RESEARCH-ARTICLE

# Structural Information Flow: A Fresh Look at Types for Non-interference

**HEMANT GOUNI**, Carnegie Mellon University, Pittsburgh, PA, United States

**FRANK PFENNING**, Carnegie Mellon University, Pittsburgh, PA, United States

**JONATHAN ALDRICH**, Carnegie Mellon University, Pittsburgh, PA, United States



**Published:** 09 October 2025
**Accepted:** 12 August 2025
**Received:** 25 March 2025


# Structural Information Flow: A Fresh Look at Types for Non-interference

HEMANT GOUNI, Carnegie Mellon University, USA
FRANK PFENNING, Carnegie Mellon University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA

Information flow control is a long-studied approach for establishing *non-interference* properties of programs. For instance, it can be used to prove that a secret does not interfere with some computation, thereby establishing that the former does not leak through the latter. Despite their potential as a holy grail for security reasoning and their maturity within the literature, information flow type systems have seen limited adoption. In practice, information flow specifications tend to be excessively complex and can easily spiral out of control even for simple programs. Additionally, while non-interference is well-behaved in an idealized setting where information leakage never occurs, most practical programs *must* violate non-interference in order to fulfill their purpose. Useful information flow type systems in prior work must therefore contend with a definition of non-interference extended with *declassification*, which often offers weaker modular reasoning properties.

We introduce *structural information flow*, which both illuminates and addresses these issues from a logical viewpoint. In particular, we draw on established insights from the modal logic literature to argue that information flow reasoning arises from *hybrid logic*, rather than conventional modal logic as previously imagined. We show with a range of examples that structural information flow specifications are straightforward to write and easy to visually parse. Uniquely in the structural setting, we demonstrate that declassification emerges not as an aberration to non-interference, but as a *natural* and *unavoidable* consequence of sufficiently general machinery for information flow. This flavor of declassification features excellent local reasoning and enables our approach to account for real-world information flow needs without compromising its theoretical elegance. Finally, we establish non-interference via a logical relations approach, showing off its simplicity in the face of the expressive power captured.

CCS Concepts: • **Security and privacy** → **Information flow control**; • **Theory of computation** → *Modal and temporal logics*; Proof theory; • **Software and its engineering** → *Polymorphism.*

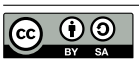
Additional Key Words and Phrases: information flow, security types, confidentiality, polarity, fine-grained, coarse-grained, dependency tracking, modal logic, polymorphism, declassification, existential quantification

**ACM Reference Format:**
Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2025. Structural Information Flow: A Fresh Look at Types for Non-interference. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 414 (October 2025), 27 pages. https://doi.org/10.1145/3764116

## 1 Introduction

Information flow control has long captured the interest of security researchers everywhere for its unique ability to establish *non-interference* [Goguen and Meseguer 1982], a powerful property which states that programs satisfying it cannot be manipulated to reveal sensitive information to untrusted parties. For instance, an e-mail notification system should never disclose password

Authors' Contact Information: Hemant Gouni, Carnegie Mellon University, Pittsburgh, USA, hsgouni@cs.cmu.edu; Frank Pfenning, Carnegie Mellon University, Pittsburgh, USA, fp@cs.cmu.edu; Jonathan Aldrich, Carnegie Mellon University, Pittsburgh, USA, jonathan.aldrich@cs.cmu.edu.




ACM 2475-1421/2025/10-ART414
https://doi.org/10.1145/3764116

data, so password processing should not interfere with the execution of the former. Types are the dominant mechanism for enforcing non-interference owing to its status as a *hyperproperty* [McLean 1996]—a question about a program that can only be answered by comparing *multiple* traces of its evaluation. Due to their innate ability to reason simultaneously over all possible program executions, type systems might be expected to offer a straightforward path to discharging non-interference invariants.

However, even in the face of their potential to stem the avalanche of security compromises faced by the computer industry, existing information flow systems have seen limited adoption. Current approaches to information flow reasoning burden programmers with complex specifications and inadequately modular mechanics. In this paper, we argue that the reasons for these deficiencies can be both explained and remedied with time-tested intuitions from the literature on modal logic, within which information flow has previously been couched [Miyamoto and Igarashi 2004]. The system crafted from this process, which we call *structural information flow*, is simpler and easier to use both for metatheory and for program reasoning despite being expressive enough to support practical programs. We start this paper with a brief introduction to information flow, requiring only some familiarity with statically typed functional programming as a prerequisite.

### 1.1 An Opinionated Crash Course in Information Flow

Programmers often express information flow properties—without any access to purpose-built systems for doing so—by leveraging parametric polymorphism [Reynolds 1984].[1] A simple example is the polymorphic identity function. The typing `id` : $\alpha$ -> $\alpha$ specifies that the return value must depend only on the data given as input. Likewise, the typing `second` : $\alpha$ -> $\beta$ -> $\beta$ expresses that its return value must depend only on its second argument. For a more interesting example, consider the typing of the standard `map` function on lists given here. The polymorphic components are the

$$\texttt{let map} : (\alpha \texttt{ -> } \beta) \texttt{ -> list } \alpha \texttt{ -> list } \beta$$

list elements, so this type communicates that the elements of the input list are permitted to flow into the higher-order argument, and that the elements of the output list depend on the return value of that argument. All three of these types express *information flow* properties: for a given computation, they relate its inputs to its outputs. However, these types also express *data abstraction* properties: `id` : $\alpha$ -> $\alpha$ expresses that the function should be able to be given an input at any type and return data at that same type. Both properties are consequences of *parametricity*, which states that not only does the argument flow to the return value, but that exactly the argument is returned.

Parametricity is quite useful, but its sheer strength greatly limits the range of programs we can write under it. What if we want to continue tracking information flow, but do not want to keep data abstract? For instance, we might like to write a function that takes as argument an integer, adds one to it, and returns it. We cannot, however, add one to data at type $\alpha$ because it is not known to be a number. Our first core intuition is that **parametric polymorphism offers two distinct features**, both **(1) providing data abstraction** and **(2) enforcing information flow properties.** We want to isolate the second, so let us try tagging types with *dependency variables* like $\alpha$, rather than having $\alpha$ *be* the type. Under this proposal, we might type the successor function on integers as $\alpha$ `int` -> $\alpha$ `int`. The $\alpha$ no longer has any role in data abstraction, but merely tracks the *identity* of the data—that is, on which data it depends. Note that this is distinct from the $\alpha$ in `list` $\alpha$, which permits the list to be generic over the type of its contents.

We are better off now than we were before—we can do interesting computation with data whose information flow content is being tracked—but are not quite there yet. For instance, try to write

[1] As distinguished from *ad hoc* polymorphism; the distinction is detailed in Strachey [2000].

down the information flow type of the addition function. We would like to be able to state that it takes two integers as arguments and returns an integer dependent on both. We can start out by writing `add : α int -> β int -> ?`. What should go in the `?`? Our current syntax is constrained to mention a single dependency per type, so we seem to be stuck. We can fix this by generalizing our type-level dependency variables to being ***sets* of dependency variables.** This is the other core intuition behind our system. We set `?` = `[ α β ] int`, which can be read as "this `int` depends on data from sources $\alpha$ and $\beta$." The resulting typing for addition is shown by `add`. The dependency

```
let add : [ α ] int -> [ β ] int -> [ α β ] int
```

variables on the arguments have also been turned into dependency sets for consistency. Keep in mind that these type signatures are polymorphic: now that we have generalized beyond single variables to sets of variables within our types, each variable in the set can be instantiated to another *set of variables*. Just as the $\alpha$ in $\alpha$ `->` $\alpha$ can be instantiated to `int` to obtain `int -> int`, the $\alpha$ in `add` can be instantiated to `[ secret1 secret2 ]`, representing data depending on some secrets. `add1` shows the resulting typing, taking as a first argument an an integer dependent on concrete sources `secret1` and `secret2`. It returns an integer dependent on `secret1`, `secret2`, and $\beta$.

```
let add1 : [ secret1 secret2 ] int -> [ β ] int -> [ secret1 secret2 β ] int
```

We haven't yet performed any meaningful information flow reasoning—what might a specification for preventing secret leakage look like in this setting? We consider below a simple password checker. We start by declaring some password data `pass`, which is a `string` tagged with `pwd` to indicate it contains password data. We assume `pwd` is in scope but defer detailing the mechanisms used to introduce it. Our checking function `check` takes a `string`—a password attempt—at any dependencies and returns a `bool` tagged with those dependencies and `pwd`, indicating whether the attempt was correct. This all seems to be as expected: the return value of `check` is dependent on

```
let pass : [ pwd ] string = "katya"
let check : [ α ] string -> [ α pwd ] bool = fun attempt -> attempt == pass
```

both its argument and `pass`, being the result of comparing them, so the type of the return value states exactly the same. If our purpose is to usefully detect whether password data is at risk of leaking, though, this seems too conservative: it is necessary to the function of the password checker that its boolean return value is permitted to leak.

Of course, it is intuitively a violation of non-interference to leak an arbitrary `bool` dependent on password data: consider the case where a function of the same type as `check` returns the $n$th bit of the password as a boolean value, where $n$ is the length of the argument string. Unexpectedly, we will be able to give `check`—that is, this and only this implementation of `check`—the type `[ α ] string -> [ α ] bool`. Section 4 will reveal precisely how. For a final example, consider the function `f`. What must `?` be?

```
let f : [ α ] bool -> [ ] bool = ?
```

The punchline of this paper, to be delivered in full in Section 5, will be that `f` must be constant in its argument. This is due to recovering non-interference as a flavor of parametricity over dependency variables. In fact, *any* function where the input dependency set is not a subset of the output must be the constant function. And with that, we conclude our introduction: the basic

notions we have introduced are all that will be needed for the structural information flow setting. The rest of this paper will elucidate the seemingly straightforward choices we have just made, grounding and justifying them via well-understood logical foundations and showing that the system created by these choices is expressive enough to capture information flow issues arising in the wild. In particular, the aforementioned desirable typing of `check` will turn out not to require any extensions. This is not the case with other approaches, which call this behavior *declassification* and add constructs to allow violations of non-interference. This often obstructs modular reasoning.

### 1.2 A Preview of the Rest

Section 2 will reveal that the language we have set up is a variant of hybrid logic. We have just discussed how to arrive at this language from parametric polymorphism, but it is also possible to find the way there by drawing on logical intuitions. We describe how to arrive at our setting beginning from prior work grounding information flow in conventional modal logic. Section 3 explores a number of further examples, touching on several subtle details of our system that significantly aid the straightforward and succint nature of our information flow specifications. This is detailed by comparing to equivalent programs written in systems without our insights. Section 4 introduces declassification by example, solving our issue with `check` above. Section 5 discusses the typing rules, comparing to those for hybrid logic and discussing the simple logical relations argument with which non-interference can be validated in our setting. Section 6 compares to other work investigating either declassification or the foundations of information flow. We conclude in Section 7. Our contributions are the following:

(1) We clarify and **re-cast the foundations of information flow** in the light of **hybrid logic** [Prior 1968], a well-studied generalization of modal logic designed around concerns we will show throughout this paper to be fundamental to information flow reasoning.
(2) We show that the design intuition imparted by hybrid logic has the potential to **simplify information flow specifications.** We extoll the virtues of performing information flow reasoning in terms of *dependency sets*, inspired by hybrid logic's *world paths*. We touch on the important role played by the proof-theoretic concept of *polarity* in determining the granularity of dependency tracking.
(3) Remarkably, the quantification machinery suggested by hybrid logic **realizes declassification fully internally** to the system. That is, our theory neither makes any explicit mention of declassification nor needs to be extended to support it. We remark on the nature of this as **computationally relevant information flow policies.**
(4) Our metatheory and proof of non-interference inherit the elegance and simplicity of the programmer-facing side of our system. Namely, we show how our logical relation inherently supports **non-interference reasoning in the presence of declassification**, automatically ignoring disequalities resulting from declassification by writ of quantification.

By the end of this paper, we will have taken the initial steps towards bringing to bear the full-throated no-concessions-made variant of non-interference as a practical—even desirable—regime under which to write secure programs.

## 2 Background, Logic, and Typing

Having introduced one way of arriving at the structural approach to information flow starting from ordinary functional programming, we will now reveal another starting from constructive modal logic. It will turn out that *hybrid logic* [Prior 1968], a generalization of modal logic, provides a more robust foundation for information flow reasoning than the standard modal setting in which most prior work has been cast. In particular, we will show that hybrid logic has been designed around a

number of considerations critical to information flow reasoning. We start by reviewing programs written under a more standard theory of information flow, then elaborate its logical structure in the subsections that follow.

