The Duality of Information Flow

Reconciling Robust Downgrading with Non-Interference

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

Non-interference properties have long enjoyed a position as the high water mark of program security guarantees. For instance, *confidentiality* states that a secret does not interfere with a given computation, precluding the former from escaping through the latter. *Integrity* analogously states that untrusted data does not interfere with some computation, preventing the former from manipulating the latter. *Information flow type systems* comprise the primary means for obtaining non-interference properties of programs, but their potential as a holy grail for secure programming has remained latent. Prior work bifurcates the type system along confidentiality and integrity, resulting in duplicate reasoning machinery and complex specifications. Furthermore, long-held wisdom dictates that non-interference must be weakened with *downgrading* mechanisms to accommodate the needs of practical programs, nearly all of which violate confidentiality and integrity in the course of fulfilling their purpose. This often pierces abstraction barriers and compromises modular reasoning.

We introduce *parametric information flow*, which uses recent insights from modal type theory to shed light on these issues. In particular, we draw inspiration from work on the *open* and *closed* modalities, highlighting their rich interplay. Though each individual modality finds uses throughout the literature, our key insight is that *their joint interaction* suffices to reconstruct full-spectrum information flow reasoning, producing a single framework accounting for both confidentiality and integrity. Downgrading and analogues of advanced reasoning tools in the lineage of *robust declassification* are recovered without extensions to our theory, strengthening prior results. We show non-interference via a binary logical relations argument, realizing robustness as an ordinary 2-hyperproperty mediated by our modalities. Our work reveals state-of-the-art downgrading mechanisms to be wholly compatible with those for abstraction and modularity, arising precisely from the semantics of the latter under full-strength non-interference.

1 Introduction

Shall secure information flow focus on tracking *information* or *flows*? That is, in considering the dependency of a computation on some data, should primacy be given to what can be *gleaned about* that data by the computation, or to the manner of its *movement through* the computation? We say both. Much prior work in information flow chooses one or the other, a seemingly immaterial choice between options so subtly distinct as to be without a difference. However, we will show that it makes all the difference in the world to consider them as equal parts of a whole.

1.1 Information Flow by Parametric Polymorphism

In embarking on illuminating the preceding remark—a process which will span the length of this work—we begin with a quick tour of our approach to information flow. We start at the familiar machinery of parametric polymorphism [Reynolds 1983]. Consider the typing for the polymorphic identity `id` : α -> α, which constrains its return value to be exactly its argument. Likewise, the typing `fst` : α -> β -> α specifies a function which returns its first argument and ignores its second. Finally, the type **`list`** α -> (α -> β) -> **`list`** β of the `map` function on lists states that the elements of the argument list are passed into the higher order function, whose return values in turn populate the elements of the returned list. In each case, *parametricity* allows us to reason simultaneously about properties of both the *information* referenced by each signature and the *flows* described by it.

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

Specifically, we reason about flows by interpreting each type variable as naming some data, with its positioning inside the signature explicating the movement of that data through the computation. In the case of α -> α the appearance of α in the return type tells us that the computation *depends on* the argument at α. On the other hand, we reason about information through data abstraction, or the hiding of information. This is what lets us know that a computation at type α -> α *cannot depend on* the details of the particular α passed to it. Taken together, these two properties of data dependency—or information flow—narrow the space of possible behaviors of a computation at type α -> α to contain only the identity function, a classic result of parametricity; similar results can be obtained for `fst` and `map`. Usual appeals to this principle merge both threads of reasoning into one.

The unification by parametricity of concerns about both flows and information is often quite useful, but its sheer strength significantly constrains the range of expressible programs. For instance, what if we wish to speak of the information flow properties of a function `increment : int -> int` which computes with—so cannot be polymorphic in—its input? Remedying this necessitates distinguished notions of flows and information able to be deployed separately. We proceed to isolate each in turn, making it available in a more general setting. We initially retrace the development by Gouni et al. [2025] of certain forms of information flow via parametric polymorphism.

1.1.1 Isolating Flow Reasoning. The preceding approach to reasoning about the movement of data within some signature was to trace its associated type variables α, but this will not work if we also wish to perform arbitrary computations with that data. Let us instead try *tagging* types with *dependency variables* like α, rather than having α *be* the type. Under this proposal, we might type the increment function on integers as α `int` -> α `int`. The α names the attached data, allowing us to track its movement from the argument at α to the return value dependent on α. Importantly, unlike for type variables, α no longer plays any role in reasoning about information hiding.

This seems to leave us better off than before, in that we can simultaneously track the movement of data and compute with it, but it is still impractical. To see why, imagine how we might type addition on integers. We would like to express that it takes two arguments and returns a result dependent on both. We start by writing α `int` -> β `int` -> $\boxed{?}$ `int`. How should the $\boxed{?}$ be filled? Our current syntax limits us to mentioning a single dependency variable inside it, but we clearly need to mention both α and β because both arguments flow to the return value. So we generalize our syntax to handle *sets* of dependency variables within types, rather than just *singular* dependency variables. Setting $\boxed{?}$ = [α β], we successfully type addition as [α] `int` -> [β] `int` -> [α β] `int`. For consistency, the dependency variables on the arguments have also been generalized to dependency sets.

We refer to α as a dependency *variable* because our signatures are still polymorphic, but over dependencies rather than types. In particular, each α ranges over another dependency set [α_1 … α_n], meaning the former may be instantiated to the latter. Just as the α in α -> α can be instantiated to `int` to obtain `int -> int`, we can instantiate the α in our type for addition to `[secret1 secret2]` to obtain `[secret1 secret2] int -> [`β`] int -> [secret1 secret2` β`] int`. A function at this type takes an integer at the dependencies `[secret1 secret2]` and eventually returns an integer at those dependencies plus those of its second argument. As a notational convention, we will use α, β, δ, … to indicate polymorphic dependency variables; all others are fixed.

For a more grounded example of tracking flows consider the case of a password checker in Figure 1. We first declare a `string pass` tagged with `secret` to indicate that it represents password data. We assume `secret` to be in scope here, but defer detailing the mechanisms used to introduce it. `check` takes a password `attempt string` at any dependencies α and compares it to `pass`, returning a `bool` tagged with α and `secret` to indicate the flows induced from `attempt` and `pass`.

Note that the type of `check` is slightly redundant: the appearance of α in both the argument and return type means any call to `check` will *always* depend on the argument passed

```
let password : [secret] string = "takver"

let check : [α] string -> [α secret] bool =
    fun attempt -> attempt == password
```

Fig. 1. Simple Password Checker

to it. We can use *dependency elision* [Gouni et al. 2025] to simplify its type, eliding α to get `check : string -> [secret] bool`. When a dependency is not mentioned at a formal argument, those on the actual argument are propagated directly to the call site. So assuming we have some `x : [alice] string`, the application `check x` type checks under `[secret alice] bool`. Similar reasoning applies to the previously discussed types for integer increment and addition. Our examples will use this ability going forward. A further complaint about the type of `check` regards its impracticality: a useful password checker must leak its result, so we do not wish the returned `bool` to be `secret`. This too will be allayed through Section 3 and Section 4, where we discuss our ability to support *downgrading*[1] using only features already introduced.

The moves we have made so far are analogous to those of Gouni et al. [2025]; indeed, part of the preceding exposition has been adapted from there. However, having exhausted their insights we find ourselves at an as-yet-incomplete picture. From a theoretical perspective, it is suspicious that the other aspect of parametricity—information hiding—has not yet played a role. Practically, we lack the ability to speak about a number of important secure programming patterns.

For instance, while it is useful to track the dependence of a computation on password data, we might like to determine that the result of some computation *cannot* depend on—that is, cannot leak—password data. Our only chance of doing this at present is to manually verify that its dependency set does not contain `secret`. But even this falls flat, for instance, in the presence of polymorphic dependencies α which may later be instantiated to dependency sets containing `secret`: we would have to verify upon every instantiation that the resulting set is still free of `secret`. As another example, we would like to know that data tagged with `untrusted` does not affect some sensitive computation. We could again look through its dependency set and manually validate that it is free of `untrusted`, but this is subject to the same pitfalls as before. The current state of affairs is evidently unsatisfactory. Conveniently, filling in the theoretical part of our incomplete picture will turn out to do the same on the practical side, so let us investigate information hiding.

1.1.2 Isolating Information Reasoning. Having isolated reasoning about flows, can we do the same for information? We focus in particular on its *hiding*, going by our deconstruction of parametricity, because this is the essence of data abstraction. To account for this we introduce new syntax for *anti*-dependencies explicating that on which a computation *cannot* depend. Specifically, we provide a way to prohibit some data from being used by certain computations by tagging the former with anti-dependencies against the latter. Data tagged as such can be seen as *conditionally* abstract, that is, only from the perspective of computations possessing those dependencies. This is a generalization of information reasoning over that afforded by parametricity, as with reasoning about flows above.

