Structural Information Flow

A Fresh Look at Types for Non-Interference

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

ACM Reference Format:
Hemant Sai Gouni, Jonathan Aldrich, and Frank Pfenning. 2025. Structural Information Flow: A Fresh Look at Types for Non-Interference. *Proc. ACM Program. Lang.* 1, 1 (June 2025), 65 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 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

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

2025. ACM 2475-1421/2025/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 battle-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 engage in impressive feats of information flow reasoning—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 type `id : α -> α` specifies that the return value must depend only on the data given as input. Likewise, the type `second : α -> β -> β` expresses that its return value must depend only on its second argument. For a more interesting example, consider the type of the standard `map` function on lists given in Figure 1. The polymorphic components here are the 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.

```
let map : (α -> β) -> list α -> list β
```

Fig. 1. Standard type for map

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 : α -> α` expresses that the function should be able to be given an input at any type, and return data at that same type. This combination is powerful, bestowing a property called *parametricity* which tells us 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 don't 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 α because it is not known to be a number. Our first core intuition is that **parametric polymorphism plays two distinct roles**, both **(1) providing data abstraction** and **(2) enforcing information flow properties**. We want to isolate the second, so let's try tagging types with *information flow variables* like α, rather than having α *be* the type. Under this proposal, we might type the successor function on integers as α `int -> α int`. The α 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 α in `list α`, which permits the list to be generic over the type of its contents.

We're better off now than we were before—we can do interesting computation with data whose information flow content we're tracking—but we're not quite there yet. For instance, try to write 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 information flow variables to being **sets of variables**. This is the other core intuition

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

behind our system. Let's set $\boxed{?}$ = [α β] **int**, which can be read as 'this **int** depends on data from sources α and β.' The full type of addition is then as given in Figure 2.

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

Fig. 2. Information Flow Types for Addition

The information flow variables on the arguments have also been turned into (here, singleton) 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*. That is, just as the α in α -> α can be instantiated to **int** to obtain **int** -> **int**, the α in add can be instantiated to [secret1 secret2], representing data depending on some secrets. This produces add1 in Figure 3, whose first argument expects an integer dependent on concrete sources secret1 and secret2 and whose return value depends on secret1, secret2, and β.

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

Fig. 3. Addition, After Instantiating

So far, we have been skirting around performing meaningful information flow reasoning—what might a specification for preventing secret leakage look like in this setting? Let's consider a simple password checker, shown in Figure 4. We start by declaring some password data pass, which is a **string** tagged with pwd to indicate it contains password data. 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 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.

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

Fig. 4. Enterprise Grade Password Checker

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 nth 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 in Figure 5. What must $\boxed{?}$ be?

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

Fig. 5. The Constant Function

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 strict subset of the output must be the constant function. And with that, we have already arrived at our destination: 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

In Section 2 we 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 with only the winds of logical intuition gusting into one's sails. We describe how to navigate this course from the shores of 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 these insights, also introduced in the prior section. Section 4 extends this with examples involving declassification, solving our issue with `check` above. Section 5 finally discusses the formalism consisting of the typing rules and operational semantics, showing the hard-fought elegance and simplicity of the logical relation with which non-interference can be discharged in our setting. Section 6 compares to other work investigating the logical foundations of information flow, and identifies similar approaches to declassification within the literature. 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**, 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 also touch on the important role played by the proof-theoretic concept of *polarity* here.
(3) Remarkably, the quantification machinery suggested by our setting **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 quotienting out 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 stumbling across the structural approach to information flow starting from ordinary functional programming, we'll now reveal another starting from constructive modal

logic. It will turn out that *hybrid logic* [Braüner 2022], 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 exactly the considerations critical to information flow reasoning. We start by reviewing a number of programs written under a more standard theory of information flow, then reveal its logical structure in the subsections that follow.

2.1 Introduction: Round Two

Before diving in, let's 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 2003], an information flow system for OCaml.[2] Figure 6 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. 6. 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 the type for addition be handled?

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

Fig. 7. Lattice constraints enter the fray!

Here, we catch our first sighting of the *lattice constraints* which ordinarily comprise information flow specifications.[3] At the point in our prior exploration where we chose to generalize to sets of information flow variables rather than individual information flow 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 option exercised here.

A semilattice has two operators: *partial ordering*, written $\sqsubseteq$, and *join*, written $\sqcup$. $\alpha \sqsubseteq \beta$ allows us to compare two information flow variables α and β to check if data from α should be permitted to flow to β. $\alpha \sqcup \beta$ produces the 'lowest' δ (with respect to $\sqsubseteq$) such that $\alpha \sqsubseteq \delta$ and $\beta \sqsubseteq \delta$. We use both operators in the type for add, writing $\sqsubseteq$ syntactically as < and $\sqcup$ syntactically as a comma ', '. So we can parse the contents of the **with** clause as $(\alpha \sqcup \beta) \sqsubseteq \delta$, can can in turn interpret it in information flow terms as "both α and β must be able to flow to δ." In other words, the information flow variable for the return value is δ, and flows to the return value can be specified by way of partial orderings which place δ above other variables representing flow sources. When this function is called, the caller must instantiate α, β, and δ—just as we previously set α = [c d] in our typing for add1 in Figure 3—with concrete variables such that the two former variables are both ordered less than the latter one. This will satisfy the type constraints because if δ is partially ordered greater than both α

[2]Though we do not follow their metatheory because it is rife with concerns about particular features of ML, like exceptions, which are not currently relevant.

