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-

$\textbf{val}\ \mathrm{fst} : a \to b \to a$
$\textbf{let}\ \mathrm{fst}\ x\ y = x$

$\textbf{val}\ \mathrm{both} : a \to b \to a * b$
$\textbf{let}\ \mathrm{both}\ x\ y = (x, y)$

$\textbf{val}\ \mathrm{add} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$
$\textbf{let}\ \mathrm{add}\ x\ y = x + y$

$\textbf{type}\ a\ b\ \mathsf{sum} = \mathsf{Left\ of}\ b \mid \mathsf{Right\ of}\ a$
$\textbf{val}\ \mathrm{branch} : \mathsf{bool} \to a \to b \to a\ b\ \mathsf{sum}$
$\textbf{let}\ \mathrm{branch}\ b\ x\ y = \textbf{if}\ b\ \textbf{then}\ \mathsf{Left}(x)\ \textbf{else}\ \mathsf{Right}(y)$

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

$$\begin{array}{l}
\textbf{val}\ \mathrm{fst} : [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ ]\ \mathsf{int} \\
\textbf{let}\ \mathrm{fst}\ x\ y = x \\[1em]
\textbf{val}\ \mathrm{both} : [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ ]\ \mathsf{int} * [\ b\ ]\ \mathsf{int} \\
\textbf{let}\ \mathrm{both}\ x\ y = (x, y) \\[1em]
\textbf{val}\ \mathrm{add} : [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ b\ ]\ \mathsf{int} \\
\textbf{let}\ \mathrm{add}\ x\ y = x + y \\[1em]
\textbf{val}\ \mathrm{branch} : [\ c\ ]\ \mathsf{bool} \to [\ a\ ]\ \mathsf{int} \to [\ b\ ]\ \mathsf{int} \to [\ a\ b\ c\ ]\ \mathsf{int} \\
\textbf{let}\ \mathrm{branch}\ b\ x\ y = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y
\end{array}$$

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

$$\begin{array}{rcl}
\text{Dependencies}\ \phi & ::= & \circ \mid \phi\ \alpha \\
\text{Types}\ \tau & ::= & \mathsf{bool} \mid \mathsf{int} \mid [\tau \cdot \phi] \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \\
\text{Expressions}\ M, N & ::= & \mathsf{true} \mid \mathsf{false} \mid n \mid x \mid M + N \mid \lambda x.\, M \mid M\ N \mid \Lambda \alpha.\, M \mid M\ [\phi] \\
& \mid & \textbf{if}\ N\ \textbf{then}\ M_1\ \textbf{else}\ M_2
\end{array}$$

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.

$$\begin{tikzcd}
& \ell_H \arrow[dl, dash] \arrow[dr, dash] & \\
\ell_{L_1} \arrow[dr, dash] & & \ell_{L_2} \arrow[dl, dash] \\
& \bot &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& \alpha\ \beta \arrow[dl, dash] \arrow[dr, dash] & \\
\alpha \arrow[dr, dash] & & \beta \arrow[dl, dash] \\
& \circ &
\end{tikzcd}$$

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, $\textit{password}$ or $\textit{gradebook}$ 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.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \mid \circ}\ \textsc{T-True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \mid \circ}\ \textsc{T-False} \qquad \frac{}{\Gamma \vdash n : \mathsf{int} \mid \circ}\ \textsc{T-Int}$$

$$\frac{\Gamma \vdash M : \mathsf{int} \mid \phi_1 \quad \Gamma \vdash N : \mathsf{int} \mid \phi_2}{\Gamma \vdash M + N : \mathsf{int} \mid \phi_1 \sqcup \phi_2}\ \textsc{T-Add}$$

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 $\alpha\ \beta$.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \mid \circ}\ \textsc{T-Var}$$

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.

$$\frac{\Gamma \vdash M : \tau \mid \phi}{\Gamma \vdash M : [\tau \cdot \phi] \mid \circ}\ \textsc{T-Consume} \qquad \frac{\Gamma \vdash M : [\tau \cdot \phi_1] \mid \phi_2}{\Gamma \vdash M : \tau \mid \phi_1 \sqcup \phi_2}\ \textsc{T-Produce}$$