Let us see how this helps us with locally preventing a computation from leaking secrets, as previously desired. Consider some computation `public : ... -> [α] int` whose result we would like to declare cannot depend on password information. With anti-dependencies in hand, this is as simple as inserting one regulating against `secret` into its return type: `... -> [α] ![secret] int`. If α is instantiated to `secret` when `public` is used, we will be left with `[secret] ![secret] int` ⚠.

[1]We use 'downgrading' here to avoid referring individually to declassification or endorsement, since our deconstruction of information flow will yield a theoretical divide distinct from the one between confidentiality and integrity.

```
1 let user_input : [untrusted] string = "saio pae"
2 let neural_net : ![untrusted] (string -> bool) = ...
```

```
3 let _ : [secret] ![untrusted] bool = neural_net password
4 let _ : [untrusted] ![untrusted] bool = neural_net user_input ⚠
```

Fig. 2. Preventing Adversarial Input to a Neural Network

This is an error because a `![secret] int` is hidden from computations dependent on `secret`. Figure 2 provides further motivation, protecting sensitive computations from `[untrusted]` data.

We begin by declaring `user_input` tagged `[untrusted]` and a function `neural_net` whose type is enclosed in `![untrusted]` to indicate that it must not be invoked within an `untrusted`-dependent computation, because of its inherent vulnerability to adversarial input. It takes a `string` and returns a `bool`, using dependency elision to slightly simplify what ordinarily would have been written `[`α`] string -> [`α`] bool`. We make use of it on line 3, passing it the `password` variable from Figure 1 and storing the result into a `bool` appropriately tagged `secret` to indicate the dependency. The next line swaps out `password` for `user_input`, culminating in an error due to the conflict between the `[untrusted]` argument and the restriction against such a dependency.

Observe our phrasing around the mechanics of anti-dependencies: **a conflict between dependencies and anti-dependencies arises when a computation at some dependency *contains* data which has declared an anti-dependency on the same.** In terms of the prior example, we care specifically about the existence of `[untrusted]` with `![untrusted]` nested *inside* it. One might wonder whether the other order of composition `![untrusted] [untrusted] bool` results in an error. It does not, because in this case the data which has declared an anti-dependency on `untrusted` is not yet itself dependent on `untrusted`, not being under such a dependency. The semantics of dependencies and anti-dependencies given by *polarity* will quell any unease about the possibility of mis-specification owing to this subtlety, however. *Suspending* `[untrusted] bool` will render the reverse order of composition inert in Section 3. We look now to it and the rest.

1.2 A Preview of the Rest

Parametricity has provided a convenient inroads to exposing our approach to information flow, motivating its deconstruction into *dependencies* and *anti-dependencies*. It turns out that the type connectives `[`$\alpha\ \beta\ \ldots$`]` and `![`$\alpha\ \beta\ \ldots$`]` correspond to the *open* and *closed* modalities previously discussed by Sterling and Harper [2021a] in the context of parametricity for program modules, alongside others. In this work we shift our attention from parametricity to non-interference, driving the design of information flow types using the complementary dependency tracking behaviors of these modalities. Their semantics deeply inform the distinction between information and flows.

Perhaps surprisingly, all the core features of our system have been discussed at this point; what remains is to reveal how. Section 2 looks to the broader body of work on information flow, isolating two distinct varieties of dependency tracking which appear independently and interchangeably throughout. In Section 3 we establish the correspondence of these varieties to the open and closed modalities, showing that though taken separately they may seem similar, taken together their differences shine. We walk through a number of further examples in Section 4 which exploit the interaction between our modalities to exhibit full-spectrum information flow reasoning encompassing both confidentiality and integrity, furthermore reconstructing *robust downgrading*. Section 5 formally grounds these examples, demonstrating non-interference for our system and

shedding light on the relationship between the joint semantics of our modalities and robustness. Section 6 reviews related work, and we conclude in Section 7. In summary, our contributions are:

(1) We review the literature on information flow and find **two distinct flavors of dependency tracking,** deployed largely without regard to their differences. Each flavor is realized by a different formulation of the primary dependency tracking modality. We tease apart the metatheory of each, finding that one deals with flows and the other with information.
(2) We **re-cast each flavor in correspondence to either the open or the closed modality,** illuminating their semantics from the point of view of proof-theoretic polarity and modal type theory. We explicate why one corresponds to *dependencies* and the other to *anti-dependencies* when appearing in concert, despite acting analogously when apart.
(3) **We reconstruct full-spectrum information flow, covering both confidentiality and integrity, through our modalities and their joint interaction.** We push this further to recover an improved form of robust information flow reasoning over prior work, including *robust downgrading*—all without extensions to our framework beyond what has already been introduced. A number of practical examples serve to ground these points, moreover demonstrating the **benefits to specification simplicity of our framework.**
(4) We give a logical relations argument for non-interference which **realizes robustness as an ordinary 2-hyperproperty,** simplified from prior presentations. We owe this to robustness being *built in* to the semantics of our modalities; its recovery through their interplay places prior work in this area on firm logical footing. Remarkably, **our non-interference property accounts for sophisticated downgrading mechanisms without making semantic concessions** to them, providing extraordinarily strong modularity guarantees. Our results have been mechanized in the Lean theorem prover.

Our work ultimately aims to advance the perspective that practical, full-spectrum information flow programming can be pursued under full-strength non-interference, and that this can be negotiated through the duality that results from our deconstruction of information flow. We begin by tracing the roots of this duality through the literature.

2 Background: Two Modalities for Dependency Tracking

Prior work on dependency tracking has deployed a diversity of modalities for doing so. Of these we survey those that form *monads* [Moggi 1989], arranging them along two distinct lines: one dealing with flows, the other with information. These can respectively be re-cast as the *open* and *closed* modalities of Rijke et al. [2020]. For narrative ease, we start with the latter. This section takes notational liberties as needed for clarity in presenting prior systems. The type system in Section 3 will be revealed to import the dependency tracking machinery introduced in this one.

2.1 Isolating Information, Formally

The Dependency Core Calculus (DCC) of Abadi et al. [1999] pioneered the use of monads for dependency tracking. It is organized around a monadic type connective $T_{\alpha_1 \dots \alpha_n}(A)$ denoting that A is dependent on $\alpha_1 \dots \alpha_n$. The rules for this connective are given in Figure 3 (top). The introduction rule Seal annotates an expression at any type with a set of dependencies ϕ, where ϕ is of the form $\alpha_1 \dots \alpha_n$. The elimination rule Unseal unwraps the dependency annotation from e_1, extracting the contained value at the inner type and passing it to e_2. The critical portion is the final premise $A_2 \geq \phi$, which states that *all the information in* A_2 *must be sealed at* ϕ *or above.* We defer a formal description of this sealing obligation to the next section; it is easier to illustrate instead by example. Figure 3 (bottom) shows the validity of different choices of A in $A \geq \alpha_1\ \alpha_2$.

$$\textsc{Seal}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \texttt{seal}[\phi](e) : T_\phi(A)} \qquad \textsc{Unseal}\ \frac{\Gamma \vdash e_1 : T_\phi(A_1) \qquad \Gamma, x : A_1 \vdash e_2 : A_2 \qquad A_2 \geq \phi}{\Gamma \vdash \texttt{unseal}\ e_1\ \texttt{in}\ x.e_2 : A_2}$$

unit bool ⚠ T_{α_1}(bool) ⚠ $T_{\alpha_1\,\alpha_2}$(**bool**) **unit** * **unit**
bool * unit ⚠ $T_{\alpha_1\,\alpha_2}$(**bool**) * **unit** int ⚠ list unit ⚠ **bool** -> **unit**

Fig. 3. DCC Dependency Tracking Connective (Top); Compatible Sealings (Bottom)

In short, Unseal ensures that any computation using sealed data places all the information it produces under the same level of sealing. Any part of the result type which risks exfiltrating previously sealed information is affected by this requirement. Accordingly, the need to `seal` is entirely elided when the result type is one incapable of revealing any information, such as **unit** or **unit** * **unit** or **bool** -> **unit**. Sealing as employed here crystallizes the idea that **the DCC dependency tracking monad fundamentally concerns information, or what can be *gleaned about data*, analyzing that represented by types.** Importantly, information is its only concern, not any other property of the computation at hand. Indeed, $T_\phi(A)$ can be understood as *internalizing* the judgement-level correspondence between types and information given by $A \geq \phi$.