[3]The *simplification algorithms* that are usually deployed alongside such systems will attempt to prevent constraints from appearing in types. We hold off on discussing or applying these until the next section, because they would only serve to obscure the underlying mechanics.

and β, then their join is at worst equivalent to δ, if not ordered strictly less than it. As an aside and signpost, note that all this must be digested by programmers to appreciate the type of `add'`, rather than relying on existing programming intuition about parametric polymorphism as in Section 1.1. The argument for the simplicity of the latter is already taking shape.

While picking through the above definitions, you may have noticed something curious. Our sets of information flow variables also have semilattice structure! Partial ordering is given by subset inclusion, and join is given by set union. 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 underlying set—of information flow variables, in our case. We simply make each element of the variable set into a singleton set to make the first elements of our forming semilattice, and apply set union to generate more elements until we reach closure. This is analogous to taking the power set. Where before we used $\alpha_1 < \alpha_2$ to denote that data from source α_1 flowed to destination α_2, we now have `[ α β ]` to mean that data from sources `[ α ]` and `[ β ]`—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 aren't yet prepared to discuss why; that will come out later. For now, let's review a final example from the prior section under a conventional information flow system. The retyped password checker is shown in Figure 8.

```
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. 8. Comparing Our Password Checkers

The situation is much the same as in Figure 7. 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 α is accounted for similarly, as before. Now that we've got some surface-level intuition under our belts, let's see what's going on underlyingly.

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 ℓ 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 ℓ, but retains none of its original purpose within modal logic. We cannot provide the full story here—it is provided in Appendix A for the interested reader—but an alternative, cleaner logical foundations is possible.

We'll 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 partial ordering $\sqsubseteq$ on worlds ℓ, where $\ell_1 \sqsubseteq \ell_2$ means that ℓ_2 is reachable from ℓ_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 ℓ: each world-indexed necessitation $\Box_\ell$ identifies the world ℓ at whose accessible worlds it is true. In other words, in order to access some information at world ℓ, the current world ℓ' must have accounted for ℓ as a dependency—ℓ' must be reachable from ℓ 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 due to its generality—it is oblivious to the fact that it is working with world indices.

This is a job for *hybrid logic* [Braüner 2022], 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 9.

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

Fig. 9. Hybrid Logic to the Rescue!

This setup is adapted from Reed [2009]. Rather than using abstract worlds, these rules work in terms of elements of a *free commutative monoid* ϕ generated from a set of 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 ϕ into an element of a *free semilattice*. The partial ordering is given by subset inclusion 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 ℓ and ℓ' have no meaning except that provided externally, outside the syntax, but the ordering $\alpha \sqsubseteq \alpha * \beta$ is inherent to 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're not quite at a system suitable for information flow yet. We'd like to be able to state that M has dependencies ϕ at type A—something akin to $\Box_\ell A$—by saying $@_\phi A$, but it turns out this won't quite suffice. The problem is the @E rule, which wholesale replaces the current world ϕ' with the ϕ inside the satisfaction operator. For information flow, we need to keep track of the old world, as well. We musn't 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.** We update the elimination rule to @E-new, which now preserves the old world *path* ϕ' and joins it with the world *path* ϕ obtained by eliminating the satisfaction operator. Each dependency α is a world. For subtle reasons elucidated in Appendix A and by Nanevski [2004], no notion of necessity remains in this connective. Other

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

Fig. 10. Derivation for the Body of check

approaches to information flow place extra load on the necessity connective by indexing it, so cannot remove it fully. By isolating the relevant part of the typical indexed necessity connective via hybrid logic, we analyze the core machinery for information flow as a matter of *satisfaction*.

Hybrid logic has at least one more fruit to bear, but before that, we have roughly all the tools we need to work through the body of check from Figure 8, shown in Figure 10. We'll 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 ϵ 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's the final gift 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 dependency polymorphism. We have not yet written a single program that does not rely on 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 mind-bending 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. This is our third and last core insight: **dependency polymorphism is essential and must be a first-class notion**. Thankfully, hybrid logic will come to the rescue 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. We'll discuss its mechanics in the next section.