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.

$$\frac{\dfrac{\dfrac{}{x : [\mathsf{int} \cdot \alpha], y : [\mathsf{int} \cdot \beta] \vdash x : [\mathsf{int} \cdot \alpha] \mid \circ}\ \textsc{T-Var}}{x : [\mathsf{int} \cdot \alpha], y : [\mathsf{int} \cdot \beta] \vdash x : \mathsf{int} \mid \alpha}\ \textsc{T-Produce} \qquad \ldots\ y : \mathsf{int} \mid \beta\ \ldots}{x : [\mathsf{int} \cdot \alpha], y : [\mathsf{int} \cdot \beta] \vdash x + y : \mathsf{int} \mid \alpha\ \beta}\ \textsc{T-Add}$$

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).

$$\frac{\Gamma \vdash N : \mathsf{bool} \mid \phi_b \quad \Gamma \vdash M_1 : \tau \mid \phi \quad \Gamma \vdash M_2 : \tau \mid \phi}{\Gamma \vdash \textbf{if}\ N\ \textbf{then}\ M_1\ \textbf{else}\ M_2 : \tau \mid \phi_b \sqcup \phi}\ \textsc{T-If}$$

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.

$$\frac{\Gamma \vdash N : \mathsf{bool} \mid \phi_b \quad \Gamma \vdash M_1 : \tau \mid \phi_1 \quad \Gamma \vdash M_2 : \tau \mid \phi_2}{\Gamma \vdash \textbf{if } N \textbf{ then } M_1 \textbf{ else } M_2 : \tau \mid \phi_b \sqcup \phi_1 \sqcup \phi_2}\ \text{T-IF?}$$

The first rule turns out to be just as expressive as this one. To see why, consider the following program in ML:

$$\begin{array}{l}\textbf{val } \text{branch}' : [\ b\ ]\ \mathsf{bool} \to [\ a\ ]\ \mathsf{int} \to [\ a\ b\ ]\ \mathsf{int} \\ \textbf{let } \text{branch}'\ b\ x = \textbf{if } b \textbf{ then } x \textbf{ else } 0\end{array}$$

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 $\mathit{password}$, 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.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash b : [\mathsf{bool} \cdot b] \mid \circ}\ \text{T-VAR}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash b : \mathsf{bool} \mid b}\ \text{T-PRODUCE} \quad \boxed{x} \quad \boxed{0}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \textbf{if } b \textbf{ then } x \textbf{ else } 0 : \mathsf{int} \mid a\ b}\ \text{T-IF}}{b : [\mathsf{bool} \cdot b], x : [\mathsf{int} \cdot a] \vdash \textbf{if } b \textbf{ then } x \textbf{ else } 0 : [\mathsf{int} \cdot a\ b] \mid \circ}\ \text{T-CONSUME}$$

$$\dfrac{\dfrac{\dots b \dots \qquad \dfrac{\dfrac{}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash x : [\mathsf{int}\cdot a] \mid \circ}\ \textsc{T-Var}}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash x : \mathsf{int} \mid a}\ \textsc{T-Produce} \qquad \boxed{0}}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash \textbf{if } b \textbf{ then } x \textbf{ else } 0 : \mathsf{int} \mid a\ b}\ \textsc{T-If}}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash \textbf{if } b \textbf{ then } x \textbf{ else } 0 : [\mathsf{int}\cdot a\ b] \mid \circ}\ \textsc{T-Consume}$$

$$\dfrac{\dfrac{\dots b \dots \qquad \dots x \dots \qquad \dfrac{\dfrac{}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash 0 : \mathsf{int} \mid \circ}\ \textsc{T-Int}}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash 0 : \mathsf{int} \mid a}\ \textsc{T-Weaken}}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash \textbf{if } b \textbf{ then } x \textbf{ else } 0 : \mathsf{int} \mid a\ b}\ \textsc{T-If}}{b : [\mathsf{bool}\cdot b], x : [\mathsf{int}\cdot a] \vdash \textbf{if } b \textbf{ then } x \textbf{ else } 0 : [\mathsf{int}\cdot a\ b] \mid \circ}\ \textsc{T-Consume}$$