The DCC monad was recognized as the *closed modality* from later work in univalent foundations [Rijke et al. 2020] by Sterling and Harper [2022], who elegantly reworked its categorical semantics. *Modality* here refers to an *idempotent monad*, specifically a *semantic* view of one. Idempotency means that we have $T_\phi(T_\phi(A)) \cong T_\phi(A)$. This allows modalities to be characterized by their *semantics*, or behavior, rather than by their syntax. Specifically, we say A is T_ϕ-modal if $T_\phi(A) \cong A$. This is what sets the DCC dependency tracking connective apart from an ordinary monad: Bind does not conclude at $T_\phi(-)$, as the latter would, but at a type A satisfying $A \geq \phi$. This is a lightweight way of checking that A is T_ϕ-modal. In other words, our concern is whether the resulting type *behaves* in the expected way in terms of revealing information, rather than being conservatively forced to exhibit a particular syntax. We defer further discussion to Rijke et al. [2020], who provide a comprehensive overview of modalities in the sense deployed here, and Choudhury [2022, §4.4], who focuses on the DCC modality and its relationship to standard monads.

Much later work exploits the DCC approach to dependency tracking [Bowman and Ahmed 2015; Cecchetti et al. 2017; Choudhury 2022; Hirsch and Cecchetti 2021; Liu et al. 2024; Rajani et al. 2025; Shikuma and Igarashi 2008; Sterling and Harper 2022], or references doing so. However, despite crediting the DCC—or closed—modality, some such works cut much closer in their chosen flavor of dependency tracking to its complement, the *open modality*. This is an easy oversight, for the latter has been comparatively neglected in the information flow literature; we remedy this now.

2.2 Isolating Flows, Formally

We show the setup for the second kind of modality in Figure 4 (top), taken from Shikuma and Igarashi [2008, §2.2] who introduce it as a reformulated variant of that from DCC. This modality is still monadic, but makes no mention of the sealing judgement defining the previous. It works instead by interacting with the ϕ environment newly added to the typing judgement, denoting the ambient security level. Specifically, it represents the current set of dependencies of the expression being typed. The introduction rule Consume takes an expression at type A and ambient dependencies $\phi \cup \phi'$ and captures the ϕ' dependencies within a newly formed modality $\phi' \Rightarrow A$, moving them

$$\textsc{Consume}\ \frac{\Gamma; \phi \cup \phi' \vdash e : A}{\Gamma; \phi \vdash \lambda(\_.e) : \phi' \Rightarrow A} \qquad \textsc{Produce}\ \frac{\Gamma; \phi \vdash e : \phi' \Rightarrow A \qquad \phi' \subseteq \phi}{\Gamma; \phi \vdash \mathsf{ap}(e; \phi') : A}$$

$$\dfrac{\dfrac{x : \alpha \Rightarrow \mathbf{int}, y : \beta \Rightarrow \mathbf{int}; \alpha \vdash x : \alpha \Rightarrow \mathbf{int} \qquad \alpha \subseteq \alpha}{x : \alpha \Rightarrow \mathbf{int}, y : \beta \Rightarrow \mathbf{int}; \alpha \vdash \mathsf{ap}(x; \alpha) : \mathbf{int}}\ \text{P}}{x : \alpha \Rightarrow \mathbf{int}, y : \beta \Rightarrow \mathbf{int}; \varnothing \vdash \lambda(\_.\mathsf{ap}(x; \alpha)) : \alpha \Rightarrow \mathbf{int}}\ \text{C}$$

$$\dfrac{\dfrac{x : \alpha \Rightarrow \mathbf{int}, y : \beta \Rightarrow \mathbf{int}; \alpha \vdash y : \beta \Rightarrow \mathbf{int} \qquad \beta \not\subseteq \alpha}{x : \alpha \Rightarrow \mathbf{int}, y : \beta \Rightarrow \mathbf{int}; \alpha \vdash \mathsf{ap}(y; \beta) : \mathbf{int}}\ \text{P}\ \warning}{x : \alpha \Rightarrow \mathbf{int}, y : \beta \Rightarrow \mathbf{int}; \varnothing \vdash \lambda(\_.\mathsf{ap}(y; \beta)) : \alpha \Rightarrow \mathbf{int}}\ \text{C}$$

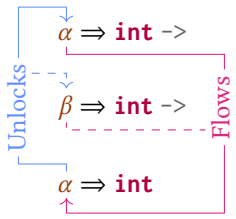

Fig. 4. Shikuma and Igarashi [2008] Dependency Tracking Connective (Top); Usage Example (Bottom)

into the type. The elimination rule Produce checks that the ambient security level ϕ is higher than the dependency set ϕ' of the modality to be eliminated, then concludes at the inner type.

The notation we use here for the modality is suggestive: it is exactly a function with a dependency set as its argument type. Through this lens, the rules for its introduction and elimination mimic exactly those standard for functions, where the former adds its argument as an assumption to the typing environment for its body and the latter checks that the current assumptions suffice for the expected argument. The benefits of viewing this modality as a function are twofold. First, it permits the dependency tracking mechanics at play to be understood as a matter of *access token passing*, illuminating its sensitivity to *flows*. Second, it establishes a firm connection to the *reader monad* for ϕ, which is precisely the *open* modality [Sterling and Harper 2021a]. We review each in turn.

On the first point, consider a function at type $(\alpha \Rightarrow \mathbf{int})$ -> $(\beta \Rightarrow \mathbf{int})$ -> $\alpha \Rightarrow \mathbf{int}$. At first glance, we might guess that its return value depends only on its first argument. Why? Stepping through, we assume two modalities at α and β respectively. We conclude with one at α. Reading the modality itself as a function, this means we *assume* α from the overall return type $\alpha \Rightarrow \mathbf{int}$ and have it available to *unlock* other data where it is expected as an argument. This is only possible for the input modality $\alpha \Rightarrow \mathbf{int}$. For $\beta \Rightarrow \mathbf{int}$ we do not have β as an assumption, so attempting to apply Produce fails to unlock its data and extract the contained **int**. Figure 4 (bottom) shows candidate function *bodies* for each case, starting at the bottom with Consume followed by Produce.

Observe that we are exclusively concerned here with the provenance of various permissions to unlock data. We pass around such access tokens to speak by proxy about *flows*; assuming an α from the return type and having it available to use to unlock an argument at α permits that argument to flow to the return value. That is, the presence of an ambient α denotes flows from data guarded by it. **The nature of this modality is to indicate *where data is moving* by way of tracking the authority to access it.** It is defined by *internalizing access assumptions*, similarly to usual function types internalizing variable typing assumptions. Unlike the previous modality, it involves no notion of the information contained in types. One could return an informationally inert **unit** from some computation, but continue to have to track which permissions were used to compute it because the flow has already occurred. Effects bring this special relevance; consider `print : io ⇒ (string -> unit)`. Failing to track `io` would be tantamount to forgetting potential uses of `print`! Section 6 will revisit this. For now, we confront the second point from before.

Given that both its underlying machinery and practical uses differ subtly but fundamentally from those of the DCC modality, we now have reason to suspect that the modality here should be cast as something quite distinct from the latter. It is in reality the *open modality*, introduced by Rijke et al. [2020] as the sibling to the closed modality. The open modality is defined as a function with its antecedent containing dependencies ϕ, a construction familiar to functional programmers as the reader monad for ϕ. Besides Shikuma and Igarashi [2008], the rules presented here are also analogous to those for the open modality in Sterling and Harper [2021b], a fact not observed there or elsewhere. The open modality makes further appearances in Gouni et al. [2025] as discussed in Section 1, in Liu et al. [2024, §3.2] where it is again cast as a variant of the DCC monad inspired by Shikuma and Igarashi [2008], and in Sterling and Harper [2022] which reviews it as distinct from the closed modality in an information flow setting. The lattermost notes the algebraic relationship between the two, but focuses on the closed modality and its semantic insights towards DCC. In this work we afford each equal standing, situating both within the same system to take advantage of the rich information flow reasoning springing forth from their *joint* semantics.

Section 1 gives one exposition of the divide between flows and information, in terms of dependencies and anti-dependencies, and we have just given another here, in terms of differing information flow modalities. We unify these perspectives in the next section by way of formally introducing our approach to information flow, which features both modalities. Though both act to track dependencies when apart, when combined one is flipped into the anti-dependencies view. Apart from the two throughlines we have seen, there is a *hybrid* modality within the literature which can be understood as combining aspects of both. This is briefly summarized for thoroughness in Appendix A, but will not be relevant to our purposes here. We look now to our type system.