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 hinted, 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. We first get a sense of the core typing rules.

2.3 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 11. Our typing judgment is $\Delta\ \Gamma \vdash e : A \mid \phi$ and can be read "Under in-scope dependency variables Δ and in-scope term variables Γ the expression e has type A with set of

Dependency ϕ Type A, B Expr e Dependency Vars $\alpha_1, \alpha_2, \ldots$ Vars $x_1, x_2, \ldots$

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

T-Unit
$$\frac{}{\Delta\ \Gamma \vdash \langle\rangle : \mathtt{unit} \mid \circ}$$

T-Var
$$\frac{}{\Delta\ \Gamma, x : A \vdash x : A \mid \circ}$$

T-Consume
$$\frac{\Delta\ \Gamma \vdash e : A \mid \phi}{\Delta\ \Gamma \vdash \#e : [A \cdot \phi] \mid \circ}$$

T-Produce
$$\frac{\Delta\ \Gamma \vdash e : [A \cdot \phi_1] \mid \phi_2}{\Delta\ \Gamma \vdash\, !e : A \mid \phi_1 \sqcup \phi_2}$$

T-Lam
$$\frac{\Delta\ \Gamma, x : A_1 \vdash e : A_2 \mid \circ \qquad \Delta \vdash A_1}{\Delta\ \Gamma \vdash \lambda(x.e) : A_1 \to A_2 \mid \circ}$$

T-Ap
$$\frac{\Delta\ \Gamma \vdash e : A_1 \to A_2 \mid \phi \qquad \Delta\ \Gamma \vdash e_1 : A_1 \mid \phi_1}{\Delta\ \Gamma \vdash \mathsf{ap}(e; e_1) : A_2 \mid \phi \sqcup \phi_1}$$

T-DepLam
$$\frac{\Delta, \alpha\ \Gamma \vdash e : A \mid \circ}{\Delta\ \Gamma \vdash \Lambda(\alpha.e) : \forall(\alpha.A) \mid \circ}$$

T-DepAp
$$\frac{\Delta\ \Gamma \vdash e : \forall(\alpha.A) \mid \phi' \qquad \Delta \vdash \phi}{\Delta\ \Gamma \vdash e[\phi] : [\phi/\alpha]A \mid \phi'}$$

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. 11. TS/SCI: Type System for the Structural Calculus of Indistinguishability (Selected Rules)

dependencies ϕ." Dependencies ϕ play the same role as before, now called an *ambient security level* in information flow parlance. This will be motivated thoroughly in Section 3.2.1. The type $[A \cdot \phi]$ corresponds to a satisfaction operator $@_\phi A$, and has syntax $\#e$ and $!e$ for introducing and eliminating it. Notice that the rules which do this look quite familiar, given the preceding context: T-Consume and T-Produce mirror @I and @E-new exactly, modulo syntax.

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 Γ, 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 ϕ 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. We'll justify this is in Section 3.2.1. There's also a premise $\Delta \vdash A_1$, which we'll 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 α into Δ, the environment responsible for tracking the in-scope information flow variables. This can be thought of analogously to the way that T-Lam introduces a variable x into Γ. The elimination rule T-DepAp for quantifiers is instantiation, substituting the instantiated ϕ into inner type of the quantifier. This is analogous to the dynamics for lambda application, but occurs statically. The type of `id` in Figure 6 under this syntax might be $\forall(\alpha.[\mathtt{int} \cdot \alpha] \to [\mathtt{int} \cdot \alpha])$, 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. We'll work an example of the latter in Section 3.2.1. The rules for functions and quantifiers will suffice to obtain declassification.

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 worlds α in the expression e, type A, and dependency set ϕ in the typing judgement are contained within Δ. We also want to ensure that any term variables x in e are contained under Γ. Regularity establishes this rigid lexical scoping. $\Delta \vdash A$ can be read as "all worlds α mentioned in A are members of Δ"; analogously for the other three scoping judgements.

Theorem 1 (Regularity). *If* $\Delta\ \Gamma \vdash e : A \mid \phi$ *and* $\Delta \vdash A_i$ *for each assumption* $x_i : A_i$ *in* Γ*, then* $\Delta \vdash e$ *and* $\Gamma \vdash e$ *and* $\Delta \vdash A$ *and* $\Delta \vdash \phi$.

Finally, T-Sub represents *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. This permits the security levels of expressions to be raised—that is, it permits them to add extra dependencies to themselves.[4] We have $[A_1 \cdot \phi_1] \sqsubseteq_\Delta [A_2 \cdot \phi_2]$ when $\phi_1 \subset \phi_2$ and $A_1 \sqsubseteq_\Delta A_2$. Beyond this, 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. In addition to the constructs shown here, we support sums $A_1 + A_2$ and products $A_1 \otimes A_2$ but defer explicating these to Section 5. We will generally adhere closely to the system as formalized, but once or twice will assume inductive or recursive types for the purpose of demonstration in a broader setting. We'll now look to a healthy helping of examples oriented towards the programming view.

3 More Examples and Subtleties

In this section, we'll review a couple 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 core issues 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` < `bob` and so `alice` ⊔ `bob` = `bob`. Why is this useful? Functions like in Figure 12 become typable:

This slightly silly 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's the right person. There's something odd going on here. We pass data at levels `alice` and α 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 β indicate that it only contains data from α and

[4]Note that this bears similarity to a necessity semantics, causing types to be valid in all future worlds, but is separate from the satisfaction connective.