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:

$$\dfrac{\Gamma, x : P \vdash N : Q}{\Gamma \vdash (\lambda x.\, N) : P \to Q}\ {\to}R \qquad\qquad \dfrac{\Gamma \vdash N : P \quad \Gamma, M\,N : Q \vdash O : \delta}{\Gamma, M : P \to Q \vdash O : \delta}\ {\to}L$$

$$\dfrac{\Gamma, x : \tau_1 \vdash M : \tau_2 \mid \circ}{\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2 \mid \circ}\ \textsc{T-Lam} \qquad\qquad \dfrac{\Gamma \vdash M : \tau_1 \to \tau \mid \phi \quad \Gamma \vdash N : \tau_1 \mid \circ}{\Gamma \vdash M\ N : \tau \mid \phi}\ \textsc{T-Ap}$$

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:

$$\dfrac{\Gamma, x : \tau_1 \vdash M : \tau_2 \mid \phi}{\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2 \mid \phi}\ \textsc{T-Lam?}$$

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.

$$\frac{\Delta, \alpha; \Gamma \vdash M : \tau \mid \circ}{\Delta; \Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\, \tau \mid \circ}\;\textsc{T-DepLam} \qquad \frac{\Delta; \Gamma \vdash M : \forall\alpha.\, \tau \mid \phi' \qquad \Delta \vdash \phi\ \mathsf{dep}}{\Delta; \Gamma \vdash M\ [\phi] : [\phi/\alpha]\tau \mid \phi'}\;\textsc{T-DepAp}$$

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.

$$\dfrac{\Delta;\Gamma \vdash M : \tau \mid \phi \qquad \Delta \vdash \alpha\ \mathsf{dep}}{\Delta;\Gamma \vdash M : \tau \mid \phi\ \alpha}\ \text{T-Weaken}$$

$$\dfrac{\Delta;\Gamma, x : \tau_1 \vdash M : \tau_2 \mid \circ \qquad \Delta \vdash \tau_1\ \mathsf{type}}{\Delta;\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2 \mid \circ}\ \text{T-Lam}$$

Let's try to type the identity function in our system. First, what does this look like in ML? The following seems reasonable:

```
val id : [ a ] int → [ a ] int
let id x =  x
```

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:

$$\dfrac{\dfrac{\dfrac{}{\cdot,\alpha;\ \cdot, x : [\mathsf{int}\cdot\alpha] \vdash x : [\mathsf{int}\cdot\alpha] \mid \circ}\ \text{T-Var} \qquad \dfrac{\cdots}{\cdot,\alpha \vdash [\mathsf{int}\cdot\alpha]\ \mathsf{type}}}{\cdot,\alpha;\ \cdot \vdash \lambda x.\, x : [\mathsf{int}\cdot\alpha] \to [\mathsf{int}\cdot\alpha] \mid \circ}\ \text{T-Lam}}{\cdot;\ \cdot \vdash \Lambda\alpha.\,\lambda x.\, x : \forall\alpha.\,[\mathsf{int}\cdot\alpha] \to [\mathsf{int}\cdot\alpha] \mid \circ}\ \text{T-DepLam}$$

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:

$$\dfrac{\dfrac{\cdots}{\cdot, b, c;\ \cdot \vdash \Lambda\alpha.\,\lambda x.\, x : \forall\alpha.\,[\mathsf{int}\cdot\alpha] \to [\mathsf{int}\cdot\alpha] \mid \circ} \qquad \dfrac{\ldots}{\cdot, b, c \vdash b\ c\ \mathsf{dep}}}{\cdot, b, c;\ \cdot \vdash \Lambda\alpha.\,\lambda x.\, x\ [b\ c] : [\mathsf{int}\cdot b\ c] \to [\mathsf{int}\cdot b\ c] \mid \circ}\ \text{T-DepAp}$$