### 2.1 Introduction: Round Two

Let us revisit the examples from the introduction within a conventional theory of information flow in order to build intuition. In particular, our examples are lightly inspired by the syntax of Flow Caml [Pottier and Simonet 2002], an information flow system for OCaml. Figure 1 compares the typing of the identity function on integers between our approach and a standard one.

```
let id : [ α ] int -> [ α ] int (* Ours *)
let id' : α int -> α int        (* Theirs *)
```

Fig. 1. Similar-looking types...

They look pretty similar. In fact, the standard typing looks much like ours did before we generalized from singular variables to *sets* of variables. How, then, might we type addition?

```
let add : [ α ] int -> [ β ] int -> [ α β ] int      (* Ours *)
let add' : α int -> β int -> δ int with α, β <= δ (* Theirs *)
```

Fig. 2. A first sighting of lattice constraints

Here, we catch our first sighting of the *lattice constraints* which ordinarily comprise information flow specifications [Denning 1976]. At the point in our prior exploration where we chose to generalize to sets of dependency variables rather than individual dependency variables for typing `add`, two roads diverged—and we took the one less traveled by. The other option was to add additional structure to the variables themselves, transforming them into elements from a *semilattice* rather than leaving them inert. This is the more common option, so we review it here.

A semilattice has one primitive operation: *join*, written $\sqcup$. We can join two dependency variables $\alpha$ and $\beta$ by writing $\alpha \sqcup \beta$, returning another dependency variable. The returned variable is interpreted to be greater than or equal to both $\alpha$ and $\beta$—specifically the *least* such variable. That is, $\sqcup$ generates a partial ordering $\sqsubseteq$ over the carrier set of dependency variables. $\alpha \sqsubseteq \beta$ means that $\alpha$ can be joined with other variables to produce $\beta$—in other words, $\beta$ represents information from $\alpha$. $\alpha \sqsubseteq \beta$ allows us to compare two dependency variables $\alpha$ and $\beta$ to check if data from $\alpha$ is permitted to flow to $\beta$.

We use both operators in the type for `add`, writing $\sqcup$ syntactically as a comma ‘`,`’ and writing $\sqsubseteq$ syntactically as `<=`. So we can parse the contents of the **`with`** clause as $(\alpha \sqcup \beta) \sqsubseteq \delta$, or in natural language, “both $\alpha$ and $\beta$ must be able to flow to $\delta$.” The dependency variable for the return value is $\delta$, and flows to the return value can be specified by way of partial orderings which place $\delta$ above other variables representing input information. When this function is called, the caller must instantiate $\alpha$, $\beta$, and $\delta$—just as we previously set $\alpha$ = `[ secret1 secret2 ]` in our typing for `add1` in Section 1.1—with concrete variables such that the two *former* variables are *both ordered less* than the *latter* one. This will satisfy the type constraints because $\sqcup$ is guaranteed to yield the least variable higher than both operands—so no higher than the instantiation of $\delta$. As an aside, note that all this constraint information must be digested by programmers to appreciate the type of `add'`, rather than relying on existing intuitions surrounding parametric polymorphism as in Section 1.1. The argument for the simplicity of the latter is already taking shape. Note that lattice-based systems ordinarily deploy *simplification algorithms* which attempt to elide constraints from types. We hold off on applying these until the next section to avoid obscuring the fundamental mechanics.

While picking through the above definitions, you may have noticed something curious. Our sets of dependency variables also have semilattice structure! Join is given by set union, which induces the partial order given by subset inclusion. When two sets are unioned, another set is produced which has exactly the elements needed to be the superset of both, and no more. This structure is known as the *free* semilattice in algebra. *Free* indicates that it is the *simplest possible way* to arrive at a semilattice starting with some carrier set—of dependency variables, in our case. We simply make each element of the set of variables into a singleton set to create the smallest elements of our semilattice, and apply set union to generate more elements until we reach closure. This is analogous to taking the power set of the carrier. Where before we used $\alpha_1$ <= $\alpha_2$ to denote that data from source $\alpha_1$ flowed to destination $\alpha_2$, we now have [ $\alpha$ $\beta$ ] to mean that data from sources [ $\alpha$ ] and [ $\beta$ ]—subsets of the former—flowed into that destination.

Sharing the same algebraic structure does not collapse the two approaches into one, however—far from it. As we will show in Section 3 and Section 4, the choice to use the free semilattice in our setting has made all the difference. But we are not yet prepared to discuss why. For now, let us review a final example from the prior section under a conventional information flow system. The retyped password checker is shown in Figure 3.

```
let pass : [ pwd ] string = "katya"          (* Ours *)
let check : [ α ] string -> [ α pwd ] bool =
    fun attempt -> attempt == pass
let pass' : pwd string = "katya"           (* Theirs *)
let check' : α string -> β bool with α, pwd <= β =
    fun attempt -> attempt == pass
```

Fig. 3. Comparing Our Password Checkers

The situation is much the same as in Figure 2. We must somehow betray in the return type of `check'` that it depends on `pwd` data. We can achieve this by specifying that the variable annotating the return type must be ordered greater than `pwd`. The flow from the argument $\alpha$ is accounted for exactly as before. We look now to the mechanics underlying both flavors of information flow.

### 2.2 Reconstructing Information Flow via Hybrid Logic

The standard approach to information flow can be recovered from constructive modal logic [Pfenning and Davies 2001] by way of *partial necessity* [Nanevski 2004], which provides an account of indexed $\Box$ (modal necessity) connectives. Miyamoto and Igarashi [2004] follow this approach, indexing the $\Box$ operator with elements $\ell$ from a semilattice. It is common [Abadi et al. 1999; Choudhury et al. 2022; Liu et al. 2024; Shikuma and Igarashi 2008; Tse and Zdancewic 2004] to furthermore eliminate the necessity semantics and transition to a *lax modality* [Fairtlough and Mendler 1997] by admitting extra axioms on $\Box_\ell$. The $\Box_\ell$ connective is kept around, because it is indexed with information flow machinery $\ell$, but retains none of its original purpose within modal logic. We cannot provide the full story here—it is provided in Gouni et al. [2025, Appendix A] for the interested reader—but alternative, cleaner logical foundations are possible.

We will describe how to arrive at the structural approach starting at constructive modal logic as before. Modal logic was originally designed around reasoning about the possible states of affair, or configurations of reality, that can be reached from our current one. These states are known as worlds. $\Box A$ can be read as "in all reachable worlds, the proposition $A$ will be true". In usual presentations of modal logic, reachability of worlds is defined by a relation—which in our case will be a partial order $\sqsubseteq$—on worlds $\ell$, where $\ell_1 \sqsubseteq \ell_2$ means that $\ell_2$ is reachable from $\ell_1$. This is called a

*Kripke semantics* and is separate from the syntax of modal logic, underlying it. Partial necessity as previously mentioned can be deployed to pull—or internalize—worlds into the syntax of the logic by indexing $\Box$ with worlds $\ell$: each world-indexed necessitation $\Box_\ell$ identifies the world $\ell$ at whose accessible worlds it is true. In other words, in order to access some information at world $\ell$, the current world $\ell'$ must have accounted for $\ell$ as a dependency—$\ell'$ must be reachable from $\ell$ in the semantics. The first core insight here is that **information flow variables denote worlds.** Worlds hold a special place at the heart of modal reasoning, though, and partial necessity offers little insight about them—it is oblivious to the fact that it is working with world indices.

This is a job for *hybrid logic* [Prior 1968], an alternative approach to generalizing standard modal logic *designed around internal reasoning about worlds*. Hybrid logic reifies worlds as a first-class syntactic construct rather than leaving them implicit in the semantic realm or judgemental structure, or relegating them to an index into an existing connective. It does this via a *satisfaction operator* $@_w A$ read as "at world $w$, proposition $A$ should be true." The usual introduction and elimination rules for $@_w A$ are given in **red** in Figure 4. The form of the typing judgement is $\Gamma \vdash M : A\,[\phi]$ and can be read "Under assumptions $\Gamma$, expression $M$ has type $A$ at world $\phi$."

$$\textsc{@I}\ \frac{\Gamma \vdash M : A\,[\phi]}{\Gamma \vdash M : @_\phi A\,[\phi']} \qquad \textsc{@E}\ \frac{\Gamma \vdash M : @_\phi A\,[\phi']}{\Gamma \vdash M : A\,[\phi]} \qquad \textsc{@E-new}\ \frac{\Gamma \vdash M : @_\phi A\,[\phi']}{\Gamma \vdash M : A\,[\phi * \phi']}$$

$$\frac{\dfrac{\Gamma \vdash M : A\,[\phi]}{\Gamma \vdash M : @_\phi A\,[\phi']}\ \textsc{@I}}{\Gamma \vdash M : A\,[\phi]}\ \textsc{@E} \qquad \frac{\dfrac{\Gamma \vdash M : A\,[\phi]}{\Gamma \vdash M : @_\phi A\,[\phi']}\ \textsc{@I}}{\Gamma \vdash M : A\,[\phi * \phi']}\ \textsc{@E-new} \qquad \frac{\Gamma \vdash M : A\,[\phi] \qquad \phi \sqsubseteq \phi * \phi'}{\Gamma \vdash M : A\,[\phi * \phi']}\ \textsc{Sub}$$

Fig. 4. Satisfaction in Hybrid Logic and its Soundness

This setup is adapted from Reed [2009]. Rather than abstract world variables $\ell$, the worlds in these rules consist of elements $\phi$ of a *free commutative monoid* generated from a set of atomic world variables $\alpha_1, \alpha_2, \ldots, \alpha_n$. That is, $\phi = \alpha_1 * \alpha_2 * \ldots * \alpha_i$. This is quite close to a semilattice, which is what we need for information flow. It is lacking only idempotency, or the property that $\alpha * \alpha = \alpha$, so we add it to turn each $\phi$ into an element of a *free semilattice*. The join operation $\sqcup$ is given by set union $\cup$ as described in Section 2.1. There are variants of hybrid logic which do not represent the structure of worlds using a free monoid or semilattice, but the guiding intuition behind hybrid logic is to pull as much of this structure as possible into the syntax. Elements from a parameterized semilattice $\ell$ and $\ell'$ have no meaning except that provided externally, outside the syntax, but join in a free semilattice $(\alpha * \beta) \sqcup \delta = \alpha * \beta * \delta$ is immediate from the notation of each element. The next section will show that this choice simplifies specifications; the section after will place it at the heart of a sound and compositional declassification mechanism.

We are not quite at a system suitable for information flow yet. We would like to be able to state that $M$ has dependencies $\phi$ at type $A$—something akin to $\Box_\ell A$—by saying $@_\phi A$, but it turns out this will not quite suffice. The problem is the @E rule, which wholesale replaces the current world $\phi'$ with the $\phi$ inside the satisfaction operator. For information flow, we need to keep track of the old world, as well. We must not forget the ambient security level! Our second core insight comes from Pfenning and Davies [2001], who suggest a solution in the form of *world paths*. **World paths keep track of the history of your traversals through worlds**, or the sequence of worlds you have 'walked through'. We update the elimination rule to @E-new, which now preserves the old world *path* $\phi'$ and joins it with the world *path* $\phi$ obtained by eliminating the satisfaction operator. Each

dependency $\alpha$ in the judgemental $\phi$ is a world previously traversed and so recorded on the path. For subtle reasons elucidated in Gouni et al. [2025, Appendix A] and by Nanevski [2004], no notion of necessity exists in this connective. In short, indexing the necessity connective with modal worlds, as with other approaches, precludes its removal. By instead using hybrid logic to syntactically reify worlds, the core machinery for information flow emerges as a matter of *satisfaction*.[2]

Note that @E-new is no longer symmetric to @I. This appears to break local soundness, as shown in the second row of Figure 4. The issue is that reducing away pairs of introductory and eliminatory rule applications should not change the type $A$ or world $\phi$ in a way that is unachievable through the other rules of the logic—that is, introduction followed by elimination does not let us prove anything we could not before. This obviously holds for @I and @E, as evidenced by the leftmost derivation, but not when the latter is replaced with @E-new, producing an extra $\phi'$ in the conclusion. Local soundness is retained via a subsumption rule Sub which permits the $M : A\ [\phi]$ at the top of each of the first two derivations to be used directly to prove $M : A\ [\phi * \phi']$. A different solution arises from exploiting the proof-theoretic concept of *polarity*. This is the strategy we will adopt when setting up our type system in Section 5.1. But in either case, local soundness is retained.