```
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. 12. Declaring a Custom Dependency Ordering

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 be turned into bob, so it is.

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 [α] and [β], because our dependencies are *boring*. They have no implicit structure! It is all predetermined by their syntax. [α] can only be partially ordered less than a dependency set that contains it, like [α β]. 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 haven't made a strong case for the simplicity of our approach yet. We look to Figure 13 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. 13. 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 α dependency in their return types. Note that we are applying *simplification algorithms* in this example. We might have written the type of alc as:

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

Instead, the simplification algorithm recognizes that whatever the dependency level of 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're instead saddled with a bag of constraints which, once decoded, 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 rather 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 α and `alice`. The biggest difference is in the type of `both`, which states that data α 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 2003]. Next, we reconsider the heuristic of making fine-grained dependency tracking pervasive.

3.2 The Benefits of Explicit Satisfaction

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

let alice : [ alice ] int
let alice = 0

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

Fig. 14. The Constant Function

We've so far run with an implicit assumption—mirrored by other information flow systems [Pottier and Simonet 2003]—that all values 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 14. What dependency set should the `[ ? ]` be? The empty set `[ ]` of course! When we evaluate the call `const alice`, α gets instantiated to `[ alice ]`. α does not appear in the return type, so this has no effect. For the constant function, we of course 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 Figure 2. Need it have any information flow content in its types 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 if an element has some dependencies those 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 granularity or coarseness of tracking on us, besides as a per-type default, but permits us to choose. The connective `[ α ] A`—which we identified in Section 2.2 as the *satisfaction operator* from hybrid logic—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 doesn't encapsulate any dependencies, and because this eliminates all input dependencies the return value need not encapsulate any output dependencies. The ambient dependencies ϕ from the typing judgement in Section 2.3 can be justified now: they track dependencies not encapsulated inside types. We saw that the syntax `#e` introduces satisfaction, internalizing the current ambient dependencies into a type. Thus if `x` has type `int` and the ambient security context is α, the expression `#x` will have type `[ α ] int`. Similarly, we use `!e` to move the annotated dependencies from the type of an expression to the ambient security context. Putting these together, the expression `#(add !alice !alice)` would have type `[ alice ] int` as before.

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 `[ α ] A`, then attempt to apply satisfaction introduction. Analogously for the reverse direction. One could also do away again with the syntax, letting it be inferred. 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 leverage 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 15, 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. 15. Finding Simplifications in Map

Reading from left-to-right, we first annotate the function argument given to `map1` with α so that when it is used inside the body it can induce an α dependency. The function itself takes a `[ β ] int` as argument, which is the type of the contents of the list, and returns a `[ δ ] 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 β and δ 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 σ dependency for the argument list to represent the structure information. The structure of the returned list is dependent on both σ and α, 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 α and σ 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 directly to the return value, as formalized by T-Ap in Section 2.3. 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 2003], 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 we have said, 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 16, we omitted the actual bindings of the α variables. We now give a more explicit version as `id_explicit` in the figure, matching the formal system to be described in Section 5. The difference is the **`forall`** α sitting in front of the type signature, called a *quantifier*. This construct acts as a binder for α: 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. 16. 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 some dependencies and expecting that the α in its type will reflect these dependencies. `v2` shows the plumbing: we first instantiate `id_implicit` to `[ pwd ]`, whereupon the type system substitutes away the α for that set of dependencies. The type that results is still

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

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

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

Fig. 17. Existential Quantification

a function with the same input and output types, but is no longer polymorphic in α. 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`.

4.2 A Taste of 'Where' Declassification

We will 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. To support declassification, we need to encode existentials that hide the definition of dependency variables from client code. Consider the strange-looking type **`exists`** in Figure 17. The **`forall`** quantified β here is no longer in prefix position, because it has been moved inside the higher-order function. In fact, the β isn't *universally* quantified anymore—what we call **`forall`**—but *existentially* quantified. The ability to encode existential quantification using universal quantification is discussed in [Pitts 2000]. Existential quantification is the first piece of the declassification puzzle, and the program in Figure 17 already exhibits a form of declassification.

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 is done, the final answer is revealed via declassification.

Observe that `num` in `client` has type `[ β ] int`. We add `123` to it, and then return it as the result of the higher-order function. This function is passed to `impl`, which gives us back an **`int`**. Inspecting `impl` we see that it sets `num` to `7`, returning `130` to us after adding `123`. This result is obviously dependent on the initial value of `num`, which has dependency β, but the latter does not occur in the type of the final result! Indeed, β cannot be in the result type, as it is not in scope there. The reason β disappears is because `client` is polymorphic in β. This means that its logic must be written without knowing what β actually *is*—as though β could be anything. Its computation is then passed to `impl`, which leverages this polymorphism to instantiate β to `[ ]`. Inside the higher-order function in the body of `client`, the dependency β must be treated like any other, respecting the secrecy of the value. However, precisely because it is scoped inside `client`, the implementation in charge of β must set it to something else. It then unwraps[5] the now-unnecessary satisfaction operator on the `[ ] int` returned from `client` and returns it. In Sabelfeld and Sands's framework, we restrict *where* declassification can occur to the lexical scope where β variable is known to be bound to `[ ]`.

4.3 'What' Declassification: Revisiting Password Checking

We can now solve the problem with `check` from the introduction, laid out in Figure 18, and see how our approach can control *what* information can be declassified. We start by generalizing the basic schema of existential quantification a little. There's now a quantified variable α which allows us to return data at any dependencies. Importantly, α *cannot* mention β because it is scoped outside of it: this is a form of bounded quantification. Once again we use a higher-order function to allow the