3 Syntax and Typing

We set up our core language, *parametric information flow*, in the Call-By-Push-Value framework of Levy [2003] extended with the just-discussed modalities.[2] We use its explicit treatment of proof-theoretic polarity to further distinguish each modality. We then discuss their *joint* semantics at a theoretical level, leading into its practical implications in the section which follows.

3.1 Information, or Values

Call-By-Push-Value bifurcates its connectives between *values* and *computations*. We start with the values first; their syntax and rules are shown in Figure 5. We have dependency variables $\alpha_1, \alpha_2, \ldots$ and term variables $x_1, x_2, \ldots$, with the former collected into dependency sets ϕ. The value types A are all *positive* types in a proof-theoretic sense, with the slight complication of the *shift* connective $\mathsf{U}(\underline{B})$. $\underline{B}$ will range over the *computation* types, to be discussed in the next subsection. The form of the value typing judgement is $\Delta; \Gamma; \phi \vdash v : A$, read as "under in-scope dependency variables Δ, in-scope term variables Γ, and ambient dependencies ϕ, the value v has type A."

The rule T-Var for typing variables is standard for the polarized setting, stating that variables must be at value type. It permits this typing at any ambient dependencies ϕ. The introduction rules for `unit`, sums $A_1 + A_2$, and positive products $A_1 \otimes A_2$ likewise follow largely by convention, threading through the ambient dependencies as needed. For sums, we check the well-scopedness $\Delta \vdash A$ of the type not mentioned by the typing assumption in the premise. This is to prevent the appearance of dependency variables $\alpha \notin \Delta$. For products, each projection is checked at the same ambient dependencies ϕ. This may seem an onerous restriction, but is mediated by a *subsumption* rule T-Sub, shown in Figure 5. The notation $\phi'_\bigcirc \vdash \phi_\bigcirc$ can be read as $\phi' \supseteq \phi$; Section 3.3.2 will use it to

[2]The Lean mechanization of our type system uses intrinsically well-typed-and-scoped syntax with binding via De Bruijn indices, approximated here using explicit scoping directives and capture-avoiding substitution over named variables.

$$\text{Dependency Vars } \alpha_1, \alpha_2, \ldots \in \Delta \qquad \text{Vars } x_1, x_2, \ldots \in \Gamma$$

$$\begin{aligned}
\text{Dependencies } \phi &::= \epsilon \mid \alpha; \phi \\
\text{Value Types } A &::= \mathtt{unit} \mid A_1 + A_2 \mid A_1 \otimes A_2 \mid \exists(\alpha.A) \mid \bullet_\phi(A) \mid \mathsf{U}(\underline{B}) \\
\text{Values } v &::= x \mid \langle\rangle \mid \mathtt{l} \cdot v \mid \mathtt{r} \cdot v \mid \langle v_1, v_2 \rangle \mid \mathtt{pack}[\phi](v) \mid \mathtt{seal}[\phi'](v) \mid \mathtt{susp}(e)
\end{aligned}$$

$$\textsc{T-Var}\ \frac{}{\Delta; \Gamma, x : A; \phi \vdash x : A} \qquad \textsc{T-Unit}\ \frac{}{\Delta; \Gamma; \phi \vdash \langle\rangle : \mathtt{unit}} \qquad \textsc{T-InjL}\ \frac{\Delta; \Gamma; \phi \vdash v : A_1 \qquad \Delta \vdash A_2}{\Delta; \Gamma; \phi \vdash \mathtt{l} \cdot v : A_1 + A_2}$$

$$\textsc{T-InjR}\ \frac{\Delta; \Gamma; \phi \vdash v : A_2 \qquad \Delta \vdash A_1}{\Delta; \Gamma; \phi \vdash \mathtt{r} \cdot v : A_1 + A_2} \qquad \textsc{T-Pair}\ \frac{\Delta; \Gamma; \phi \vdash v_1 : A_1 \qquad \Delta; \Gamma; \phi \vdash v_2 : A_2}{\Delta; \Gamma; \phi \vdash \langle v_1, v_2 \rangle : A_1 \otimes A_2}$$

$$\textsc{T-Pack}\ \frac{\Delta; \Gamma; \phi \vdash v : [\phi'/\alpha]A \qquad \Delta \vdash \phi'}{\Delta; \Gamma; \phi \vdash \mathtt{pack}[\phi'](v) : \exists(\alpha.A)} \qquad \textsc{T-Seal}\ \frac{\Delta; \Gamma; \phi \vdash v : A \qquad \Delta \vdash \phi'}{\Delta; \Gamma; \phi \vdash \mathtt{seal}[\phi'](v) : \bullet_{\phi'}(A)} \qquad \textsc{T-Susp}\ \frac{\Delta; \Gamma; \phi \vdash e : \underline{B}}{\Delta; \Gamma; \phi \vdash \mathtt{susp}(e) : \mathsf{U}(\underline{B})}$$

$$\textsc{T-Sub}\ \frac{\Delta; \Gamma; \phi \vdash v : A \qquad \phi'_\bigcirc \vdash_{(\supseteq)} \phi_\bigcirc \qquad \Delta \vdash \phi'}{\Delta; \Gamma; \phi' \vdash v : A'} \qquad \textsc{T-Sub-Type}\ \frac{\Delta; \Gamma; \phi \vdash v : A \qquad A \subseteq A' \qquad \Delta \vdash A'}{\Delta; \Gamma; \phi \vdash v : A'}$$

Fig. 5. Connectives for Values

explain the semantics of our system. This rule allows, for instance, the lower of two projections with differing security levels to be heightened as needed. We scope check the heightened dependency set ϕ'. $\mathsf{U}(\underline{B})$ is the only *negative* connective among the value types. T-Susp introduces it by suspending a computation e, shifting it to a value. Its ambient dependencies ϕ flow to the conclusion untouched.

The first non-standard connective, fundamentally concerning information flow, is $\bullet_\phi(A)$. It is introduced by the rule T-Seal, which we have already encountered. This connective is a variant of the *closed* modality, acting analogously to the Seal rule from Figure 3; its syntax has been updated to follow Sterling and Harper [2021a]. Subsumption mechanics are afforded to it by the rule T-Sub-Type in Figure 5, whose premise $A \subseteq A'$ defines a standard subtyping relation on types. The case of this relation for the closed modality is shown to the right, with $\phi_\bullet \vdash \phi'_\bullet$ read as $\phi \subseteq \phi'$. Section 3.3.2 will illuminate the notation. T-Sub-Type scope checks the heightened type containing ϕ'. A further information flow connective is the existential quantifier $\exists(\alpha.A)$ over dependencies. T-Pack checks the implementation of an existential at the implementing dependency set ϕ' before obscuring ϕ' from view in the conclusion, where it is hidden behind the existential variable α. This will provide *downgrading* behavior, as recognized by Gouni et al. [2025]. Section 4 will explain by example.

$$\textsc{Sub-Closed}\ \frac{\phi_\bullet \vdash_{(\subseteq)} \phi'_\bullet \qquad A \subseteq A'}{\bullet_\phi(A) \subseteq \bullet_{\phi'}(A')}$$

Existentials can be approximated through universals [Girard et al. 1989, §11.3.5], so prior work [Gouni et al. 2025] does not treat them explicitly. (Observe that universals have been in use towards dependency polymorphism since Section 1, providing the precursors for downgrading.) In our setting, however, existentials exhibit sufficiently interesting behavior to invite first-class treatment.

$$\textsc{S-Unit}\ \frac{}{\mathtt{unit} \geq \phi} \qquad \textsc{S-Pair}\ \frac{A_1 \geq \phi \qquad A_2 \geq \phi}{A_1 \otimes A_2 \geq \phi} \qquad \textsc{S-Sealed}\ \frac{\phi_\bullet \vdash_{(\subseteq)} \phi'_\bullet}{\bullet_{\phi'}(A) \geq \phi} \qquad \textsc{S-Unsealed}\ \frac{A \geq \phi}{\bullet_{\phi'}(A) \geq \phi} \qquad \textsc{S-Thunk}\ \frac{\underline{B} \geq \phi}{\mathsf{U}(\underline{B}) \geq \phi}$$

Fig. 6. Sealing Judgement for Value Types

To elaborate, recall *sealing* from Section 2.1. Figure 6 defines the sealing judgement for the value types, which will be the interesting half of its definition. As we saw in Figure 3, `unit` is always sealed due to conveying no information; likewise for pairs with sealed projections. Besides S-Unit, the other base case for this judgement is provided by S-Sealed, which states that $\bullet_{\phi'}(A) \geq \phi$ is satisfied when ϕ is included in ϕ'. S-Unsealed gives the case for the closed modality where its ϕ' does not satisfy the inclusion, instead recursing onto the inner type. S-Thunk recurses onto its contents with the sealing judgement at computation type, alleged to be straightforward.