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{pass} : [\texttt{pwd}]\ \textbf{string} \in \ldots}{\ldots \vdash \texttt{pass} : [\texttt{pwd}]\ \textbf{string}\ [\alpha]}\ \textsc{Var}}{\ldots \vdash \texttt{pass} : \textbf{string}\ [\alpha * \texttt{pwd}]}\ \text{@E-new} \qquad \dfrac{\dfrac{\texttt{attempt} : [\alpha]\ \textbf{string} \in \ldots}{\ldots \vdash \texttt{attempt} : [\alpha]\ \textbf{string}\ [\texttt{pwd}]}\ \textsc{Var}}{\ldots \vdash \texttt{attempt} : \textbf{string}\ [\alpha * \texttt{pwd}]}\ \text{@E-new}}{\texttt{pass} : [\texttt{pwd}]\ \textbf{string},\ \texttt{attempt} : [\alpha]\ \textbf{string} \vdash \texttt{pass} == \texttt{attempt} : \textbf{bool}\ [\alpha * \texttt{pwd}]}\ \textsc{Equals}}{\texttt{pass} : [\texttt{pwd}]\ \textbf{string},\ \texttt{attempt} : [\alpha]\ \textbf{string} \vdash \texttt{pass} == \texttt{attempt} : [\alpha\ \texttt{pwd}]\ \textbf{bool}\ [\epsilon]}\ \text{@I}$$

Fig. 5. Derivation for the Body of check

Hybrid logic will offer one more fundamental insight, but before that, we have roughly all the tools we need to work through the body of check from Figure 3, shown in Figure 5. We will assume a typing for a comparison operator which requires both of its arguments to be at the same world path, and a variable rule which permits variables to be typed at any world. The syntax $[\ \delta\ \beta\ ]\ \mathbf{A}$ is interpreted as a satisfaction operator, namely $@_{\delta * \beta}\mathbf{A}$. We use $\epsilon$ for the empty world path following Reed [2009]. Reading from the bottom of the derivation, we start by applying @I—reading the rule itself bottom-up—to extract the **bool** from the satisfaction operator. We then apply our imagined equality rule, producing a goal for each operand. For the left goal, we continue bottom-up by applying @E-new to give us pass at type $[\ \texttt{pwd}\ ]\ \textbf{string}$, drawing pwd from the world path in its conclusion. We finish with the variable rule. The right operand is analogous.

What is the final affordance of the hybrid setting? From the logic perspective, one of the primary motivations for hybrid logic is in its explicit treatment of quantification over worlds. From the information flow side, observe in the examples we have seen the prevalence of quantification—or polymorphism—over dependencies. We have not yet written a single program that does not rely on dependency polymorphism, even in introductory cases, and the rest of this paper will not contain any. Generic programming is broadly useful, but in the information flow setting it becomes absolutely essential. An information flow system without polymorphism cannot express useful programs without significant amounts of duplication. You may need to rewrite the same function for almost every single call site, because each usage will likely differ in its information dependencies.

[2]Satisfaction here is at a *lax* modality [Fairtlough and Mendler 1997; Moggi 1989], due to @E-new structuring the judgemental $\phi$ as an effect [Katsumata 2014]. The latter is connected to possibility [Benton et al. 1998], which becomes lax in the absence of necessity [Nanevski 2004, §4.1.1]. Section 5.1 treats this effectful structure via *polarity*.

This is our third and last core insight: **quantification over dependencies is essential and must be a first-class concern.** Hybrid logic will be of assistance one last time, deploying its inbuilt ability to quantify over the worlds in its syntax. The setup for quantification will be both simple yet general enough to let declassification emerge as a consequence.

Observe the parallels between the core insights from the preceding story and those from Section 1.1. Our earlier introduction of dependency sets corresponds to world paths here, and polymorphism (or quantification) shows up fundamentally in both. As promised there, the affordances of our setup—the combination of world paths/dependency sets and quantification/polymorphism—will turn out to be the key to performing declassification in a modular, elegant way. We hold off on discussing these points until Section 4 and Section 5. The next section builds more intuition through a number of examples.

## 3 More Examples and Subtleties

In this section we review a few potentially subtle details that make the structural approach to information flow easier to use in practice. These do not emerge as ad-hoc heuristics, but are motivated by fundamental affordances bubbled up from the logical and type-theoretic foundations of our system. We start not with a feature we *have*, but with one we *lack*.

### 3.1 Uniformity, or Absence of Policies

In standard theories of information flow working in terms of an arbitrary semilattice, it is common to tweak the structure of the lattice to model information flow *policies*. For instance, imagine that Alice trusts Bob. We could have a two element semilattice where `alice` $\sqcup$ `bob` = `bob` and so `alice` <= `bob`. Why is this useful? Functions like in Figure 6 become typable.

```
policy alice <= bob

let expected : bob string = "nemmerle"
let msg_bob : alice string -> α string -> β string with α, bob <= β =
    fun alice_secret msg_str ->
        if alice_secret == expected then msg_str else panic
```

Fig. 6. Declaring a Custom Dependency Ordering

This program allows Alice to message Bob by passing `msg_bob` **`string`**s, which Bob can then read. Alice must provide the correct secret to `msg_bob` so Bob knows it is the right person. There is something odd going on here. We pass data at levels `alice` and $\alpha$ as the first two arguments to `msg_bob`. The computation of the function body is certainly dependent on both arguments: `alice_secret` is used in a conditional guard, and `msg_str` is returned from one of its branches. However, the constraints on the return dependency $\beta$ indicate that it only contains data from $\alpha$ and `bob`. This is because the `alice` dependency induced by comparing against `alice_secret` in the conditional guard is subsumed by the `bob` dependency induced by the `expected` variable against which it is compared. The policy declares that `alice` can flow into `bob`, so it does.

This can be expressed structurally, but not in this way. In particular, it is not possible in our setting to simply declare a partial ordering on dependencies [ $\alpha$ ] and [ $\beta$ ], because our dependencies are *inert*. They have no implicit structure, being determined by their syntax. [ $\alpha$ ] can only be partially ordered less than a dependency set that contains it, like [ $\alpha$ $\beta$ ]. This means that two dependency sets can be compared for partial ordering at a glance, without having to keep declared policies in working memory. **Our experience is that the usage of orderings which violate the one given**

**structurally are the *exception* rather than the *norm*, so should not be applied pervasively for the whole program.** Section 4.4 will show a *local*, *computationally relevant* alternative to the above, leveraging the same machinery as for declassification. We still have not made a strong case for the simplicity of our approach yet. We look to Figure 7 to make it.

```
let alc : α int -> α int
    with alice <= α
let bob : α int -> α int
    with bob <= α

let both : α int -> β int * δ int
    with bob <= δ
    and  alice <= β
    and  α <= β, δ
let both x = (alc x, bob x)
```

```
let alc : [ α ] int -> [ α alice ] int
let bob : [ α ] int -> [ α bob ] int

let both : [ α ] int ->
    [ α alice ] int * [ α bob ] int
let both x = (alc x, bob x)
```

Fig. 7. Alice and Bob Sharing a Computation

This program moderates flows between Alice and Bob, who want to perform computation together but *do not want their information to be intermingled or revealed to the other.* Looking first to the program on the left, `alc` and `bob` are the functions that represent their computations. Each takes as argument an `int` and mixes either `alice`'s or `bob`'s data into it. This is indicated by the `alice` and `bob` dependencies lower bounding the $\alpha$ dependency in their return types. Note that we are applying *simplification algorithms* in this example. We might have typed `alc` as given for `alc'` below. Instead the simplification algorithm recognizes that whatever the dependency level of

```
let alc' : α int -> β int
    with alice, α <= β
```

the argument passed to `alc`, it can be raised until it is above `alice`, which loses no generality and preserves the soundness of dependency tracking. Next, the function `both` operationally invokes `alc` and `bob` in each projection of a pair and returns the pair. That this computation respects the desired separation property is not easy to determine from the type, however. We must instead confront a bag of constraints which, once analyzed, will hopefully say what we want.

Looking to the conjoining program on the right, the terms are exactly the same. The types of `alc` and `bob` again state that the return type of each depends on its input and on data from `alice` and `bob`. The indication of this fact with `[ α alice ]` is arguably already more direct. We need no simplification algorithms to arrive at this type—it is the only one that accounts for the flows from the argument $\alpha$ and `alice`. The biggest difference is in the type of `both`, which states that data $\alpha$ from its argument flows to each element of the returned pair, and data from `alice` and `bob` flows separately to each projection. From this we immediately know that the separation property we wanted is preserved by `both`. If data from Alice had been passed to Bob, or vice versa, we would see a set `[ alice bob ... ]` containing dependencies from both. At a glance, we see nothing of the sort. It is of course possible to use `both` to violate the separation property, but this would again be obvious at the callsite; this is in line with prior work [Pottier and Simonet 2002]. Next, we reconsider the heuristic of making fine-grained dependency tracking pervasive.

```
let const : [ α ] int -> [ ] int
let const _ = 10

let alice : [ alice ] int
let alice = 0

let result : [?] int
let result = const alice
```

Fig. 8. The Constant Function

### 3.2 The Benefits of Explicit Satisfaction

We have so far carried an implicit assumption—mirrored by other information flow systems [Pottier and Simonet 2002]—that all dependencies are *tracked granularly*. That is, all values carry their dependencies with them through being passed into and returned from functions. As an illustration, look to Figure 8. What dependency set should the [?] be? The empty set `[ ]` of course! When we evaluate the call `const alice`, $\alpha$ gets instantiated to `[ alice ]`. $\alpha$ does not appear in the return type, so this has no effect. For the constant function, we want precise tracking. However, most functions are not the constant function: they have at least some arguments upon which the return value is guaranteed to depend. For instance, consider the type for `add` given in Section 1.1. Need it have any information flow content at all, since its return value will *always* depend on both of its arguments? The clean logical foundations of our system help us answer this.

*3.2.1 Polarity.* A tool from proof theory, *polarity*, suggests that we need not. Polarity classifies types into *positive*—defined by their constructors—or *negative*—defined by their behavior when used. Booleans and lists are positive because we think of them by the form of their inhabitants, like `True` and `Cons(...)`. Functions are negative because they are characterized by their behavior when we apply them to arguments, not by their implementation. Positive types are connected to *values*, and negative types to *computations* [Levy 1999]. Remember that the goal of information flow is to map the inputs of a computation to its outputs. Polarity implies that positive types need not have interesting information flow specifications, because they do not pertain to computations, but negatives must. In particular, positive types should not granularly track—that is, encapsulate—information flows, but negative types should. For instance, lists are a positive type, so the dependencies of its elements are not tracked separately but propagated to the dependency set for the entire list. Meanwhile, function types should not leak information dependencies contained in their bodies until they are called, so these dependencies must be captured in their return type. Not so for their arguments, which should be determined by the polarity of each argument type. Luckily, polarity does not force a predetermined coarseness of tracking on us, besides as a per-type default, but permits us to choose. The connective `[ α ] A`—which is the *satisfaction operator* from Section 2.2—is of negative type. If granular information flow tracking is desired within positive types, a satisfaction operator can be introduced to do so.

Our view on polarity in information flow provides the following: **the granularity of dependency tracking should be *type driven*, rather than using a heuristic of maximally precise tracking everywhere.** This will allow us to simplify types, writing the type of `add` as `int -> int -> int`. `int` is a positive type, so does not encapsulate any dependencies, and because this eliminates all dependencies on the arguments the return value need not encapsulate any either. The ambient world path—or security context—$\phi$ from the typing judgement in Figure 4 is given programmatic meaning now: it tracks dependencies not encapsulated inside the type. Though

we did not provide syntax for introducing and eliminating satisfaction in that section, we might imagine that the syntax `#`$e$ introduces it, internalizing the current ambient dependencies into the type. Thus if `x` has type **`int`** and the ambient security context is $\alpha$, the expression `#x` will have type `[` $\alpha$ `]` **`int`**. Similarly, we use `!`$e$ to eliminate satisfaction and move the dependencies annotated in the type of an expression from the type to the ambient security context. Putting these together, the expression `#(add !alice !alice)` would have type `[ alice ]` **`int`** as before. `alice` is tracked ambiently after `!alice` until `#(...)`.