[5]We return to implicit unwrapping, without using a !, to keep the example clean.

```
type exists = forall α . (forall β . F(β) -> [ α ] bool) -> [ α ] bool

F(π) ≜ {
    pass : [ π ] string,
    check : [ δ ] string -> [ δ ] bool,
    length : [ π δ ] string -> [ δ ] int
}

let impl : exists = fun compute -> compute [ ] {
    pass = "katya",
    check = fun attempt -> attempt == pass,
    length = fun pass_str -> strlen pass_str,
}

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

Fig. 18. Enterprise Grade Password Checker v2

client to compute on a secret value, the declassify the final result. This time, however, we provide the client's computations with a secret value `pass` along with *methods* that can manipulate it: `check` and `length`. To describe an object type encapsulating these, we use the $F(\beta)$ notation, which is a template that takes a dependency variable as an argument and splices in a record containing methods which mention that variable. `impl` works on the same principle as before, instantiating the existential dependency β 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` walks the happy path, instantiating α to `[ ]` and using `check`. `client1` is of type **bool**, as was promised in Section 1.1. `client2` attempts to be sneaky, instantiating α with `[ β ]` so it can try to do the password check without going through `check`, but existential quantification is having none of it. β is not even in scope at the point of instantiation! `client3` instantiates α to `[ ]`, but again accesses password data through `imports.pass`. This induces a β dependency— since the template was called with β as an argument—which will not check against the empty set.

There's one point left to illuminate. Look to `length`, which permits password-affected strings' lengths to be leaked. 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.* Declassification requires that labels can be uniquely decomposed into the concrete variables which comprise them, so functions may be written which 'match' on those constants in their inputs types and remove them, as `length` does. Joining in arbitrary lattices destroys information about the inputs, but the same in a free semilattice preserves this information. An alternative is to deploy a *Heyting algebra*, which adds implication to the existing $\sqsubseteq$ and $\sqcup$ operations. Implications involving information flow variables can be used to keep track of which existentially quantified dependencies they contain.

Scaling the existential approach to practical programs necessitates being able to express declassifiers like `length`. 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 `length`. For instance, in the case of Figure 18, you might have to re-export every string processing function through the password checker interface in anticipation of which ones users may wish to use without inducing password dependencies. And, naively done,

```
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. 19. Declaring a Custom Dependency Ordering, Computation-Relevantly

this would be unsafe because you likely do not want the output of most string processing functions to be able to leak! The general problem here is highlighted by [Cruz and Tanter 2019], which imports the the machinery of *faceted types* to address it. It is remedied here without the need for specialized modifications to the type system. Declassifying functions like `length` can be seen as a **computationally relevant ordering** on dependency sets which may violate the one given structurally. Specifically, `length` can be read as an ordering—notably where the left hand side is not a subset of the right— $[\ \pi\ \delta\ ] \sqsubseteq [\ \delta\ ]$ that must be manually applied wherever it is used. So this allows us to preserve the uniform structure of our information flow specifications while, in effect, introducing information flow policies on them. Let's review the example in Section 3.1 where we confronted policy declarations to see if we can capture them now.

4.4 'Who' Declassification

The program in Figure 19 revisits Figure 12, showing how our system can model an information flow policy that allows Alice's data to be shared to Bob, but not to anyone else. This corresponds to the *who* dimension of declassification from Sabelfeld and Sands [2009]. Our example uses a slightly higher-level syntax for existential dependencies here, closer to what ordinary programmers would see while employing our approach to declassification. In particular, we might imagine that **Alice** is defined like so:

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

Assume α is instantiated as needed for the eventual return dependencies of the program in Figure 19. Assume similarly that A is as needed for the program's eventual return type, since our core system lacks polymorphism over types. β corresponds to the instantiation **with** `[ bob ]` in Figure 19, enabling the latter to appear in the signatures of **Alice**'s interface. We then introduce our existential dependency `alice`, and both β and `alice` are passed to the interface template $F(\ldots)$ to generate the methods of **Alice**. We only want one of these methods—`reveal_alice`—so we bring only that one into scope, in addition to the dependency `alice` itself.

The purpose of the `reveal_alice` function is to declassify `alice` data to Bob by relabeling it as `bob` data. As before, the `reveal_alice` function can easily by implemented by the Alice module, as it knows the implementation of alice (presumably as the empty set). The only difference compared to Figure 12 is that `reveal_alice` must be called to realize the declassification. The computational interpretation of declassification shown here makes declassification more explicit—and therefore makes the program more understandable and secure—as this function must be called everywhere `alice` data is transformed to `bob` data. This is what is meant by *computationally relevant*: **policies**

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

T-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-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-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. 20. Positive Connectives for the Structural Calculus of Indistinguishability

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 for declassification: it is rare that a secret value should be leaked fully intact rather than after some computation which renders it inert. 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.

A further tweak is possible: we could introduce another existential module **Bob** which permits messages tagged with bob to be read without inducing a dependency, effectively declassifying messages after processing them. This is even better than what we had before, which would have forced the bob dependency to infect the program. 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 first before running a callback to declassification.

5 Metatheory

We have surveyed a zoo of interesting examples ranging beyond those in Section 1.1, but curiously, have not introduced any new primitive notions! Even existential quantification as revealed in the last section is simply leveraging the same machinery for polymorphism revealed in Section 2.3. This makes our job in this section relatively easy. First, we build on Figure 11 by providing the rest of the grammar and typing rules, shown in Figure 20.