So where lies the interesting portion of sealing for values? Exactly in the cases which have been *omitted*: those for sums and existentials. There is no case of sealing for sums because a term of type $A_1 + A_2$ always has at least two inhabitants, contributed by the left and right injections, which inevitably reveals information. As for **bool** in Figure 3—which can be defined as `unit + unit`—the only way for a sum to exist within a type satisfying the sealing judgement is to be placed under the closed modality. The case (or lack of it) for existentials is less clear, but analogous reasoning applies. Readers may recall that an existential type can be interpreted as a sum over a family of types; this is exactly the reason for their omission from sealing. In particular, the implementing dependency set ϕ' in $\mathsf{pack}[\phi'](e)$ may differ between different terms of type $\exists(\alpha.A)$. However, one might correctly protest that there is no way to *use* an existential so as to reveal its implementation. The preceding behavior is indeed *not* a true semantic necessity but a concession made for metatheoretic simplicity. This will allow us to clarify our core results and draw closer comparisons with prior work, some of which exhibits the same limitation. Section 5 will offer further justification on this point.

Only the value types lack certain cases of sealing; all computation types will possess routine definitions for the sealing judgement. As embodied by sealing, information is defined by an analysis of possible inhabitants; it is no coincidence that this parallels the usual characterization of positive types by their methods of introduction. This is why $\bullet_\phi(A)$ exists among the values: they are the only class of types worth sealing! In other words, only value types possess even the *potential* to represent information—to be unable to be sealed—and so the connective for tracking information is conferred to them. This indicates that **information is fundamentally a matter of values.** It is reasonable to wonder, then, what information flow reasoning gains from computations.

3.2 Flows, or Computations

We show the rules for computation types $\underline{B}$ in Figure 7. These comprise the *negative* connectives—save again for the shift connective $\mathsf{F}(A)$ which forms the sibling to $\mathsf{U}(\underline{B})$ and is positive, inserting values into the computation domain. They follow the judgement $\Delta; \Gamma; \phi \vdash e : \underline{B}$, read as "under in-scope dependency variables Δ, in-scope term variables Γ, and ambient dependencies ϕ, the expression e has type $\underline{B}$." The new rules for computation connectives live alongside the elimination rules for value types in the computation domain because the latter also eliminate into motives at computation type. The former set of rules are highlighted in **red** and the latter in **blue**.

Starting with function types $A \to \underline{B}$, we first show the relevant case of sealing below. The argument type A is irrelevant as far as sealing is concerned because one can ultimately only leak information through the return type $\underline{B}$. So it recurses straightforwardly onto this inner type; the

$$\text{Computation Types } \underline{B} ::= A \to \underline{B} \mid \forall(\alpha.\underline{B}) \mid \bigcirc_\phi(\underline{B}) \mid \mathsf{F}(A)$$
$$\text{Expressions } e ::= \lambda(x.e) \mid \mathsf{ap}(e_1; e_2) \mid \Lambda(\alpha.e) \mid e[\phi] \mid \mathtt{consume}(e) \mid \mathtt{produce}(e) \mid \mathtt{ret}(v)$$
$$\mid \mathtt{bind}\ e_1\ \mathtt{in}\ x.e_2 \mid \mathtt{case}\ v\ \mathtt{of}\ \{\ \mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\ \} \mid \mathtt{force}(v)$$
$$\mid \mathtt{split}\ v\ \mathtt{into}\ \langle x_1, x_2\rangle\ \mathtt{in}\ e \mid \mathtt{open}\ v\ \mathtt{in}\ \alpha, x.e \mid \mathtt{unseal}\ v\ \mathtt{in}\ x.e$$

T-Ret
$$\frac{\Delta; \Gamma; \phi \vdash v : A}{\Delta; \Gamma; \phi \vdash \mathtt{ret}(v) : \mathsf{F}(A)}$$

T-Consume
$$\frac{\Delta; \Gamma; \phi'\phi \vdash e : \underline{B}}{\Delta; \Gamma; \phi' \vdash \mathtt{consume}(e) : \bigcirc_\phi(\underline{B})}$$

T-Produce
$$\frac{\Delta; \Gamma; \phi' \vdash e : \bigcirc_\phi(\underline{B})}{\Delta; \Gamma; \phi\,\phi' \vdash \mathtt{produce}(e) : \underline{B}}$$

T-Bind
$$\frac{\Delta; \Gamma; \phi \vdash e_1 : \mathsf{F}(A) \qquad \Delta; \Gamma, x : A; \phi \vdash e_2 : \underline{B}}{\Delta; \Gamma; \phi \vdash \mathtt{bind}\ e_1\ \mathtt{in}\ x.e_2 : \underline{B}}$$

T-Lam
$$\frac{\Delta; \Gamma, x : A; \phi \vdash e : \underline{B}}{\Delta; \Gamma; \phi \vdash \lambda(x.e) : A \to \underline{B}}$$

T-Force
$$\frac{\Delta; \Gamma; \phi \vdash v : \mathsf{U}(\underline{B})}{\Delta; \Gamma; \phi \vdash \mathtt{force}(v) : \underline{B}}$$

T-Ap
$$\frac{\Delta; \Gamma; \phi \vdash e : A \to \underline{B} \qquad \Delta; \Gamma; \phi \vdash v : A}{\Delta; \Gamma; \phi \vdash \mathsf{ap}(e; v) : \underline{B}}$$

T-Unseal
$$\frac{\Delta; \Gamma; \phi \vdash v : \bullet_{\phi'}(A) \qquad \Delta; \Gamma, x : A; \phi \vdash e : \underline{B} \qquad \underline{B} \geq \phi'}{\Delta; \Gamma; \phi \vdash \mathtt{unseal}\ v\ \mathtt{in}\ x.e : \underline{B}}$$

T-All
$$\frac{\Delta, \alpha; \Gamma; \phi \vdash e : \underline{B} \qquad \Delta \vdash \phi}{\Delta; \Gamma; \phi \vdash \Lambda(\alpha.e) : \forall(\alpha.\underline{B})}$$

T-Split
$$\frac{\Delta; \Gamma; \phi \vdash v : A_1 \otimes A_2 \qquad \Delta; \Gamma, x_1 : A_1, x_2 : A_2; \phi \vdash e : \underline{B}}{\Delta; \Gamma; \phi \vdash \mathtt{split}\ v\ \mathtt{into}\ \langle x_1, x_2\rangle\ \mathtt{in}\ e : \underline{B}}$$

T-Inst
$$\frac{\Delta; \Gamma; \phi \vdash e : \forall(\alpha.\underline{B}) \qquad \Delta \vdash \phi'}{\Delta; \Gamma; \phi \vdash e[\phi'] : [\phi'/\alpha]\underline{B}}$$

T-Case
$$\frac{\Delta; \Gamma; \phi \vdash v : A_1 + A_2 \qquad \Delta; \Gamma, x_1 : A_1; \phi \vdash e_1 : \underline{B} \qquad \Delta; \Gamma, x_2 : A_2; \phi \vdash e_2 : \underline{B}}{\Delta; \Gamma; \phi \vdash \mathtt{case}\ v\ \mathtt{of}\ \{\ \mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\ \} : \underline{B}}$$

T-Open
$$\frac{\Delta; \Gamma; \phi \vdash v : \exists(\alpha.A) \qquad \Delta, \alpha; \Gamma, x : A; \phi \vdash e : \underline{B} \qquad \Delta \vdash \underline{B} \qquad \Delta \vdash \phi}{\Delta; \Gamma; \phi \vdash \mathtt{open}\ v\ \mathtt{in}\ \alpha, x.e : \underline{B}}$$

Fig. 7. Connectives for Computations

S-Arrow
$$\frac{\underline{B} \geq \phi}{A \to \underline{B} \geq \phi}$$

remaining cases of sealing for computation types proceed analogously and straightforwardly, as promised. The introduction and elimination rules for function types—T-Lam and T-Ap—likewise present no surprises. Note that in the case of T-Ap, the function expression e and its argument v must be at the same ambient dependencies ϕ. Discrepancies between them are, as previously discussed for products, negotiated via a subsumption rule analogous to T-Sub in Figure 5, operating instead on the computation judgement. The situation is identical for all future rules concerning multiple typed values or expressions, so we will make no further note of it.

The rule T-All introduces universal quantification over dependencies $\forall(\alpha.\underline{B})$, binding the α in scope for the typing premise under a universal quantifier and removing it from the dependency

variable environment in the conclusion as it does so. The auxiliary premise $\Delta \vdash \phi$ ensures the ambient dependency set ϕ of the typing premise does not mention the just-bound dependency α, preventing α from appearing in the concluding ambient dependencies where it is no longer in scope. The elimination rule T-Inst substitutes the instantiating dependency set ϕ' for the bound dependency variable α within the inner type $\underline{B}$ of the quantifier. ϕ' is checked for well-scopedness.