We can easily handle higher-order functions, too! Consider the below ML program which executes some arbitrary function twice:

```
val twice : [ a ] int → ([ a ] int → [ a ] int) → [ a ] int
let twice x f =  f (f x)
```

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{f}\qquad\boxed{f\ x}}{\cdot,\alpha;\ \cdot,x:[\mathsf{int}\cdot\alpha],f:[\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha]\vdash f\ f\ x:[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Ap}}{\cdot,\alpha;\ \cdot,x:[\mathsf{int}\cdot\alpha]\vdash\lambda f.\,f\ f\ x:([\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha])\to[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Lam}}{\cdot,\alpha;\ \cdot\vdash\lambda x.\,\lambda f.\,f\ f\ x:[\mathsf{int}\cdot\alpha]\to([\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha])\to[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Lam}}{\cdot;\ \cdot\vdash\Lambda\alpha.\,\lambda x.\,\lambda f.\,f\ f\ x:\forall\alpha.\,[\mathsf{int}\cdot\alpha]\to([\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha])\to[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-DepLam}$$

Then we type check f via T-VAR, eliding the typing environment because it is the same as the conclusion:

$$\dfrac{\dfrac{}{\cdot,\alpha;\ \ldots\vdash f:[\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Var}\qquad\boxed{f\ x}}{\cdot,\alpha;\ \cdot,x:[\mathsf{int}\cdot\alpha],f:[\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha]\vdash f\ f\ x:[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Ap}$$

And finally we type check its argument, which contains another call to f, in much the same way:

$$\dfrac{\ldots f\ldots\qquad\dfrac{\dfrac{}{\ldots\vdash f:[\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Var}\quad\dfrac{}{\ldots\vdash x:[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Var}}{\cdot,\alpha;\ \ldots\vdash f\ x:[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Ap}}{\cdot,\alpha;\ \cdot,x:[\mathsf{int}\cdot\alpha],f:[\mathsf{int}\cdot\alpha]\to[\mathsf{int}\cdot\alpha]\vdash f\ f\ x:[\mathsf{int}\cdot\alpha]\mid\circ}\ \textsc{T-Ap}$$

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:

$$\begin{array}{l}
\textbf{val}\ \mathrm{map} : ([\ a\ ]\ \mathsf{int}\to[\ b\ ]\ \mathsf{int})\to[\ a\ ]\ \mathsf{int\ list}\to[\ b\ ]\ \mathsf{int\ list}\\
\textbf{let}\ \mathrm{map}\ f\ \mathit{lst}\ = \textbf{match}\ \mathit{lst}\ \textbf{with}\\
\qquad\qquad\qquad \mid [\,]\to[\,]\\
\qquad\qquad\qquad \mid \mathit{hd} :: \mathit{tl}\to f\ \mathit{hd} :: \mathrm{map}\ \mathit{tl}
\end{array}$$

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 $[\ a\ ]$ int list is hiding a second dependency environment, constrained to be empty– its true form is $[\ \ ]$ ([a] int) 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:

```
val map : ([ a ] int → [ b ] int) → [ l ] ([ a ] int) list → [ l ] ([ b ] int) list
let map f lst  = match lst with
                  | [ ] → [ ]
                  | hd :: tl → f hd :: map tl
```

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:

$$\text{const} \triangleq \lambda x.\, 1$$

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:

$$\text{const}\ x \equiv \text{const}\ y \text{ for all } x, y$$

[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.

$$impl \triangleq \lambda client.\ client\ [\circ]\ 4\ (\lambda password.\ \lambda attempt.\ password = attempt)$$

$$client \triangleq impl\ (\Lambda p.\ \lambda password.\ \lambda check.\ \textbf{if}\ check\ password\ 4\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

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 $\forall p.\ [\mathsf{int} \cdot p] \to \mathsf{int} \to \mathsf{bool}$.

$$check\ x\ 1 \equiv^{?} check\ y\ 1 \text{ for all } x, y$$

$$check'\ [\phi]\ x\ 1 \equiv check'\ [\phi]\ y\ 1 \text{ for all } \phi, x, y$$

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.