Practically, we should be able to let the language infer satisfaction for us. An algorithm to do this seems straightforward enough: if we have an $e$ : **A** but need an expression of type `[` $\alpha$ `]` **A**, then attempt to apply satisfaction introduction. Analogously for the reverse direction. We leave the formulation of this algorithm as future work, assuming it for the time being for convenience. The important point is that we can exploit polarity to inform and control the granularity of dependency tracking. Prior work has pursued expressivity results [Rajani and Garg 2018] regarding different degrees of granularity, but has not identified the connection to polarity which informs *when* dependencies should be tracked.

*3.2.2 Dependency Elision.* Based on the ideas above, our system supports an interesting and useful type simplification pattern. The type for `map1` in Figure 9, specialized to lists of integers, precisely characterizes information flow for this function, and is comparable (even slightly better, due to the benefits noted in Section 3.1) to the types given for `map` by other information flow systems.

```
let map1 : [ α ] ([ β ] int -> [ δ ] int) -> [ σ ] list ([ β ] int) ->
    [ α σ ] list ([ δ ] int)
let map2 : ([ β ] int -> [ δ ] int) -> list ([ β ] int) -> list ([ δ ] int )
let map3 : (A -> B) -> list A -> list B
```

Fig. 9. Finding Simplifications in Map

Reading from left-to-right, we first annotate the function argument given to `map1` with $\alpha$ so that when it is used inside the body it can induce an $\alpha$ dependency. The function itself takes a `[` $\beta$ `]` **`int`** as argument, which is the type of the contents of the list, and returns a `[` $\delta$ `]` **`int`**, the type of the contents of the returned list. This does not suffice to describe the dependencies of either the argument or the returned lists, though, because while $\beta$ and $\delta$ describe the dependencies of their *contents*, the *structure* of the list may itself have dependencies. For instance, the length of a list may betray information about the number of bits in a cryptographic key. Since lists are positive types, it would not ordinarily be allowed to treat these distinctly, but we manually do so by using a satisfaction type for the elements. So we introduce a $\sigma$ dependency for the argument list to represent the structure information. The structure of the returned list is dependent on both $\sigma$ and $\alpha$, because the dependencies from the higher-order argument must be captured in the return value.

Precision can be useful, but this type is more complicated than we might like. Looking at the type of `map1` more carefully, we see that $\alpha$ and $\sigma$ both occur in outermost satisfaction types in the argument types and on the return value. When such a pattern arises, the corresponding variables can be removed entirely without losing any precision, which we call *dependency-elision*. The unmentioned dependencies on the arguments will then be propagated ambiently to the whole application expression. The resulting type is given for `map2`. This simplification is not possible in systems which do not follow the directive given by polarity and instead track dependencies maximally granularly everywhere [Liu et al. 2024; Pottier and Simonet 2002], for instance forcing positive types like **`list`** to always hold their structural dependencies. A further simplification can be made in this case: the standard polymorphic type of map, reproduced in `map3`, now captures

that of map2. So it can be used to track information flow with no loss in precision from map1. For simplicity and clarity the formal system in Section 5 focuses on dependency polymorphism; we leave an extension of the system that supports type polymorphism to future work.

## 4 Declassification

The essence of declassification, as hinted, is *quantification* and *dependency sets* (i.e. world paths from hybrid logic). We rely on the same fundamental machinery used to ensure modularity across most modern typed languages. In that respect it should be uncontroversial. The core of the technique can be subtle for those unfamiliar with existentials from prior work on type abstraction [Mitchell and Plotkin 1985], but we will use simple examples to make the ideas more accessible.

### 4.1 Explicit, Higher-Rank Quantification and Dependency Sets

When we typed the identity function as with id_implicit in Figure 10, we omitted the bindings of the $\alpha$ variables. We now give a more explicit version as id_explicit, matching the formal system to be described in Section 5. The difference is the **forall** $\alpha$ sitting in front of the type signature, called a *quantifier*. This construct acts as a binder for $\alpha$: where **let** binds term-level variables, **forall** binds type-level variables.

```
let pass : [ pwd ] string = "katya"

let id_implicit : [ α ] int -> [ α ] int
let id_explicit : forall α . [ α ] int -> [ α ] int

let v1 : [ pwd ] int = id_implicit pass
let v2 : [ pwd ] int -> [ pwd ] int = id_explicit [ pwd ]
let v2' : [ pwd ] int = v2 pass
```

Fig. 10. Exposing the Type of the Identity Function

v1 shows the function id_implicit being applied as we have done so far, simply passing it an argument with dependencies [ pwd ] and expecting that the $\alpha$ in its type will change to reflect these dependencies. v2 shows the plumbing: we first instantiate id_implicit to [ pwd ], whereupon the type system substitutes away the $\alpha$ for that set of dependencies. The type that results is a function with nearly the same input and output types, but which is no longer polymorphic in $\alpha$. In v2' we apply v2 to the same argument pass as before—which matches the expected dependency set [ pwd ]—yielding the same type as in v1. **forall**s are usually handled transparently when polymorphism is in use, but declassification will require us to explicate them.

### 4.2 'Where' Declassification: Disappearing Dependencies with Quantification

We now illustrate declassification in our system, using the what-where-when-who framework of Sabelfeld and Sands [2009] to structure our discussion. We start by asking *where can quantifers go?* They have so far appeared exclusively and implicitly in *prefix* position, or at the beginning of the function signature. Consider the type **higher_rank** in Figure 11. The **forall**-quantified $\beta$ here is no longer in prefix position, because it has been moved inside the higher-order function. This is called *higher-ranked* quantification.

Here our goal is to allow a function defined by client code to compute with a secret number without being able to reveal it to the outside world until the computation is done. When the computation finishes, the final answer is revealed. Observe that num in client has type [ $\beta$ ] **int**.

```
type higher_rank = (forall β . [ β ] int -> [ β ] int) -> int

let impl : higher_rank = fun compute -> compute [ ] 7

let client : int = impl (fun num -> num + 123)
```

Fig. 11. Higher-Rank Quantification

We add 123 to it, and then return it as the result of the higher-order function. The whole function is passed to `impl`, which gives us back an **`int`**. Inspecting `impl` we see that it sets `num` to 7. So the higher order function passed to it by `client` will add 123 to 7. The sum 130 is returned to `client` as the result of the computation. This is dependent on the initial value of `num`, which has dependency $\beta$, but $\beta$ does not occur in the type of `client`! Indeed, $\beta$ is not in scope there. The reason $\beta$ disappears is because `client` is polymorphic in $\beta$. This means that its logic must be written without knowing what $\beta$ actually *is*—as though $\beta$ could be anything. `impl` takes advantage of this by instantiating $\beta$ to `[ ]`. Inside the higher-order function in the body of `client`, the dependency $\beta$ must be treated like any other. However, when 130 is sent to `impl`, where $\beta$ is set to `[ ]`, it returns a `[ ]` **`int`**. We implicitly unwrap the now-unnecessary satisfaction operator to yield just `[ ]` **`int`**.

In Sabelfeld and Sands's framework, we bound *where* the disappearance—or declassification—occurs to the lexical scope for $\beta$. Observe that this is *not* declassification in the typical [Sabelfeld and Sands 2009] sense of violating the faithfulness of dependency tracking. Rather, we simply exploit the internal knowledge of $\beta$'s emptiness to eventually elide it. This work demonstrates that declassification in the usual sense is not needed to expose the programming facilities offered by it.

### 4.3 'What' Declassification: Revisiting Password Checking

We can now solve the problem with `check` from the introduction, which will involve controlling *what* information can be declassified. Our solution is laid out in Figure 12. We start by generalizing the schema of higher-ranked quantification, represented by the **`passwd_checker`** type. There is now a quantified variable $\alpha$ which allows us to return data at any dependencies. Importantly, $\alpha$ *cannot* mention $\pi$ because it is scoped outside of it. In fact, the $\pi$ is not *universally* quantified anymore—another name for **`forall`**—but *existentially* quantified. The ability to encode existential quantification by leveraging universal quantification as shown is discussed in Girard et al. [1989, §11.3.5]. **Existentials allow us to realize declassification fully generally.**

$\pi$ plays the same role as `pwd` from our first attempt in Section 1.1. As in the preceding section we use a higher-order function to allow the client to compute on a secret value, then declassify the final result. This time, however, we provide the client with a secret value `pass` along with *methods* that can manipulate it: `check` and `hash`. To describe an interface exposing these elements we use the $F(\pi)$ notation, which is a template that takes a dependency variable as an argument and splices in a record containing methods typed at that variable. `impl` works on the same principle as before, instantiating the existential dependency $\pi$ to the empty set of dependencies. It lives up to its namesake, providing implementations of each of the exposed methods. The clients are more interesting. `client1` exhibits a standard usage, instantiating $\alpha$ to `[ ]` and using `check`. `client1` is of type **`bool`**, as was promised in Section 1.1. So we have successfully declassified exactly the **`bool`** resulting from the password check. `client2` attempts to instantiate $\alpha$ with `[ π ]` so it can try to do the password check without going through `check`, but existential quantification bars it from doing so. $\pi$ is not in scope at the point of instantiation! `client3` instantiates $\alpha$ to `[ ]`, but again accesses password data through `imports.pass`. This induces a $\pi$ dependency—since the template was called with $\pi$ as an argument—which will not check against the empty set.

```
F(π) ≜ {
    pass : [ π ] string,
    check : forall β . [ β ] string -> [ β ] bool,
    hash : forall β . [ π β ] string -> [ β ] string
}

type passwd_checker = forall α . (forall π . F(π) -> [ α ] bool) -> [ α ] bool

let impl : passwd_checker = fun compute -> compute [ ] {
    pass = "katya",
    check = fun attempt -> attempt == pass,
    hash = fun pass_str -> sha256sum pass_str,
}

let client1 : bool = impl [ ] (fun imports -> imports.check "arren")
let client2 : bool = impl [ π ] (fun imports -> "arren" == imports.pass) ⚠
let client3 : bool = impl [ ] (fun imports -> "arren" == imports.pass) ⚠
```

Fig. 12. A Fancier Password Checker

There is one point left to illuminate. Look to hash, which permits password-dependent strings' hashes to be leaked. From a client's perspective, such a function is possible when specified using world paths / dependency sets, but *is not possible under an arbitrary semilattice-based theory of information flow*. When calling hash, the information flow content of the data passed to hash's first argument must be able to be uniquely decomposed into the dependencies which comprise it. This allows hash to 'match' on dependencies in its input type and remove them, as it does with $\pi$. Joining in arbitrary lattices does not necessarily preserve information about the inputs, preventing the client from performing this decomposition, but the same in a free semilattice does.

Scaling the existential approach to practical programs necessitates being able to express declassifiers like hash. Otherwise, one is relegated to declassifying only by virtue of exported methods which *do not induce* some dependency, like check, rather than being able to *actively remove* that dependency in the style of hash. For instance, you may wish to perform some string processing on password data—say, padding it with a nonce—before hashing it. If we could not express functions like hash which remove dependencies from a given computation, this would require a specialized function in the style of check which does the desired padding, followed by hashing, to be exported from the password checker interface. In the general case, each individual use-case would require specialized support from the password checker itself. This is neither modular nor scalable.

The general problem here is highlighted by Cruz and Tanter [2019], who import the the machinery of *faceted types* to address it. It is remedied here without the need for specialized modifications to the type system. At an intuitive level, declassifying functions like hash can be seen as a computationally relevant ordering on dependency sets which may violate the one given structurally. Specifically, hash can be read as an ordering $[\ \pi\ \delta\ ] \sqsubseteq [\ \delta\ ]$—note that the left hand side is not a subset of the right—that must be manually applied wherever it is used, transforming expressions at $[\ \pi\ \delta\ ]$ into those at $[\ \delta\ ]$. So this allows us to preserve the uniform structure of our information flow specifications while, in effect, introducing information flow policies on them. Let us review the example in Section 3.1 where we confronted policy declarations to see if we can capture them now.

```
open Alice with [ bob ] importing
    alice,
    reveal_alice : [ alice ] string -> [ bob ] string

let expected : [ bob ] string = "nemmerle"
let msg_bob : [ alice ] string -> [ α ] string -> [ α bob ] string =
    fun alice_secret msg_str ->
        if reveal_alice alice_secret == expected then msg_str else panic
...
```

Fig. 13. Declaring a Custom Dependency Ordering, Computation-Relevantly

### 4.4 'Who' Declassification: Alice talks to Bob

The program in Figure 13 uses a slightly higher-level syntax for existential dependencies, closer to what programmers would see while employing our approach to declassification. In particular, we might imagine that **Alice** is defined as follows, where $\alpha$ is used to propagate the output

```
type Alice = forall α β .
    (forall alice . F(alice, β) -> [ α ] A) -> [ α ] A
```

dependencies as before and `alice` is existentially quantified. **with** instantiates $\beta$ with `[ bob ]`, which propagates it to the interface template $F(\ldots)$. The **importing** clause brings into scope the existential variable `alice` and a function `reveal_alice` from the instantiated interface $F$(`alice`, `bob`). `bob` is assumed to be in-scope. Functionally, this program revisits Figure 6, showing how our system can model an information flow policy that allows Alice's data to be sent to Bob. This corresponds to the *who* dimension of declassification from Sabelfeld and Sands [2009].