The rules T-Consume and T-Produce for introducing and eliminating $\bigcirc_\phi(B)$ look familiar: they are the rules from Figure 4 for our variant of the open modality! As in the case of the closed modality we have updated its syntax, replacing the function notation within the connective [Sterling and Harper 2021a] and its terms [Gouni et al. 2025] in favor of that used by prior work, but the structure of both rules remains intact.

Sub-Open
$$\frac{\phi'_\bigcirc \vdash_{(\supseteq)} \phi_\bigcirc \qquad \underline{B} \subseteq \underline{B}'}{\bigcirc_\phi(\underline{B}) \subseteq \bigcirc_{\phi'}(\underline{B}')}$$

Also as for the closed modality it enjoys subsumption, given by a rule analogous to T-Sub-Type from Figure 5 but operating instead on computation types. We give the relevant case here; it allows its indexing dependency set ϕ to be raised to any superset ϕ', so is structurally identical to that for the closed modality. We use different notation this time to reflect the contravariance owed to the open modality due to its reading as a function from Section 2.2, again to be explained shortly in Section 3.3.2. Comparing Produce and T-Produce, rather than checking that the ambient dependencies in the conclusion contain the dependencies guarding the modality, as in the former, the latter releases the dependencies ϕ guarding the modality into the concluding ambient set. In both cases, the modality may only be unlocked in the presence of a sufficiently high concluding ambient dependency set, guaranteeing its use is tracked. The symmetry of our setup for the open modality exposes its proof-theoretic harmony—introduction followed by elimination is type-preserving.

Recall that the ambient dependencies denote the flows in play, under the view that they represent permissions to unlock and use data—guarded by other open modalities—within the current computation. The only connective whose rules directly affect the ambient dependencies is the open modality itself, which *lives entirely within the realm of computations.* Of these rules, observe that T-Produce uses the ambient dependencies—or permissions—to extract data. Its invocation can be seen as the definitive point where the flow occurs. It is also the eliminator for the open modality, which being a negative type is exactly characterized by it; the open modality is *defined by causing flows.* This indicates that **flows are fundamentally a matter of computations,** which should come as no surprise given their inherent connection to the dynamic behavior of programs.

A couple complications threaten this perspective. First, the subsumption rule T-Sub from Figure 5 also modifies the ambient dependencies. However, this modification is vacuous, raising the ambient security level as needed for compatibility between subexpressions. We will lend formal support to the idea that the ambient security level only *necessarily* changes for computations in Section 5, where we show subsumption to be admissible under a semantics which behaves as such. Second, why are ambient dependencies present in the value judgement at all? The permissions they represent cannot be applied within the value judgement to realize flows. These are indeed rather *flows-to-be*, enabling flows when the value becomes part of a computation. Section 3.3 will hint, and Section 5 will formalize, that knowing about flows-to-be at the value level bridges our two modalities. Only ambient dependencies within the computation judgement denote active flows.

The remaining rules closely follow their standard formulations, threading through ambient dependencies as needed. Of note, T-Open eliminates existential quantifiers, binding within e the existential dependency variable α and a variable at type A mentioning it. The type $\underline{B}$ and ambient dependencies ϕ of this expression are checked to be well-scoped under a dependency variable environment not containing α. This means one must eventually downgrade or otherwise avoid mentioning the existentially quantified dependency variable in the result of any computation using it. Section 4 will exploit this restriction towards practical programming.

As promised, our type system features both the open and closed variants of dependency tracking. We have furthered here the characterization of information versus flows initiated in the previous section, aligning each flavor of dependency tracking with the value-computation duality of Call-By-Push-Value. Following Levy [2003], *a value is whereas a computation does*. While the former views programs as inert data, the latter is intertwined with their dynamic behavior. The insights in this section give rise to the parallel slogan ***information is whereas a flow does*; the distinction between the two mirrors the value-computation duality.** Having said this, we have thus far left by the wayside the characterization of information versus flows via *dependencies* and *anti-dependencies* from the introduction. We remedy this now, unifying it with that just discussed.

3.3 Information and Flows Simultaneously

We have just shown that the open and closed modalities respectively capture reasoning regarding information versus flows. However, Section 1 teased apart these same ideas via *dependencies* and *anti-dependencies*. The *joint* semantics of the open and closed modalities give rise to a consistent resolution of this discrepancy. Afterwards, we comment on the distinction between the standard formulation of the open and closed modalities and that given by the preceding exposition.

We hinted at the nature of the reconciliation between these differing views earlier, with Section 1 stating that Gouni et al. [2025] exploit the *flows* or *dependencies* fragment, and Section 2.2 relating their information flow connective to the open modality. So **the open modality is associated to reasoning about dependencies,** with the remaining possibility being that **the closed modality is linked to anti-dependencies.** In Section 1 the former was written $[\alpha\ \beta\ \ldots]$ and the latter $![\alpha\ \beta\ \ldots]$. But why this grouping—of flows, dependencies, and the open modality on the one side, and information, anti-dependencies, and the closed modality on the other?

The answer lies in the interaction between our modalities. In particular, they satisfy (roughly) the equation to the right, adapted from Sterling and Harper [2021b], which states that all terms sealed at ϕ' are equivalent under ambient dependencies ϕ. The effect of the premise $\phi_{\circ} \vdash \phi'_{\bullet}$—which we will fully elucidate next—is to check that ϕ and ϕ' have *overlapping elements*. In other words, their intersection is non-empty. Recall from Section 2.1 that ϕ' in $A \geq \phi'$ refers to dependencies situated within the closed modality, which internalizes this judgement, and from Section 2.2 that the ambient dependencies ϕ can be thought of as connected to the open modality, being internalized by it. The meaning of the equation, then, is that data sealed within the closed modality at some set of dependencies is rendered *indistinguishable* under the purview of an ambient set of dependencies which overlaps with it. The dependencies within the closed modality therefore regulate against their mention in the ambient dependencies, or the open modality; values at the former are *anti-dependent* on computations at the latter. We will continue to refer to αs as dependencies, identifying anti-dependencies by placement within the closed modality. Indistinguishability corresponds to the central soundness property, amounting to faithfully tracking both open and closed: *data at non-overlapping type cannot depend on data at overlapping type*. Section 5 will need to enrich the preceding to account for downgrading.

$$\frac{A \geq \phi' \qquad \phi_{\circ} \vdash \phi'_{\bullet}}{\Delta\ \Gamma\ \phi \vdash v_1 \equiv v_2 : A}$$

In summary, the action of the closed modality is to protect data from being accessed in settings where one has assumed permissions, and therefore indicated flows, adverse to it. **While open represents flows, closed hides information from them.** Note that overlapping, or running into a *conflict* in the nomenclature of Section 1.1.2, does not necessarily lead to a type error. The stated property is merely one of separation, not of well-formedness: it is perfectly valid to have a value at type $\bullet_{\phi}(A)$ with ambient dependencies ϕ. It simply cannot be depended upon by any non-conflicted parts of the program. A practical implementation invites producing type errors, as implied by the examples in Figure 2, but how this might be done is not prescribed theoretically.

$$\frac{P : \mathsf{Prop} \qquad A : \mathsf{Type}_{\mathbb{V}}}{\bullet_P(A) : \mathsf{Type}_{\mathbb{V}}} \qquad \frac{P : \mathsf{Prop} \qquad \underline{B} : \mathsf{Type}_{\mathbb{C}}}{\circ_P(\underline{B}) : \mathsf{Type}_{\mathbb{C}}}$$

$$\begin{aligned} [\![\alpha]\!] &\triangleq p \text{ where } p = \mathcal{P}[\alpha] \\ [\![\epsilon]\!] &\triangleq (\top, \bot) \\ [\![\alpha; \phi]\!] &\triangleq ([\![\alpha]\!] \wedge [\![\phi]\!].\pi_1, [\![\alpha]\!] \vee [\![\phi]\!].\pi_2) \end{aligned}$$

Fig. 8. Signatures of the Standard Modalities (Left); Interpretation of Dependencies (Right)

We bring this section to a close by briefly discussing the issues of composition order and the relationship of our modalities to the standard ones.

3.3.1 On Composition Order. What we have just discussed corresponds to the order of composition $[\alpha\ \beta\ \ldots]\ ![\alpha\ \beta\ \ldots]\ \mathtt{t}$ seen in Section 1, with anti-dependencies at the closed modality situated within dependencies at the open modality. What about the reverse order? This turns out to be inert. To begin, view the examples of the introduction under a standard call-by-value semantics. The translation from call-by-value into the explicitly polarized core language presented in this section proceeds routinely via the procedure given by Levy [2003, §2.7.1].