5.1 Syntax and Typing, Continued

It turns out we ignored the rules for positive types—positive products and sums—in Section 2.3. T-InjL is one of two introduction rules for sums, the other being T-InjR. Observe that the dependencies ϕ of the expression e are propagated straightforwardly in both rules to their conclusion. The type

$$
\begin{aligned}
e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \phi_1 \not\sqsubseteq_\Delta \phi_2 \text{ or } e_1 \text{ val}, e_1' \text{ val}, e \longmapsto^* e_1, e' \longmapsto^* e_1', \\
&\quad e_1 \sim e_1' \in A \mid \phi_1\ [\phi_2]\ [\Delta] \\
e \sim e' \in \mathtt{unit} \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq e = \langle\rangle, e' = \langle\rangle \\
e \sim e' \in A_1 + A_2 \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq e = \mathtt{l} \cdot e_1, e' = \mathtt{l} \cdot e_1', \\
&\quad e_1 \overset{*}{\sim} e_1' \in A_1 \mid \phi_1\ [\phi_2]\ [\Delta] \text{ or} \\
&\quad e = \mathtt{r} \cdot e_1, e' = \mathtt{r} \cdot e_1', \\
&\quad e_1 \overset{*}{\sim} e_1' \in A_2 \mid \phi_1\ [\phi_2]\ [\Delta] \\
e \sim e' \in A_1 \otimes A_2 \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq e = \langle e_1, e_2\rangle, e' = \langle e_1', e_2'\rangle, \\
&\quad e_1 \overset{*}{\sim} e_1' \in A_1 \mid \phi_1\ [\phi_2]\ [\Delta], \\
&\quad e_2 \overset{*}{\sim} e_2' \in A_2 \mid \phi_1\ [\phi_2]\ [\Delta] \\
e \sim e' \in [A \cdot \phi] \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \mathtt{!}e \overset{*}{\sim} \mathtt{!}e' \in A \mid \phi \sqcup \phi_1\ [\phi_2]\ [\Delta] \\
e \sim e' \in A_1 \to A_2 \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \forall \phi_2' \sqsubseteq_\Delta \phi_2\ .\ e_1 \text{ val}, e_1' \text{ val}, \\
&\quad e_1 \overset{*}{\sim} e_1' \in A_1 \mid \phi_1'\ [\phi_2']\ [\Delta] \implies \\
&\quad \mathsf{ap}(e; e_1) \overset{*}{\sim} \mathsf{ap}(e'; e_1') \in A_2 \mid \phi_1' \sqcup \phi_1\ [\phi_2']\ [\Delta] \\
e \sim e' \in \forall(\alpha.A) \mid \phi_1\ [\phi_2]\ [\Delta] &\triangleq \Delta \vdash \phi \implies e[\phi] \overset{*}{\sim} e'[\phi] \in [\phi/\alpha]A \mid \phi_1\ [\phi_2]\ [\Delta]
\end{aligned}
$$

Fig. 21. Semantic Equality for the Structural Calculus of Indistinguishability

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 ϕ in the conclusion to those coming from e_1 or e_2. We require e_1 and e_2 to have the same dependency level as a simplification, because subsumption can be used to increase the level of both to their least upper bound. Introducing positive products works similarly with respect to the flow of dependencies, with the introduction rule T-Pair propagating the dependencies ϕ_1, ϕ_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.

There is an alternative way to formulate an introduction rule for products, by forcing e_1 and e_2 to consume their dependencies into their types as we saw in Section 2.3 for the negative connectives. 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. Note then that negative (or *lazy*) products would track dependencies more granularly than positive products, as expected from Section 3.2.1: each projection's dependencies may be distinguished from the other's. Not so for positive products, which blend both projections' dependencies ϕ_1 and ϕ_2 together. Positive products can be viewed as more general than negative products in our setting, by simply suspending one or both elements of the pair, so we do not provide them directly. 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 semantic binary logical relation—*semantic* because the relation is defined entirely by the behavioral properties of programs related by it. 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. Noninterference is captured with a corollary stating that any function whose output dependencies are not a strict superset of the input dependencies must be a constant function, fufilling our promise from Figure 5. Unique to our approach, declassification reasoning emerges without any additional effort besides that needed for handling quantifiers. We show the full-fledged logical relation in Figure 21.

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. $e \overset{*}{\sim} e'$ can be read as "e relates to e' at type A with security level ϕ_1 and observer level ϕ_2 under in-scope dependency variables Δ." Here e, e' are closed expressions, containing no variables. If the security level—which plays the same role as the ϕ in the typing judgment—is a subset of, or equivalent to, 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 $e \overset{*}{\sim} e'$ codifies this, and is called the *non-interference condition*. If the security level of some related expressions ϕ_1 is *not* a subset of the current observer level ϕ_2, then to that observer, the expressions are equal. This is the core of why non-interference is a *hyperproperty*, or inherently a matter of two or more related traces of a program: we need some intuitively inequal programs to be equal as a matter of non-interference. The intuition is that the logical relation knows the observer won't be able to see what the programs end up doing, so it can equate them trivially.

In the case where the equality is not trivial, a few things occur. First, e and e' must evaluate to values e_1 val and e_1' val. And e_1, e_1' must be related at $e_1 \sim e_1'$, the non-starred relation. The definition of $e \overset{*}{\sim} e'$ by evaluation immediately gives us *forward closure* and *backwards closure*, listed below. These two theorems capture why the logical relation is semantic: it is preserved by evaluation, *in both directions*. Backwards closure 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 semantically well-typed, but not syntactically well-typed.

Lemma 2 (Forward Closure). *If* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ *and* $e \longmapsto^* e_1$ *then* $e_1 \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$.

Lemma 3 (Backwards Closure). *If* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ *and* $e_1 \longmapsto^* e$ *then* $e_1 \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$.

The relation $e_1 \sim e_1'$ 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 if equality of the inner or eliminated expressions—for instance, the projections of a product, or a function application form—can be obtained via non-interference. 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 is a larger set and therefore less likely to be a subset of the observer level. Note that while the security level ϕ_1 is *monotonic*, the observer level ϕ_2 is *anti-monotonic*. This is crystallized by the following two lemmas, proved by induction on types.