Looking to the client machinery, assume $\alpha$ is instantiated as needed for the eventual return dependencies of the program in Figure 13, and similarly that A is as needed for the program's eventual return type since our core system lacks polymorphism over types. The purpose of the `reveal_alice` function is to declassify `alice` data to Bob by relabeling it as `bob` data. As in the prior two examples, the `reveal_alice` function can be implemented by the **Alice** module because the latter has internally instantiated `alice` to a convenient dependency set—perhaps `[ ]` or `[ bob ]`. The only difference compared to Figure 6 is that `reveal_alice` must be called explicitly, rather than the flow being permitted implicitly. As a result, `reveal_alice` can transform Alice's data arbitrarily before revealing it to Bob, such as by redacting certain information or cryptographically signing it. This is what is meant by *computationally relevant*: **policies and declassification inherently have computational content in our system.** In our view, information flow policies and declassifiers are *one and the same*; there should be no distinction between the two. The setup of both within our system makes this apparent. This is particularly desirable in the context of, say, revealing secrets: it is rare that a secret value should be leaked fully intact rather than after some redacting computation. The computational *irrelevance* of ordinary policy declarations makes them unfit to serve as a declassification mechanism, so computational relevance can be seen as unifying the two.

We could extend this example by introducing another existential module **Bob** which permits messages tagged with `bob` to be read without inducing a dependency, effectively declassifying messages after processing them. Only *when* declassification remains of the major flavors of declassification [Sabelfeld and Sands 2009]. This is easy enough: simply integrate the ordering constraints into the function type which performs declassification. For instance, to declassify bids only after an auction has closed, require the auction to run to completion before running a callback to declassification.

Dependency $\phi$ Type $A, B$ Expr $e, v$ Dependency Vars $\alpha_1, \alpha_2, \ldots \in \Delta$ Vars $x_1, x_2, \ldots \in \Gamma$

$$\begin{aligned}
\text{Dependencies } \phi &::= \circ \mid \phi; \alpha \\
\text{Types } A &::= \mathtt{unit} \mid [A \cdot \phi] \mid A_1 \to A_2 \mid \forall(\alpha.A) \\
\text{Expressions } e, v &::= \langle\rangle \mid x \mid \#e \mid \,!e \mid \lambda(x.e) \mid \mathsf{ap}(e_1; e_2) \mid \Lambda(\alpha.e) \mid e[\phi]
\end{aligned}$$

$$\text{T-Unit}\ \frac{}{\Delta;\Gamma \vdash \langle\rangle : \mathtt{unit} \mid \circ} \qquad \text{T-Var}\ \frac{}{\Delta;\Gamma, x : A \vdash x : A \mid \circ} \qquad \text{T-Consume}\ \frac{\Delta;\Gamma \vdash e : A \mid \phi}{\Delta;\Gamma \vdash \#e : [A \cdot \phi] \mid \circ} \qquad \text{T-Produce}\ \frac{\Delta;\Gamma \vdash e : [A \cdot \phi_1] \mid \phi_2}{\Delta;\Gamma \vdash\, !e : A \mid \phi_1 \sqcup \phi_2}$$

$$\text{T-Lam}\ \frac{\Delta;\Gamma, x : A_1 \vdash e : A_2 \mid \circ \qquad \Delta \vdash A_1}{\Delta;\Gamma \vdash \lambda(x.e) : A_1 \to A_2 \mid \circ} \qquad \text{T-Ap}\ \frac{\Delta;\Gamma \vdash e : A_1 \to A_2 \mid \phi \qquad \Delta;\Gamma \vdash e_1 : A_1 \mid \phi_1}{\Delta;\Gamma \vdash \mathsf{ap}(e; e_1) : A_2 \mid \phi \sqcup \phi_1}$$

$$\text{T-DepLam}\ \frac{\Delta, \alpha;\Gamma \vdash e : A \mid \circ}{\Delta;\Gamma \vdash \Lambda(\alpha.e) : \forall(\alpha.A) \mid \circ} \qquad \text{T-DepAp}\ \frac{\Delta;\Gamma \vdash e : \forall(\alpha.A) \mid \phi' \qquad \Delta \vdash \phi}{\Delta;\Gamma \vdash e[\phi] : [\phi/\alpha]A \mid \phi'} \qquad \text{T-Sub}\ \frac{\Delta;\Gamma \vdash e : A_1 \mid \phi \qquad A_1 \sqsubseteq_\Delta A_2}{\Delta;\Gamma \vdash e : A_2 \mid \phi}$$

Fig. 14. TS/SCI: Type System for the Structural Calculus of Indistinguishability (Core Rules)

## 5 Metatheory

We have surveyed a zoo of interesting examples ranging beyond those in Section 1.1, but have not introduced any new primitive notions! Even existential quantification as revealed in the last section is simply re-using the same machinery introduced initially for polymorphism. This makes our job in this section relatively straightforward.

### 5.1 Syntax and Typing: A Hybrid Type System

The core syntax and typing rules of our system, the *Structural Calculus of Indistinguishability*, are shown in Figure 14. Our typing judgment is $\Delta;\Gamma \vdash e : A \mid \phi$ and can be read "Under in-scope dependency variables $\Delta$ and in-scope term variables $\Gamma$ the expression $e$ has type $A$ with set of dependencies $\phi$." Dependencies $\phi$ play the same role as in Section 2.2, reifying the *ambient security level* from Section 3.2.1. The type $[A \cdot \phi]$ corresponds to a satisfaction operator $@_\phi A$ and uses the syntax $\#e$ and $!e$ for introduction and elimination. The corresponding rules T-Consume and T-Produce look familiar, nearly mirroring @I and @E-new modulo syntax. The biggest difference is that T-Consume concludes at the empty dependency set $\circ$, which will be important.

Starting simple, the introduction rule for `unit` states that a unit expression $\langle\rangle$ incurs no dependencies. The variable typing rule shows the structure of the typing context $\Gamma$, which contains variables at a particular type. Unlike other information flow systems, we do not annotate variables in the context with security levels. They may carry dependency information in their type—using $[A \cdot \phi]$—if needed, but the entries in the context themselves are not annotated. This is in line with the interpretation of the $\phi$ in the typing judgment as an *effect*. Only computations should have effects; variables, being connected in a call-by-value language to values, should not [Levy 1999].

Moving on to functions, T-Lam is relatively standard. Strangely, however, it requires that its body have no dependencies. This is a way of forcing its body to consume all its dependencies—that is, represent them within its type $A_2$ using T-Consume—before a lambda is allowed to form around it. Forcing dependencies into function types is in line with the intuitions about granular tracking

and polarity from Section 3.2.1. There is also a premise $\Delta \vdash A_1$, which we will return to shortly. The application rule T-Ap propagates the function and argument's dependencies to the application expression in the conclusion, but is otherwise as usual. T-DepLam is the introduction rule for the quantifier type $\forall(\alpha.A)$. It again requires the dependencies in its body to be consumed. Its premise introduces an $\alpha$ into $\Delta$, the environment responsible for tracking the in-scope information flow variables. This can be thought of analogously to the introduction rule for type abstraction in System F [Girard 1986]. The elimination rule T-DepAp for quantifiers is instantiation, substituting the instantiated $\phi$ into the inner type of the quantifier. This is analogous to type application in System F. The type of `id` in Figure 1 under this syntax might be $\forall(\alpha.[\mathsf{int} \cdot \alpha] \to [\mathsf{int} \cdot \alpha])$, using int as a base type. A term under this type is $\Lambda(\alpha.\lambda(x.x))$, but more are possible by using $!x$ to extract out the underlying int and compute with it as in $\Lambda(\alpha.\lambda(x.\#(!x + !x)))$.

The premise $\Delta \vdash \phi$ appearing in T-DepAp is toward the same end as $\Delta \vdash A_1$ from T-Lam. Its purpose is to ensure the well-scopedness of all elements of the typing judgement within any derivation. We want to be sure that whenever we have a valid type derivation, all dependencies $\alpha$ in the expression $e$, type $A$, and dependency set $\phi$ in the typing judgement are contained within $\Delta$. We also want to ensure that any term variables $x$ in $e$ are contained in $\Gamma$. Regularity establishes this rigid lexical scoping. $\Delta \vdash A$ can be read as "all dependencies $\alpha$ mentioned in $A$ are members of $\Delta$"; analogously for the other three scoping judgements.

**Theorem 5.1** (Regularity). *If $\Delta; \Gamma \vdash e : A \mid \phi$ and $\Delta \vdash A_i$ for each assumption $x_i : A_i$ in $\Gamma$, then $\Delta \vdash e$ and $\Gamma \vdash e$ and $\Delta \vdash A$ and $\Delta \vdash \phi$.*

Proof. By induction on a derivation of $\Delta; \Gamma \vdash e : A \mid \phi$. □

Observe that all introduction rules for the connectives so far introduced, namely $[A \cdot \phi]$, $A_1 \to A_2$, and $\forall(\alpha.A)$, conclude at the empty set of dependencies $\circ$. This is because they are *negative connectives*. In line with the reasoning in Section 3.2.1, **all negative connectives must consume their dependencies.** This also provides a solution to the local soundness issue noted with the satisfaction rules in Section 2.2, because T-Consume (unlike @I) does not create a spurious $\phi'$ in its conclusion. The derivation is reproduced here.

$$\dfrac{\dfrac{\Delta; \Gamma \vdash e : A \mid \phi}{\Delta; \Gamma \vdash \#e : [A \cdot \phi] \mid \circ}\ \text{T-Consume}}{\Delta; \Gamma \vdash !\#e : A \mid \phi}\ \text{T-Produce}$$

Finally, T-Sub allows *subsumption*, which enables us to raise the security levels of programs' types. Reading from the top, if one has $e$ at type $A_1$ and a proof that $A_1$ is a subtype of $A_2$, then T-Sub provides $e$ at type $A_2$. We have $[A_1 \cdot \phi_1] \sqsubseteq_\Delta [A_2 \cdot \phi_2]$ when $\phi_1 \subset \phi_2$ and $\Delta \vdash \phi_2$ and $A_1 \sqsubseteq_\Delta A_2$. This permits the security levels of expressions to be raised—that is, it permits them to add extra dependencies to themselves. Beyond for $[A \cdot \phi]$ the subtyping judgment is standard so we elide its definition. We also omit the operational semantics, which are as usual but for the syntax $\#e$ and $!e$ for $[A \cdot \phi]$. These act respectively as thunking and forcing thunks. Removing the syntactic forms for these—and therefore the thunking semantics—presents no formal obstacle. We have found it pedagogically easier to give them explicit syntax, since they can then be mentioned explicitly if needed. The thunking semantics for their syntax simply aligns with the negative polarity [Levy 1999] of the satisfaction connective, discussed in Section 3.2.1. We look now to the positives.