In particular, setting $\phi = \alpha; \beta; \ldots$, we have $\langle\!\langle [\alpha\ \beta\ \ldots]\ \mathtt{t} \rangle\!\rangle \rightsquigarrow \mathsf{U}(\circ_\phi(\mathsf{F}(\langle\!\langle \mathtt{t} \rangle\!\rangle)))$ for the translation of the open modality, and $\langle\!\langle ![\alpha\ \beta\ \ldots]\ \mathtt{t} \rangle\!\rangle \rightsquigarrow \bullet_\phi(\langle\!\langle \mathtt{t} \rangle\!\rangle)$ for the closed. The call-by-value open modality $[\alpha\ \beta\ \ldots]$ is surrounded by the shift connectives U and F due to being a negative type, and the call-by-value closed modality $![\alpha\ \beta\ \ldots]$ is left untouched because it is a positive type. Translations for all other types are given analogously. The translation for the reverse composition is then $\langle\!\langle ![\alpha\ \beta\ \ldots]\ [\alpha\ \beta\ \ldots]\ \mathtt{t} \rangle\!\rangle \rightsquigarrow \bullet_\phi(\langle\!\langle [\alpha\ \beta\ \ldots]\ \mathtt{t} \rangle\!\rangle) \rightsquigarrow \bullet_\phi(\mathsf{U}(\circ_\phi(\mathsf{F}(\langle\!\langle \mathtt{t} \rangle\!\rangle))))$.

Reifying the reverse composition in Call-By-Push-Value allows us to see why it poses no danger, even if someone mistakenly assumed it would be considered a conflict and therefore covered by the soundness guarantee. The closed modality at ϕ does not *directly* contain the open modality at ϕ at the end of the translation, but rather encapsulates it *inside a suspension* U. Accordingly, these open dependencies ϕ and the flows they represent are not ambient but suspended with respect to the closed modality, so are not considered conflicting! It is only once the inner computation runs and exposes its dependencies to the closed modality that any danger arises of information protected under anti-dependencies participating in the flows denoted by conflicting ambient dependencies.

3.3.2 Comparing to the Standard Modalities. In certain cases, we have been careful to refer to our connectives $\bullet_\phi(A)$ and $\circ_\phi(\underline{B})$ as *variants* of the usual closed and open modalities. Those in our setting have been modified to increase specification readability and confer modular downgrading behavior. The formation rules for the standard modalities might be written as on the left of Figure 8.

Each takes a proposition P followed by a value or computation type A or $\underline{B}$ and returns $\bullet_P(A)$ or $\circ_P(\underline{B})$. Working with *propositions* rather than sets of dependencies is the essence of the difference. The meaning of $\circ_P(\underline{B})$, recalling its interpretation as a function, is to say that we have $\underline{B}$ under assumption P. And the meaning of $\bullet_P(A)$ is that when P holds its information becomes indistinguishable, or $\bullet_P(A) \cong \mathtt{unit}$. For nomenclature, when $\circ_P(\underline{B}) \cong \mathtt{unit}$, we say $\underline{B}$ is $\circ_P$-connected. Figure 8 shows to the right the propositional structure of a ϕ to help clarify our connection to this setup. Assume a logic with the usual truth values $\top$ and $\bot$, in addition to a set of *included middles* $\mathcal{P}$—extra truth values neither $\top$ nor $\bot$. We generate one such value $p \in \mathcal{P}$ to interpret each in-scope dependency α. We then interpret dependency sets ϕ as a pair where the first projection is a sequence of conjunctions over its elements and the second a sequence of disjunctions. As an example, where $[\![\alpha]\!] \triangleq p_1$ and $[\![\beta]\!] \triangleq p_2$, we have $[\![\alpha; \beta; \epsilon]\!] \triangleq (p_1 \wedge p_2, p_1 \vee p_2)$.

We now define our modalities as the standard ones precomposed with the projection maps: $\circ \circ \pi_1$ and $\bullet \circ \pi_2$. The first premise of their formation rules changes to take a *pair* of propositions

Prop × Prop representing a ϕ. This means $\bigcirc_\phi(\underline{B})$ is effectively indexed by a conjunction, and $\bullet_\phi(A)$ by a disjunction. **The notations $\phi_\bigcirc$ and $\phi_\bullet$ view the dependencies in ϕ as logical atoms and conjunct or disjunct them together,** corresponding to taking either the first or second projections. Their entailment is then as usual. We choose this rather than a more straightforward presentation of each ϕ as a set because it highlights a number of desirable properties. T-Sub and the case of subsumption for $\bigcirc_\phi(\underline{B})$ are contravariant with respect to $\phi_\bigcirc$, as should be expected of judgemental assumptions and function argument types, while $\bullet_\phi(A)$ is covariant for $\phi_\bullet$. Further, the indistinguishability condition $\phi_\bigcirc \vdash \phi_\bullet$ encodes the desired overlapping property by checking that a string of conjunctions entails one of disjunctions, while mirroring the standard one stating that the ambient assumptions entail a closed proposition. **We respectively refer to $\bigcirc_\phi(\underline{B})$ and $\bullet_\phi(A)$ as the *quasi-open* and *quasi-closed*[3] modalities to reflect their non-standard formulation,** particularly when their divergence from the standard modalities is relevant.

Specification simplicity ranks among the practical advantages gained from this divergence. With the standard modalities we have $\bigcirc_P(\bigcirc_Q(\underline{B})) \cong \bigcirc_{P\wedge Q}(\underline{B})$ by currying and $\bullet_P(\bullet_Q(A)) \cong \bullet_{P\vee Q}(A)$ straightforwardly from Rijke et al. [2020, Ex. 1.8] and Univalent Foundations Program [2013, Lem. 8.5.9]. As such, we would need to support both conjunction and disjunction operators within our information flow specifications, precluding the flat sets ϕ we currently use. This semantics furthermore turns out to grant downgrading behavior, because the base case for ϵ in Figure 8 evades the indistinguishability condition; Section 5 will explain. Finally, note that with only two truth values $\top$ and $\bot$, we have $\bigcirc_\top(\underline{B}) \cong \underline{B}$ and $\bullet_\top(A) \cong \mathtt{unit}$, and $\bigcirc_\bot(\underline{B}) \cong \mathtt{unit}$ and $\bullet_\bot(A) \cong A$. The included middles allow us to mention the same dependency α in both modalities without trivializing either. This is a technique used by prior work [Grodin et al. 2026, §2.1.1], originating from the anti-classical foundations of the modalities. Section 5 will show the metatheory to be organized around these extra truth values, and Section 6 will discuss further formal consequences of the preceding designs. We consider now their programming consequences.

4 Examples

Theoretical concerns have received much attention; we now seek to reap the practical benefits of the preceding intuitions surrounding duality in information flow. The closed modality will be shown to provide *robustness* reasoning, leading to a unifying view on *robust declassification* and *transparent endorsement* from the literature. We finish by showing how to program with conflicts *modularly*. In order to avoid syntactic clutter from the $\mathsf{U}(-)$ and $\mathsf{F}(-)$ shifts of the polarized setting, we work here in a call-by-value language derived from it; this follows the translation from Section 3.3.1. When we use monospace type notation as in **`t`** we refer to call-by-value types.

4.1 Robustness and the Closed Modality

The closed modality can be seen as fundamentally providing reasoning about *robustness* properties. The original definition of robustness, as explored by later work [Bugliesi et al. 2011; Gordon and Jeffrey 2001; Mackay et al. 2022; Swasey et al. 2017; Zdancewic and Myers 2001], is as follows.