Lemma 4 (Monotonicity). *If* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ *and* $\phi_1 \sqsubseteq_\Delta \phi_1'$ *then* $e \overset{*}{\sim} e' \in A \mid \phi_1'\ [\phi_2]\ [\Delta]$.

Lemma 5 (Anti-monotonicity). *If* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2]\ [\Delta]$ *and* $\phi_2' \sqsubseteq_\Delta \phi_2$ *then* $e \overset{*}{\sim} e' \in A \mid \phi_1\ [\phi_2']\ [\Delta]$.

The security level can be freely raised while preserving non-interference, while the observer can always choose to drop permissions for viewing certain dependencies. That the dependency elision heuristic from Section 3.2.2 preserves non-interference can intuitively be justified via monotonicity: it always produces more dependencies into the ambient security level than would have otherwise appeared there. As mentioned, we leave the algorithm for inferring elisions, along with the completeness of that algorithm, for future work alongside inference for the type system as a whole. We next show that our logical relation satisfies reasonable equality properties.

Lemma 6 (Symmetry). *If* $e_1 \overset{*}{\sim} e_2 \in A \mid \phi\ [\phi']\ [\Delta]$ *then* $e_2 \overset{*}{\sim} e_1 \in A \mid \phi\ [\phi']\ [\Delta]$.

Lemma 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* $e_1 \overset{*}{\sim} e_3 \in A \mid \phi\ [\phi']\ [\Delta]$

The logical relation *does not* satisfy reflexivity; semantically ill-typed expressions may not reduce to a value as the logical relation requires. Thus, in line with the standard approach to logical relations, ours is a *partial equivalence relation*. Finally, we get to the interesting part. The fundamental theorem translates syntactic well-typedness to membership in the logical relation. The typing rules work on expressions containing variables, but the logical relation only works on closed terms, so we must generalize our existing picture.

Theorem 8 (Fundamental Theorem). *If* $\Delta_0, \Delta\ \Gamma \vdash e : A \mid \phi$ *then* $\widehat{\delta}(\widehat{\gamma}(e)) \overset{*}{\sim} \widehat{\delta}(\widehat{\gamma'}(e)) \in \widehat{\delta}(A) \mid \widehat{\delta}(\phi)\ [\phi']\ [\Delta_0]$.

In standard fashion, we introduce closing substitutions $\gamma, \gamma' : \Gamma$ which map variables x contained in logically related expressions to other related expressions. We have another typing environment Δ to take care of, though, and its treatment is not rote. The statement of the fundamental theorem splits up the information flow variable environment from the typing judgment into Δ_0 and Δ. Δ_0 denotes *the observer's dependency environment*. The definition of equality requires that $\Delta_0 \vdash \phi'$. Δ_0 provides an environment for a closing substitution on dependency variables α to be *closed under*, because all dependencies considered in the logical relation must be meaningful to the observer. This is the role of δ, which maps each dependency variable α in Δ to a ϕ closed under Δ_0. Given a δ and γ, γ', we can create a fully closed pair of expressions, type, and security level. That the open and closed logical relations are *relativized* to a base Δ_0 representing the observer's environment will turn out to be the key that unlocks the declassification story. The fundamental theorem itself is proved by induction on the height of typing derivations. For notational clarity, we write $\Delta\ \Gamma \gg^{\phi'}_{\Delta_0} e \overset{*}{\sim} e \in A \mid \phi$ to mean the same as the conclusion above, where the closing substitutions δ and γ, γ' are with respect to Δ and Γ. The constant function property we desired in Figure 5 emerges as a straightforward corollary of this theorem.

Corollary 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* $\circ\ \Gamma \gg^{\phi_2}_{\Delta} e \overset{*}{\sim} \lambda(x.\mathsf{ap}(e; c)) \in [A_1 \cdot \phi_1] \to [A_2 \cdot \phi_2] \mid \circ$.

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, it states that such a function is observationally equivalent to a function which has its argument stubbed out with some constant c. We'll skim through the reasoning. Proving relatedness at function type $[A_1 \cdot \phi_1] \to [A_2 \cdot \phi_2]$ involves assuming related arguments e_1, e_1' at $[A_1 \cdot \phi_1]$ and security level $\circ$ and showing the equality of their application forms at $[A_2 \cdot \phi_2]$. We first use the fundamental

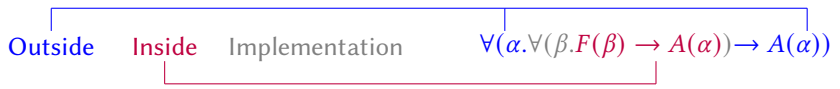
theorem on the typing assumption for e. We then obtain an instance of the logical relation relating $!e_1$ and $!c$, the constant value we assumed, at type A_1 and security level ϕ_1 by non-interference. We then apply this to the definition of the logical relation at $[A_1 \cdot \phi_1]$ to obtain an equality relating e_1, c at type $[A_1 \cdot \phi_1]$ with security level $\circ$. The equality from the fundamental theorem can then be aplied to this equality to obtain another equality at $[A_2 \cdot \phi_2]$. Finishing up, we use head expansion on the lambda $\lambda(x.\mathsf{ap}(e; c))$ which, when applied to e_1', evaluates to the expected application form to obtain the result. The actual proof contains a few more details, but not many. In short, it should not matter at all which argument our original function is applied to: from the perspective of the observer its behavior is unchanged, and an equivalent function can be provided in terms of the original that short circuits the argument.