*5.1.1 Positive Connectives.* Our system supports two positive connectives: positive products and sums. T-InjL is one of two introduction rules for sums, the other being T-InjR. Observe that the dependencies $\phi$ of the expression $e$ are propagated straightforwardly in both rules to their conclusion. The type of the injection *not* witnessed is checked for scoping, to ensure regularity. T-Case largely works as usual, but now accounts for the *indirect flows* from the expression $e$ being branched on, adding its dependencies $\phi$ in the conclusion to those coming from $e_1$ or $e_2$. We require

$$\begin{aligned}\text{Types } A &::= \ldots \mid A_1 + A_2 \mid A_1 \otimes A_2\\ \text{Expressions } e, v &::= \ldots \mid \mathtt{l} \cdot e \mid \mathtt{r} \cdot e \mid \mathtt{case}\ e\ \{\ \mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\ \}\\ &\quad \mid \langle e_1, e_2 \rangle \mid \mathtt{split}\ e_1\ \mathtt{into}\ \langle x_1, x_2 \rangle\ \mathtt{in}\ e_2\end{aligned}$$

T-InjL
$$\frac{\Delta; \Gamma \vdash e : A_1 \mid \phi \qquad \Delta \vdash A_2}{\Delta; \Gamma \vdash \mathtt{l} \cdot e : A_1 + A_2 \mid \phi}$$

T-Pair
$$\frac{\Delta; \Gamma \vdash e_1 : A_1 \mid \phi_1 \qquad \Delta; \Gamma \vdash e_2 : A_2 \mid \phi_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : A_1 \otimes A_2 \mid \phi_1 \sqcup \phi_2}$$

T-InjR
$$\frac{\Delta; \Gamma \vdash e : A_2 \mid \phi \qquad \Delta \vdash A_1}{\Delta; \Gamma \vdash \mathtt{r} \cdot e : A_1 + A_2 \mid \phi}$$

T-Split
$$\frac{\Delta; \Gamma \vdash e : A_1 \otimes A_2 \mid \phi \qquad \Delta; \Gamma, x_1 : A_1, x_2 : A_2 \vdash e_1 : A \mid \phi'}{\Delta; \Gamma \vdash \mathtt{split}\ e\ \mathtt{into}\ \langle x_1, x_2 \rangle\ \mathtt{in}\ e_1 : A \mid \phi \sqcup \phi'}$$

T-Case
$$\frac{\Delta; \Gamma \vdash e : A_1 + A_2 \mid \phi \qquad \Delta; \Gamma, x_1 : A_1 \vdash e_1 : A \mid \phi' \qquad \Delta; \Gamma, x_2 : A_2 \vdash e_2 : A \mid \phi'}{\Delta; \Gamma \vdash \mathtt{case}\ e\ \{\ \mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\ \} : A \mid \phi \sqcup \phi'}$$

Fig. 15. Positive Connectives for the Structural Calculus of Indistinguishability

$e_1$ and $e_2$ to have the same dependency level. Introducing positive products works similarly with respect to the flow of dependencies, with the introduction rule T-Pair propagating the dependencies $\phi_1, \phi_2$ from each of its subexpressions to the conclusion of the rule at $\phi_1 \sqcup \phi_2$. T-Split works the same way as T-Case from an information flow perspective.

We see that the introduction rules for **positives act transparently with respect to the judgemental dependencies** $\phi$, instead of being forced to encapsulate them and conclude at the empty set. An alternative formulation of the introduction rule for products forces $e_1$ and $e_2$ to consume their dependencies into their types; this would then be a negative product. Stemming from the choice to have a separate type connective $[A \cdot \phi]$ for tagging types with dependencies—rather than tagging each type with dependency information individually—the design of such a rule is predetermined by polarity. Observe that negative (or *lazy*) products would track dependencies more granularly than positive products, as expected from Section 3.2.1: each projection's dependencies can be distinguished from the other's. Not so for positive products, which blend both projections' dependencies $\phi_1$ and $\phi_2$ together. Positive products can be viewed as more general than negative products in our setting, since we can simply use satisfaction for one or both elements of the pair. As a rule, introduction forms for positive types will be transparent to dependency information, while those for negative types will be opaque. This is in line with the interpretation of the former as connected to values, and the latter to computations [Levy 1999].

## 5.2 Non-interference

We prove non-interference via a binary logical relation. We show that it satisfies a number of desirable properties. We then show the fundamental theorem, which relates well-typed programs to membership in the logical relation. Non-interference is captured with a corollary stating that any function whose argument dependencies are not represented in its return dependencies must be constant in its argument, fufilling our promise from Section 1.1. The proof treats declassification without additional machinery beyond that for handling quantifiers. The definition of the logical

$$
\begin{aligned}
e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \phi_1 \not\sqsubseteq_\Delta \phi_2 \text{ or } v \text{ val}, v' \text{ val}, e \longmapsto^* v, e' \longmapsto^* v', \\
&\quad v \sim v' \in A \mid \phi_1\ [\phi_2]\ [\Delta] \\
\langle\rangle \sim \langle\rangle \in \mathtt{unit} \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \text{true} \\
\mathtt{l} \cdot v \sim \mathtt{l} \cdot v' \in A_1 + A_2 \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq v \overset{*}{\sim} v' \in A_1 \mid \phi_1\ [\phi_2]\ [\Delta] \\
\mathtt{r} \cdot v \sim \mathtt{r} \cdot v' \in A_1 + A_2 \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq v \overset{*}{\sim} v' \in A_2 \mid \phi_1\ [\phi_2]\ [\Delta] \\
\langle v_1, v_2\rangle \sim \langle v_1', v_2'\rangle \in A_1 \otimes A_2 \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq v_1 \overset{*}{\sim} v_1' \in A_1 \mid \phi_1\ [\phi_2]\ [\Delta], \\
&\quad v_2 \overset{*}{\sim} v_2' \in A_2 \mid \phi_1\ [\phi_2]\ [\Delta] \\
v \sim v' \in [A \cdot \phi] \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq\ !v \overset{*}{\sim}\ !v' \in A \mid \phi \sqcup \phi_1\ [\phi_2]\ [\Delta] \\
v \sim v' \in A_1 \to A_2 \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \forall \phi_2' \sqsubseteq_\Delta \phi_2,\ \Delta \vdash \phi_1',\ v_1 \text{ val}, v_1' \text{ val}\ . \\
&\quad v_1 \overset{*}{\sim} v_1' \in A_1 \mid \phi_1'\ [\phi_2']\ [\Delta] \implies \\
&\quad \mathsf{ap}(v; v_1) \overset{*}{\sim} \mathsf{ap}(v'; v_1') \in A_2 \mid \phi_1' \sqcup \phi_1\ [\phi_2']\ [\Delta] \\
v \sim v' \in \forall(\alpha.A) \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \Delta \vdash \phi \implies v[\phi] \overset{*}{\sim} v'[\phi] \in [\phi/\alpha]A \mid \phi_1\ [\phi_2]\ [\Delta]
\end{aligned}
$$

Fig. 16. Semantic Equality for the Structural Calculus of Indistinguishability

relation is given in Figure 16. Full proofs of all theorems referenced in this section can be found in the accompanying artifact [Gouni et al. 2025, Appendix C].

The starred relation $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ can be read as "$e$ relates to $e'$ at type $A$ with security level $\phi_1$ and observer level $\phi_2$ under in-scope dependency variables $\Delta$." Here $e, e'$ are closed expressions, containing no variables. We introduce a notion of an *observer level* [Kozyri et al. 2022] which determines whether an "observer" of a program who is permitted to see certain dependencies should be allowed to see the outputs of the program in question. If the security level—which plays the same role as the $\phi$ in the typing judgment—is a subset of the observer level, then the answer is yes. Otherwise, the answer is no. The $\phi_1 \not\sqsubseteq_\Delta \phi_2$ in the definition of the starred relation codifies this, and is called the *non-interference condition*. If the security level $\phi_1$ of some related expressions is *not* a subset of the current observer level $\phi_2$, then to that observer, the expressions are equal. From their perspective, no discriminating information can be gleaned. This is why non-interference is a *hyperproperty* [McLean 1996], or inherently a matter of two or more related traces of evaluation: it reasons about the observable differences between them.

If the non-interference condition in $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ is not triggered then the equality must ultimately be established according to the type A. First, $e$ and $e'$ must evaluate to values $v$ val and $v'$ val. And $v, v'$ must be related at $v \sim v' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$, the non-starred relation. The definition of $e \overset{*}{\sim} e'$ by evaluation immediately gives us the following two properties.

**Lemma 5.2** (Closed $\to$). *If $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ and $e \longmapsto^* e_1$ then $e_1 \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$.*

Proof. By use of evaluation in $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ and determinicity of evaluation. □

**Lemma 5.3** (Closed $\leftarrow$). *If $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ and $e_1 \longmapsto^* e$ then $e_1 \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$.*

Proof. By use of evaluation in $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ and transitivity of evaluation. □

These two lemmas show that the logical relation is preserved by evaluation *in both directions*. Lemma 5.3 is the critical one: if expressions $e, e'$ are logically related, then *anything that evaluates*

*to them* is also in the logical relation. This means that there are many expressions which are related by the logical relation, but are not well-typed.

The relation $v \sim v'$ is mutually recursive with the starred relation and is defined inductively on types in a standard way. For positive types it ensures that both sides have the expected canonical forms, and that the insides of the canonical forms are related at the appropriate type. For negative types it ensures they behave correctly under elimination. Within each case it recurses back onto $e \overset{*}{\sim} e'$ to check the equality of the inner or eliminated expressions. This is important when the security level is raised as in the cases for $[A \cdot \phi]$ and $A_1 \to A_2$, because a higher security level, or larger set, may satisfy the non-interference condition. Note that while the security level $\phi_1$ is *monotonic* with respect to non-interference, the observer level $\phi_2$ is *anti-monotonic*.

**Lemma 5.4** (Monotone). *If* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ *and* $\phi_1 \sqsubseteq_\Delta \phi_1'$ *then* $e \overset{*}{\sim} e' \in A \mid \phi_1'\ [\phi_2]\ [\Delta]$.

Proof. By straightforward induction on $A$. □

**Lemma 5.5** (Anti-monotone). *If* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ *and* $\phi_2' \sqsubseteq_\Delta \phi_2$ *then* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2']\ [\Delta]$.

Proof. By straightforward induction on $A$. □

Intuitively, $\phi_1 \not\sqsubseteq_\Delta \phi_2$ is preserved by either adding elements to the left side, or removing elements from the right. The former raises the security level, and the latter means the observer drops permissions for viewing certain dependencies. That the dependency elision heuristic from Section 3.2.2 preserves non-interference, and is therefore sound, can be justified via monotonicity: it always produces equally as many or more dependencies into the ambient security level than the non-elided variant, because argument dependencies get propagated ambiently to the application expression. As mentioned, we leave the elision algorithm and its completeness for future work. We next show that our logical relation satisfies certain properties of equivalence relations.

**Lemma 5.6** (Symmetry). *If* $e_1 \overset{*}{\sim} e_2 \in A \mid \phi\ [\phi']\ [\Delta]$ *then* $e_2 \overset{*}{\sim} e_1 \in A \mid \phi\ [\phi']\ [\Delta]$.

Proof. By straightforward induction on $A$. □

**Lemma 5.7** (Transitivity). *If* $e_1 \overset{*}{\sim} e_2 \in A \mid \phi\ [\phi']\ [\Delta]$ *and* $e_2 \overset{*}{\sim} e_3 \in A \mid \phi\ [\phi']\ [\Delta]$ *then* $e_1 \overset{*}{\sim} e_3 \in A \mid \phi\ [\phi']\ [\Delta]$

Proof. By induction on $A$ and Lemma 5.6 in the $A_1 \to A_2$ case to handle contravariance. □

Importantly, the logical relation *does not* satisfy reflexivity. For instance, the expression $\langle\rangle$ cannot be self-equated at any type other than unit, and in general related expressions must behave appropriately for their declared type. Thus our logical relation is a *partial equivalence relation*, satisfying symmetry and transitivity but not reflexivity. Finally, the fundamental theorem translates well-typedness to membership in the logical relation. However, recall that while the typing rules work with open expressions, the logical relation only works on closed expressions. We must generalize the logical relation to account for open expressions. We begin by defining *closing substitutions* which replace free dependency variables and term variables with appropriate forms.

