired security property, rather than attempting to "retrofit" security as an afterthought.

References

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.