Importantly, this function is only constant *from the observer's perspective*. We can imagine a situation where the return type of the function is $[[A_1 \cdot \phi_1] \cdot \phi_2]$ and the implementation is $\#x$ where x is the argument of the function. Intuitively, this does not present an issue because $[[A_1 \cdot \phi_1] \cdot \phi_2]$ is a negative type, and therefore defined by its behavior under usage and not its contents. Upon usage, however, ϕ_2 enters the security level and the inner computation becomes unobservable. Here we chose the observer level ϕ_2 to be the security level ϕ_2 in the top layer of the return type, but it is no challenge to specialize this to the situation at hand. Observe that Figure 5 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 Declassification, Formally

It turns out we have already accounted for declassification without additional effort beyond that already expended for quantification—which is admittedly nontrivial. However, as discussed in Section 2.2, a treatment of quantification is a prerequisite to practical information flow reasoning, so in a certain view declassification comes "for free." The key is the observer's Δ_0 to which the logical relation is relativized, and the multiple-points-of-view interpretation of declassification that falls out of the existential encoding. The scheme for existential quantification is reproduced here. We write $A(\alpha)$ to mean that the type A may contain α. We consider this type from the perspective of its usage, that is, a `client` from Figure 18.



The β is the existential variable, and the α is a universally quantified variable that allows return dependencies to be specified for the existential computation marked in **red**. This portion of the type marks the *inner perspective*: setting your point of observation to it by taking its dependency environment Δ in the type derivation to be the logical relation's Δ_0 causes the latter to handle all declassifiers in $F(\beta)$. In particular β is meaningful to the observer, so it assumes via closing substitutions γ, γ' non-interference-respecting implementations of all functions in $F(\beta)$ provided in terms of β. For instance, `length` in Figure 18 will be assumed to be constant. This can be seen as a form of *quotienting*: the observer is assumed to be concerned only with their realm of observation—the current subexpression, or the body of the existential—so disequalities resulting from existentially

quantified assumptions in the environment are automatically ignored by the logical relation. This is precisely what is desired. We do not want to pay any mind to all the declassifiers in the program when reasoning about the non-interference properties of a local portion of it, only the properties of that fragment. If dependency tracking is being violated in a way that is *not* permitted by the environmental assumptions, then the logical relation will fail to produce an equality—but it is otherwise permissible.

On the other hand, the **blue** perspective is the *outer* one: it never sees the declassification occuring in the inner fragment at all. Stepping through the logical relation, the quantifier case will instantiate the α to some ϕ scoped under Δ_0—which now *does not* contain β. The function case will assume some argument at type $\forall(\beta.F(\beta) \to A(\phi))$ and apply it, immediately yielding an $A(\phi)$ back and skipping over all the declassification machinery. No declassification that occurs in the **red** portion may affect the non-interference reasoning on the **blue** side, because the ϕ substituted in for α *cannot* contain β. Note that the **gray** part of the type is the implementation of the declassifiers, given by `impl` in Figure 18. This will also satisfy the logical relation because the existential variable is known to be instantiated to empty there, so does not appear at all. So Theorem 9 does not *really* mean that a function where the return type's security level is lower than the argument's must strictly be the constant function. It rather means that it is observationally constant with respect to the observer's perspective (subexpression), including the environmental assumptions attached to that perspective, as emphasized previously. If the context is empty or no existentials are in play then such a function is guaranteed to be truly constant—but this is much less practical.

6 Related Work

6.0.1 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.

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.

6.0.2 Logical 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 from Reed [2009] and had not yet been published.

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 out setting might transfer to a principal-based approach.

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 an approach to declassification via existential quantification without the limitations or complexity of prior work. We have shown that the latter both 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

The auxiliary formal definitions and proofs contained in the appendix to this submission will be submitted as an artifact. Since they are paper-based, we will only ask for an "artifact available" badge.

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.

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.

Torben Braüner. 2022. "Hybrid logic." In: *The Stanford encyclopedia of philosophy*. Stanford University.

Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. "A Dependent Dependency Calculus." In: *European Symposium on Programming*. Springer International Publishing Cham, 403–430.

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.

Matt Fairtlough and Michael Mendler. 1997. "Propositional lax logic." *Information and Computation*, 137, 1, 1–33.

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

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.

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.

Paul Blain Levy. 1999. "Call-by-push-value: A subsuming paradigm." In: *International Conference on Typed Lambda Calculi and Applications*. Springer, 228–243.

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.

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

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.

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

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.

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.

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.

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

Andrew M. Pitts. 2000. "Parametric polymorphism and operational equivalence." *Mathematical Structures in Computer Science*, 10, 3, 321–359. doi:10.1017/S0960129500003066.

François Pottier and Vincent Simonet. Jan. 2003. "Information Flow Inference for ML." *ACM Transactions on Programming Languages and Systems*, 25, 1, (Jan. 2003), 117–158. ©ACM.

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.

Christopher Strachey. 2000. "Fundamental concepts in programming languages." *Higher-order and symbolic computation*, 13, 11–49.

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.