(1) Define $\delta \in \Delta \rightsquigarrow \Delta'$ as a map from each $\alpha \in \Delta$ to a dependency environment $\Delta' \vdash \phi$.
(2) Define:
   (a) $\gamma \in \Gamma\ [\Delta']$ as a map from $x \in \Gamma$ to an expression $e$ closed under term variables s.t. $\Delta' \vdash e$
   (b) $\gamma \overset{*}{\sim} \gamma' \in \Gamma \mid \phi\ [\phi']\ [\Delta \rightsquigarrow \Delta']$ to mean that if $\Delta' \vdash \phi'$ and $\delta \in \Delta \rightsquigarrow \Delta'$ then we have $\gamma, \gamma' \in \Gamma\ [\Delta']$ s.t. $\gamma(x) \overset{*}{\sim} \gamma'(x) \in \widehat{\delta}(A) \mid \phi\ [\phi']\ [\Delta']$ for all $x : A \in \Gamma$
(3) Define $\Delta\ \Gamma \gg^{\phi'}_{\Delta'} e \overset{*}{\sim} e' \in A \mid \phi$ to mean that for all $\Delta' \vdash \phi_\gamma$ if 3a, 3b, and 3c then 3d.
   (a) $\Delta' \vdash \phi'$

(b) $\delta \in \Delta \rightsquigarrow \Delta'$
(c) $\gamma \overset{*}{\sim} \gamma' \in \Gamma \mid \phi_\gamma\ [\phi']\ [\Delta \rightsquigarrow \Delta']$ (instantiated to 3a and 3b)
(d) $\widehat{\delta}(\widehat{\gamma}(e)) \overset{*}{\sim} \widehat{\delta}(\widehat{\gamma'}(e')) \in \widehat{\delta}(A) \mid \widehat{\delta}(\phi) \sqcup \phi_\gamma\ [\phi']\ [\Delta']$

$\delta$ replaces dependency variables $\alpha$ with dependency sets $\phi$ closed under $\Delta'$. $\gamma, \gamma'$ replace term variables $x$ with closed expressions related at the type of the variable. The type itself is closed using $\delta$. The security level $\phi$ represents the cumulative dependencies of all information contained in $\Gamma$. $\widehat{\delta}$ and $\widehat{\gamma}$ apply the mappings in $\delta$ and $\gamma$ *simultaneously* to all open variables in their arguments. The generalized logical relation in item 3 is defined by using $\widehat{\delta}$ and $\widehat{\gamma}$ on its expressions $e, e'$ and $\widehat{\delta}$ on its type $A$ and security level $\phi$. We invoke the starred relation on the substituted forms.

**Theorem 5.8** (Fundamental Theorem)**.** *If* $\Delta_0, \Delta; \Gamma \vdash e : A \mid \phi$ *then* $\Delta\ \Gamma \gg^{\phi'}_{\Delta_0} e \overset{*}{\sim} e \in A \mid \phi$.

Proof. By induction on a derivation of $\Delta_0, \Delta; \Gamma \vdash e : A \mid \phi$. □

The statement of the fundamental theorem splits up the information flow variable environment from the typing judgment into $\Delta_0$ and $\Delta$. $\Delta_0$ denotes *the observer's dependency environment*. The definition of equality of open expressions requires that $\Delta_0 \vdash \phi'$ where $\phi'$ is the observer level. $\Delta_0$ provides an environment for a closing substitution on dependency variables $\alpha$ to be *closed under*, because all dependencies considered in the logical relation must be meaningful to the observer. That the open and closed logical relations are *relativized* to a base $\Delta_0$ representing the observer's environment is key to the formal treatment of declassification. The constant function property we desired in Section 1.1 emerges as a straightforward corollary of the fundamental theorem.

**Corollary 5.9** (Constant Function)**.** *If we have* $\Delta; \Gamma \vdash e : [A_1 \cdot \phi_1] \to [A_2 \cdot \phi_2] \mid \circ$ *and* $c$ val *and* $\phi_1 \not\sqsubseteq_\Delta \phi_2$ *then* $\circ\ \Gamma \gg^{\phi_2}_{\Delta} e \overset{*}{\sim} \lambda(x.\mathsf{ap}(e; c)) \in [A_1 \cdot \phi_1] \to [A_2 \cdot \phi_2] \mid \circ$.

Proof. Follows directly from Theorem 5.8. We sketch the proof here.

(1) Assume $\phi_\gamma$ and $\Delta \vdash \phi_2$ and appropriate $\delta$ and $\gamma, \gamma'$.
(2) To show $[A_1 \cdot \phi_1] \to [A_2 \cdot \phi_2]$ assume $v_1 \sim v_1' \in [A_1 \cdot \phi_1] \mid \phi_1'\ [\phi_2']\ [\Delta]$ where $\phi_2' \sqsubseteq_\Delta \phi_2$.
(3) Suffices to show $\mathsf{ap}(\widehat{\gamma}(e); v_1) \overset{*}{\sim} \mathsf{ap}(\lambda(x.\mathsf{ap}(\widehat{\gamma'}(e); c)); v_1') \in [A_2 \cdot \phi_2] \mid \phi_1' \sqcup \phi_\gamma\ [\phi_2']\ [\Delta]$.
(4) We have $\phi_1 \not\sqsubseteq_\Delta \phi_2'$ by properties of set inclusion.
(5) Obtain $!v_1 \overset{*}{\sim} !c \in A_1 \mid \phi_1\ [\phi_2']\ [\Delta]$ by non-interference, so $v_1 \overset{*}{\sim} c \in [A_1 \cdot \phi_1] \mid \circ\ [\phi_2']\ [\Delta]$.
(6) By Lemma 5.4 we have $v_1 \overset{*}{\sim} c \in [A_1 \cdot \phi_1] \mid \phi_1'\ [\phi_2']\ [\Delta]$.
(7) Use Theorem 5.8 on the typing assumption for $e$ and instantiate with item 1 to get $\widehat{\gamma}(e) \overset{*}{\sim} \widehat{\gamma'}(e) \in [A_1 \cdot \phi_1] \to [A_2 \cdot \phi_2] \mid \phi_\gamma\ [\phi_2]\ [\Delta]$.
(8) If $\phi_\gamma \not\sqsubseteq_\Delta \phi_2$ then we have $\phi_\gamma \not\sqsubseteq_\Delta \phi_2'$ and the goal is immediate. Otherwise:
  (a) $\widehat{\gamma}(e) \longmapsto^* v_2, \widehat{\gamma'}(e) \longmapsto^* v_2', v_2$ val, $v_2'$ val
  (b) $v_2 \sim v_2' \in [A_1 \cdot \phi_1] \to [A_2 \cdot \phi_2] \mid \phi_\gamma\ [\phi_2]\ [\Delta]$
(9) Apply item 8b to item 6 to obtain $\mathsf{ap}(v_2; v_1) \overset{*}{\sim} \mathsf{ap}(v_2'; c) \in [A_2 \cdot \phi_2] \mid \phi_1' \sqcup \phi_\gamma\ [\phi_2']\ [\Delta]$.
(10) Have $\mathsf{ap}(\widehat{\gamma}(e); v_1) \longmapsto^* \mathsf{ap}(v_2; v_1)$, $\mathsf{ap}(\lambda(x.\mathsf{ap}(\widehat{\gamma'}(e); c)); v_1') \longmapsto^* \mathsf{ap}(\widehat{\gamma'}(e); c) \longmapsto^* \mathsf{ap}(v_2'; c)$.
(11) The result follows by applying Lemma 5.3 twice to item 9 and each evaluation in item 10. □

The constant function property states that a function whose argument dependencies are not a subset of those in its return value is observationally the constant function. Particularly, such a function is observationally equivalent to a function which has its argument stubbed out with some constant $c$. From the perspective of the observer both the original function and that which ignores its argument behave the same way. Observe that the final example from Section 1.1 falls under the constant function theorem because $\circ; \alpha \not\sqsubseteq_\Delta \circ$. For completeness' sake, we have $\mathtt{unit} + \mathtt{unit} \cong \mathsf{bool}$.

Note that the correspondence of our logical relation with observational equivalence is apparent from the fact that it does not introspect on the syntax of expressions under evaluation and from its synchronization with polarity. We (1) do not examine the structure of negatively typed computations, only observing their behavior under elimination, and (2) explicitly examine the canonical forms of positive values. This is all done modulo an observer level, which can be seen as internalizing observational equivalence. We have not yet made a point of declassification; we do so now.

### 5.3 Metatheoretic Mechanics of Declassification

Note that Corollary 5.9 only forces the function to be constant *from the observer's perspective*. Assume we have $\phi_1 \not\sqsubseteq \phi_2$ as before, observing at $\phi_2$. We might implement a function typed at $[A_1 \cdot \phi_1] \to [[A_1 \cdot \phi_1] \cdot \phi_2]$ as $\lambda(x.\#x)$. However, eliminating the inner satisfaction at $\phi_1$ immediately satisfies the non-interference condition and therefore the logical relation. The observer level $\phi_2$ denotes our *perspective*: it says we cannot observe information at $\phi_1$ so we may trivially equate programs at level $\phi_1$. That equality is subject to observability is central to non-interference.

Accounting for declassification requires us to generalize this idea. Instead of conditioning equality merely on *which dependencies can be observed*, it is additionally conditioned on *which dependencies were available to observe*. This is the role of the $\Delta$ which indexes the logical relation in Figure 16, and accordingly $\Delta_0$ in Theorem 5.8. All dependencies which appear in the closed logical relation must be scoped under $\Delta$. The closing map $\delta$ and the logical relation's handling of the quantifier type $\forall(\alpha.A)$ are the foremost machinery which act to ensure this. As a consequence of fixing the scope of dependency variables, we can choose whether to reason about existential dependencies according to whether we wish to reason about declassification. We can explicate this in terms of the schema for existential quantification from Figure 12.

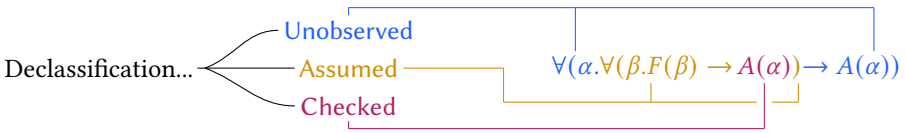



Fig. 17. Reasoning (or not) about declassification

Figure 17 shows the schema, annotated. We write $A(\alpha)$ to mean that the type $A$ may contain $\alpha$. Recall that $\beta$ is an existential dependency, and that $\alpha$ cannot mention it. $F(\beta)$ is an interface offering declassifying functions like `check` and `hash`, as before. The schema for existential quantification gives rise to two perspectives: (1) where any declassifying behavior is completely unobserved because $\beta$ is not in scope, and (2) where the functions in $F(\beta)$ are assumed to conserve equalities, and the use of this interface to produce a type $A(\alpha)$ is checked to be in the logical relation.

Starting with the first perspective, we imagine observing a program from outside the existential schema, without $\beta$ in scope. This corresponds to working with an instance of the closed logical relation at some $\Delta$ which does not include $\beta$. We call this 'unobserved' because $\beta$ will never arise in any form while reasoning with the logical relation; it can never be used to trigger non-interference. That any declassification is happening is entirely invisible from this perspective. We can demonstrate this by sketching using the logical relation to show equality for some arbitrary program at the type in Figure 17, and observing the goal it reduces to.

(1) We want to show $\ldots \in \forall(\alpha.\forall(\beta.F(\beta) \to A(\alpha)) \to A(\alpha)) \mid \ldots [\Delta]$.
(2) Assume $\Delta \vdash \phi$ and substitute it for $\alpha$ to get $\ldots \in \forall(\beta.F(\beta) \to A(\phi)) \to A(\phi) \mid \ldots [\Delta]$.
(3) Assume $\ldots \in \forall(\beta.F(\beta) \to A(\phi)) \mid \ldots [\Delta]$.
(4) Suffices to show $\ldots \in A(\phi) \mid \ldots [\Delta]$.

Observe that we never introduce $\beta$ in this process, and $A(\phi)$ does not contain $\beta$. The critical step is in item 2, where we instantiate $\alpha$—and therefore the output type $A(\alpha)$ of the program—with a $\Delta \vdash \phi$ where $\beta \notin \Delta$. So reasoning about the behavior of some program which declassifies, if our perspective is from *outside* the existential, does not permit us to observe the effect of any declassifications. Put another way, declassification in our setting is *modular*. Client code downstream of some program using declassification, but which does not expose the associated existential dependency variables in its types, must reason about the program without regard to any declassifying behavior. But what if existential variables *do* appear in types?

We move now to the second perspective, which regards the inside of the existential—specifically the higher-order function where $\beta$ is in scope. Accordingly, $\beta$ can now occur within the logical relation, with the latter being indexed at $\Delta, \beta$. This presents a problem. To illustrate, imagine our observer level does not include $\beta$ and that there is a function at type $[\mathsf{string} \cdot \beta] \to \mathsf{string}$ in $F(\beta)$, akin to hash from Figure 12. We want to use the logical relation to see what happens when it is applied. We can use non-interference to obtain related expressions at $[\mathsf{string} \cdot \beta]$ and pass them to the function. Then the application forms are equated at string, but not by non-interference. Note the *equated* outputs of the function depend on its *non-equated* inputs! This disconnect can be explained by walking through the inner part of the existential via the logical relation.

(1) We want to show $\ldots \in F(\beta) \to A(\phi) \mid \ldots [\Delta, \beta]$.
(2) Assume $\ldots \in F(\beta) \mid \ldots [\Delta, \beta]$.
(3) Suffices to show $\ldots \in A(\phi) \mid \ldots [\Delta, \beta]$.

The critical step is item 2, where we *assume* our declassifying functions to be in the logical relation, and therefore to preserve equalities. We then must *show* in item 3 that the computation that uses them itself establishes an equality. Importantly, the act of assuming declassifiers preserve equalities, or *ignoring* any disequalities induced—due to observable results depending on non-observable inputs—privileges them. The logical relation will only permit these assumed declassifiers to perform declassification in item 3. Membership in the logical relation will not be able to be shown for expressions which attempt a non-permitted (non-assumed) declassification because a disequality will result. So it is checked that any declassification which occurs is as a consequence of operations from $F(\beta)$.

## 6 Related and Future Work

### 6.1 Declassification via Type Abstraction

The first paper to recognize the relationship between declassification and type abstraction was Nanevski et al. [2013], working in a verification logic embedded in a dependent type theory. Unlike in our setting, their language works directly in terms of abstract types exporting equality predicates for non-interference reasoning. Due to being coalesced with functional correctness reasoning, their specifications are quite complex. We target a type system intended to be used outside a verification setting, in general-purpose languages. Frumin et al. [2021] also use a relational logic, integrated with a simpler type system, to reason about declassification. Due to the expressivess of the logic, their ability to verify the safety of declassification according to e.g. the concrete values taken on by a variable surpasses ours. We suspect extending our system to account for type-level value dependency may permit a similar degree of expressivity.

Ngo et al. [2020] recovers noninterference with declassification via existential quantification over types. Such quantification, however, comes with the issues noted in Section 1.1: interesting computation cannot be done on abstract types, so their approach does not permit computing with secrets until they have been declassified. A follow-up paper [Cruz and Tanter 2019] approaches from a similar angle, again existentially quantiying over types. They adapt *faceted types* from

the information flow setting to make computations on secrets possible. The approach presented there is attractive, but combining quantification over dependencies and free semilattices seems to accomplish the same goals more directly and with better-understood logical foundations.

### 6.2 Foundations for Information Flow

Miyamoto and Igarashi [2004] discuss modal logic as the logical basis for information flow, working in a partial modal logic setting. While they do not strictly make the connection to partial modal logic or hybrid logic, it is observed that their information flow tracking connective may decompose as $@_\ell \Box A$. They do not make the further step to lax logic to notice that the necessity semantics is vestigial. Many of the hybrid intuitions we relied on here emerged later from Reed [2009].

Other work [Askarov et al. 2008; Halpern and O'Neill 2008] grounds information flow in epistemic logic, a flavor of modal logic which contends naturally with principals in information flow systems such as Jif [Myers 1999]. Like Jif's model, our approach is decentralized [Myers and Liskov 2000] in that it is not based on a single trusted principal or a fixed lattice structure. Our approach differs in that our types make no statements about policy or the allowed readers and writers of data governed by such policy; this allows us to focus exclusively on information flow itself, simplifying our system. Future work could explore whether the mechanisms in Jif could be built on this foundation, and whether the intuitions from our setting might transfer to a principal-based approach.

Sterling and Harper [2022] establish a sheaf model for non-interference, using the topos-theoretic *sealing* and *transparency* modalities to selectively obscure information. We suspect that our satisfaction connective $[A \cdot \phi]$ is related to the transparency modality, exhibiting similar behavior and being of the same polarity. We are actively investigating the categorical semantics of our language and expect to shine further light on any connections here.

Finally, a fragment of the structural approach to information flow appears to be related to Algebraic Subtyping [Dolan 2017; Parreaux 2020]. In particular, one might imagine encapsulating every type `T` in a wrapper type `IFC[T, I]` playing a similar role to the satisfaction connective. Information flow dependencies can then be expressed as unions of types representing dependencies, for instance `IFC[int, PWD | 'a | 'b]`. We leave a full development of this idea to future work; this may serve as a lightweight way to port our style of information flow reasoning to existing languages. It is unclear to what extent current Algebraic Subtyping systems provide support for higher-ranked quantification, though there is ongoing work [Parreaux et al. 2024] to do so.

## 7 Conclusion

We have provided here the *Structural Calculus of Indistinguishability*. We have described a logically motivated approach to information flow which simultaneously unlocks interesting opportunities to simplify information flow specifications and offers a modular, sound approach to declassification. We have shown that the latter captures useful programming patterns from the literature and that the treatment of non-interference for it can reuse, unchanged, the machinery for hybrid-style quantification over worlds.

## 8 Data Availability Statement

Referenced appendices, auxiliary definitions, and full proofs are available at Gouni et al. [2025].

## Acknowledgements

We thank the anonymous reviewers for their helpful feedback; any remaining oversights or errors are ours. We would also like to thank Corinthia Aberlé, Harrison Grodin, Lionel Parreaux, and Tesla Zhang for insightful discussions. This research was supported by the Department of Defense and the National Science Foundation under Grant Nos. H98230-23-C-0275 and CCF-1901033.

## References


Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. "A core calculus of dependency." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '99). Association for Computing Machinery, San Antonio, Texas, USA, 147–160. ISBN: 1581130953. doi:10.1145/292540.292555.

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. "Termination-insensitive noninterference leaks more than just a bit." In: *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13*. Springer, 333–348. doi:10.1007/978-3-540-88313-5_22.

Peter Nicholas Benton, Gavin M. Bierman, and Valeria Correa Vaz de Paiva. 1998. "Computational types from a logical perspective." *Journal of Functional Programming*, 8, 2, 177–193. doi:10.1017/S0956796898002998.

Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. "A Dependent Dependency Calculus." In: *European Symposium on Programming*. Springer International Publishing Cham, 403–430. doi:10.1007/978-3-030-99336-8_15.

Raimil Cruz and Éric Tanter. 2019. "Existential Types for Relaxed Noninterference." en. In: *Programming Languages and Systems*. Ed. by Anthony Widjaja Lin. Springer International Publishing, Cham, 73–92. ISBN: 9783030341756. doi:10.1007/978-3-030-34175-6_5.

Dorothy E Denning. 1976. "A lattice model of secure information flow." *Communications of the ACM*, 19, 5, 236–243. doi:10.1145/360051.360056.

Stephen Dolan. 2017. *Algebraic subtyping*. BCS, The Chartered Institute for IT. ISBN: 9781780174150.

Matt Fairtlough and Michael Mendler. 1997. "Propositional lax logic." *Information and Computation*, 137, 1, 1–33. doi:10.1006/inco.1997.2627.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. "Compositional Non-Interference for Fine-Grained Concurrent Programs." In: *2021 IEEE Symposium on Security and Privacy (SP)*, 1416–1433. doi:10.1109/SP40001.2021.00003.

Jean-Yves Girard. 1986. "The system F of variable types, fifteen years later." *Theoretical computer science*, 45, 159–192. doi:10.1016/0304-3975(86)90044-7.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7. Cambridge university press Cambridge. ISBN: 9780521371810.

Joseph A Goguen and José Meseguer. 1982. "Security policies and security models." In: *1982 IEEE Symposium on Security and Privacy*. IEEE, 11–11. doi:10.1109/SP.1982.10014.

Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2025. *Appendices, Definitions, and Proofs for Article 'Structural Information Flow: A Fresh Look at Types for Non-interference'*. Zenodo. (2025). doi:10.5281/zenodo.17013074.

Joseph Y Halpern and Kevin R O'Neill. 2008. "Secrecy in multiagent systems." *ACM Transactions on Information and System Security (TISSEC)*, 12, 1, 1–47. doi:10.1145/1410234.1410239.

Shin-ya Katsumata. 2014. "Parametric effect monads and semantics of effect systems." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '14). Association for Computing Machinery, San Diego, California, USA, 633–645. ISBN: 9781450325448. doi:10.1145/2535838.2535846.

Elisavet Kozyri, Stephen Chong, Andrew C Myers, et al.. 2022. "Expressing information flow properties." *Foundations and Trends® in Privacy and Security*, 3, 1, 1–102. doi:10.1561/3300000008.

Paul Blain Levy. 1999. "Call-by-push-value: A subsuming paradigm." In: *International Conference on Typed Lambda Calculi and Applications*. Springer, 228–243. doi:10.1007/3-540-48959-2_17.

Yiyun Liu, Jonathan Chan, Jessica Shi, and Stephanie Weirich. 2024. "Internalizing Indistinguishability with Dependent Types." *Proceedings of the ACM on Programming Languages*, 8, POPL, 1298–1325. doi:10.1145/3632886.

John McLean. 1996. "A general theory of composition for a class of" possibilistic" properties." *IEEE Transactions on Software Engineering*, 22, 1, 53–67. doi:10.1109/32.481534.

John C. Mitchell and Gordon D. Plotkin. 1985. "Abstract types have existential types." In: *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages - POPL '85*. Association for Computing Machinery, 37–51. doi:10.1145/318593.318606.

Kenji Miyamoto and Atsushi Igarashi. 2004. "A modal foundation for secure information flow." In: *Workshop on Foundations of Computer Security*, 187–203.

E. Moggi. 1989. "Computational lambda-calculus and monads." In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, 14–23. doi:10.1109/LICS.1989.39155.

Andrew C Myers. 1999. "Mostly-static decentralized information flow control." Ph.D. Dissertation. Massachusetts Institute of Technology. doi:1721.1/16717.

Andrew C Myers and Barbara Liskov. 2000. "Protecting privacy using the decentralized label model." *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9, 4, 410–442. doi:10.1145/363516.363526.

Aleksandar Nanevski. 2004. *Functional programming with names and necessity*. Carnegie Mellon University.

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. "Dependent type theory for verification of information flow and access control policies." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35, 2, 1–41. doi:10.1145/2491522.2491523.

Minh Ngo, David A Naumann, and Tamara Rezk. 2020. "Type-Based Declassification for Free." In: *Formal Methods and Software Engineering: 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1–3, 2021, Proceedings 22*. Springer, 181–197. doi:10.1007/978-3-030-63406-3_11.

Lionel Parreaux. Aug. 2020. "The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)." *Proc. ACM Program. Lang.*, 4, ICFP, Article 124, (Aug. 2020), 28 pages. doi:10.1145/3409006.

Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. "When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism." *Proceedings of the ACM on Programming Languages*, 8, POPL, 1418–1450.

Frank Pfenning and Rowan Davies. 2001. "A judgmental reconstruction of modal logic." *Mathematical structures in computer science*, 11, 4, 511–540. doi:10.1017/S0960129501003322.

François Pottier and Vincent Simonet. 2002. "Information flow inference for ML." In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '02). Association for Computing Machinery, Portland, Oregon, 319–330. ISBN: 1581134509. doi:10.1145/503272.503302.

A. N. Prior. 1968. ""Now"." *Noûs*, 2, 2, 101–119. doi:10.2307/2214699.

Vineet Rajani and Deepak Garg. 2018. "Types for Information Flow Control: Labeling Granularity and Semantic Models." In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 233–246. doi:10.1109/CSF.2018.00024.

Jason Reed. 2009. *A hybrid logical framework*. Carnegie Mellon University.

John C. Reynolds. 1984. "Types, Abstraction, and Parametric Polymorphism." In: *Information Processing 83: Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. A Mason. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 513–523.

Andrei Sabelfeld and David Sands. 2009. "Declassification: Dimensions and principles." *Journal of Computer Security*, 17, 5, 517–548. doi:10.3233/JCS-2009-0352.

Naokata Shikuma and Atsushi Igarashi. 2008. "Proving noninterference by a fully complete translation to the simply typed lambda-calculus." *Logical Methods in Computer Science*, 4. doi:10.1007/978-3-540-77505-8_24.

Jonathan Sterling and Robert Harper. 2022. "Sheaf Semantics of Termination-Insensitive Noninterference." In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)* (Leibniz International Proceedings in Informatics (LIPIcs)). Ed. by Amy P. Felty. Vol. 228. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:19. ISBN: 978-3-95977-233-4. doi:10.4230/LIPIcs.FSCD.2022.5.

Christopher Strachey. 2000. "Fundamental concepts in programming languages." *Higher-order and symbolic computation*, 13, 11–49. doi:10.1023/A:1010000313106.

Stephen Tse and Steve Zdancewic. 2004. "Translating dependency into parametricity." In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (ICFP '04). Association for Computing Machinery, Snow Bird, UT, USA, 115–125. ISBN: 1581139055. doi:10.1145/1016850.1016868.

Received 2025-03-25; accepted 2025-08-12