> The degree to which a system or component can function correctly in the presence of invalid or stressful environmental conditions. IEEE Standards Board [1990]

An integer under the closed modality as in `![untrusted] int` can be understood as stating that it will render itself inaccessible within an environment where it is unable to behave according to its intended semantics. Namely, one where computations involving data at `[untrusted]` are in play. More broadly, the closed modality defines the permissible environments for the data it contains. The cumulative shape of these environmental restrictions gives the robustness properties of the

[3] No relevance to the modality of the same name of Schultz and Spivak [2019].

```
1 let password : [secret] string = { "takver" }
2 let user_input : [untrusted] string = { "saio pae" }
3 let neural_net : ![untrusted] (string -> bool) = seal ...
```

```
4 let _ : [untrusted] ![untrusted] bool =
5     { unseal neural_net as f in seal (f user_input) } ⚠
6 let _ : [secret] ![untrusted] bool =
7     { unseal neural_net as f in seal (f password) }
```

Fig. 9. Revisiting Adversarial Input with Fully Explicit Syntax (see §4.1.1 for notation)

program, explicating the contexts in which its various fragments may be used. We wish to advance here the perspective that **anti-dependencies speak about robustness against dependencies.**

4.1.1 Towards Integrity. Figure 2 already reviewed a case of such robustness restrictions towards integrity by excluding the `neural_net` function, vulnerable to adversarial inputs, from `[untrusted]` environments. Figure 9 revisits this example with the hindsight of the full formalism behind us, transforming it to use more explicit syntax to expose the underlying constructs used. The first line now wraps its string `"takver"` in `{ ... }`, which provides notation for the `susp(consume(−))` corresponding to the open modality in its type; we do the same on the next. The definition of `neural_net` on the third line now begins with a **`seal`** corresponding to the occurrence of the closed modality in its type, presumably followed by a function implementation.

Moving to the usages of these variables in the bottom half, as before we have an expression at `[secret] ![untrusted] bool` on line 6. However, its implementation looks rather more complex. Stepping into the `{ ... }` corresponding to `[secret]`, we first **`unseal`** `neural_net` to extract the inner component of the **`seal`**, binding it as `f`. We then call this function on the same argument as before, re**`seal`**ing the result. Note that there is no syntax deployed for invoking `produce(force(−))` on `password`. When data at the open modality appears within `{ ... }`, we automatically do so. The *idiom brackets* of McBride and Paterson [2008] provide the inspiration for this mechanic, exploiting the visual boundary given by the brackets to capture all usages of the open modality within. The erroneous usage on line 4 once again has the same type as before, and its implementation follows analogously. After unsealing `neural_net` at `![untrusted]` we call it on `user_input` at `[untrusted]` and seal the result, producing an error.

```
4 let _ : ![untrusted] [untrusted] bool =
5     unseal neural_net as f in seal { f user_input }
```

Note that if we had written the brackets inside the **`seal`** as above, this would no longer error. As mentioned in Section 3.3.1, the suspension associated to `[untrusted]` renders this inert. The sealed application expression for `f` has not necessarily been thrust into the presence of data violating its robustness requirements—`user_input`—because the brackets suspend the inner computation and capture its dependencies before reaching **`seal`**. Observe that one may still interact with this expression without introducing its invalid final result by, say, **`unseal`**ing it without using `produce(force(−))` on the contents. This is not the case for the original variant of lines 4 and 5.

Surprisingly, raising an error here is a choice left to the implementation, not a matter of soundness. The examples in this section will return errors upon a conflict being detected in the type of a top-level definition, which is a matter of checking the indistinguishability condition from Section 3.3. A practical implementation may choose to return errors more proactively in order to place them

```
1 let make_user_profile : ![secret] (int * string -> data) = seal ...
```

```
2 let _ : [untrusted] ![secret] data =
3     { unseal make_user_profile as f in seal (f 0 user_input) }
4 let _ : [secret] ![secret] data =
5     { unseal make_user_profile as f in seal (f 0 password) } ⚠
```

Fig. 10. Excluding Secrets from Public Data

closer to the offending machinery. Whatever is done, the conflict is guaranteed to be statically tracked—and therefore the user to be informed ahead-of-time—due to soundness. On the current example soundness says the following, which we will substantiate fully in the next section.

Property 4.1 (Robustness, informally)**.** *If we have expressions* $\alpha; x : ![\alpha]\ \mathbf{t}; \varnothing \vdash v : [\alpha]\ \mathbf{t'}$ *and* $\alpha; \varnothing; \alpha \vdash v_1, v_2 : ![\alpha]\ \mathbf{t}$ *then* $[v_1/x]v$ *behaves equivalently to* $[v_2/x]v$*.*

Set $\alpha =$ `untrusted` and $x =$ `neural_net`, and imagine v as the term on lines 4-5. This corollary states that the semantics of v is independent of `neural_net` by way of the behavior of the former being invariant under differing implementations of the latter. The intuition is that `neural_net` is not robust against `[untrusted]` computations, so for the program to be correct, the latter must remain ignorant of the former. That is, v_1 could be the implementation from line 3 and v_2 the constant function returning `true`, and both programs would be considered to behave equivalently. This is done by having program equality, to be made rigorous in Section 5, be *up to indistinguishability*. Indistinguishability trivializes equality upon conflict, which has the effect of isolating the offending usage from the rest of the program which does not misbehave. This means that it is not possible to use data from lines 4-5 in a program at the type in line 6. **Property 4.1 justifies that *tracking* suffices over *erroring* to ensure that data cannot be misused by conflicting computations.**

The previous point is worth reiterating: from a *formal* point of view, it is of no matter whether we return errors. Programs in conflict have derivations under the core type system from Section 3. What we focus on instead is separating misbehaving parts of the program from those well-behaved—so a program whose top-level type indicates no conflict cannot misbehave. We do this because non-conflict is the focal point of our approach to information flow, *defining its conception of dependency tracking* rather than simply being an additional reasoning facility of it. As such, its most natural phrasing is by far as a matter of separation of conflicted and non-conflicted parts, not the allowability of conflict itself. Significant theoretical advantages will be gained from this perspective in Section 5.

4.1.2 Again, for Confidentiality. We have just reviewed an example of integrity reasoning, acting to prevent an untrusted portion of the program from compromising a sensitive one. Let us exercise the same pattern towards confidentiality. Section 1.1.2 gestured at a scenario where secrets are prevented from entering public settings. Figure 10 works a full program in this spirit.

Specifically, we consider a fragment of the machinery for a program implementing a social media program. We start on line 1 with `make_user_profile`, which models functionality to create a public-facing user profile. It takes an **`int`** for the creation timestamp and a **`string`** for the username and returns some **`data`** representing this information. We use it much as we did `neural_net`, with that on line 3 passing it `user_input` from the prior example. This poses no issue here, as one might expect. The usage on line 5 runs into an error, risking exposing password data in a public setting. Property 4.1 states that its semantics is independent of `make_user_profile`. This is due to the conflict in its type giving way to indistinguishability, meaning we can freely swap

out `make_user_profile` for the constant function ignoring its inputs. This again has the practical effect of isolating misuses of `make_user_profile` from the rest of the program.

4.1.3 Confidentiality and Integrity. Note that this example of confidentiality reasoning is structurally identical to the previous covering integrity, with each splitting specifications across the two modalities in analogous ways. **Despite accounting for both, the division of reasoning within our system is not the one between confidentiality and integrity,** but between dependencies and anti-dependencies. Surprisingly, based on the examples shown, it seems that *half* of each of confidentiality and integrity is a matter of dependencies such as `[untrusted]` or `[secret]`, and the other half is a matter of *robustness against* those dependencies, as with `![untrusted]` and `![secret]`. The latter two might be written under a conventional theory of information flow as `[trusted]` and `[public]`, using our syntax for dependencies, with each respectively being allowed to combine with `[untrusted]` and `[secret]` to yield `[untrusted]` and `[secret]`. That is, the latter *infect* the former. However, this is insufficient for our usage of `![untrusted]` and `![secret]`.

The outlined scheme works for reasoning about which dependencies we *have*, but not for which dependencies we are *not allowed to have*—perhaps unsurprisingly, given the implied absence of anti-dependencies. Consider a function at type `[trusted] bool -> ...`, which seemingly wishes to avoid taking untrusted input. Despite its argument type, it can still be passed untrusted information via indirect flows, say, conditional branching surrounding its call site. Merely specifying its argument to be `[trusted]` does not suffice. This issue cannot easily be solved in the domain of dependencies alone. We gain the ability to specify against untrusted inputs in reasoning about *robustness against* dependencies, which by definition accounts for the surrounding context and its indirect flows. Prior work tends to either force users to manually inspect portions of the program to be free of undesired labels [Ørbæk and Palsberg 1997, Fig. 4], which scales poorly in the presence of polymorphism as noted in Section 1.1.1, or introduces ad-hoc specification features—like a function-level robustness label checked against a special *program counter* label denoting the current control flow dependencies [Myers 1999, §2.8; Cecchetti et al. 2017, Fig. 8]. **We reify robustness restrictions as their *own* first-class dependency tracking connective, recognizing its equal stature to the usual one towards obtaining full-spectrum information flow reasoning.** This ecumenical view justifies the preceding deconstruction of confidentiality and integrity.

4.2 Robust Downgrading

Downgrading *removes* dependencies from the quasi-open and quasi-closed modalities, for instance, eliding the α from an `![`α β`] int` to get `![`β`] int`. Robustness and information flow typing have previously made contact in service of downgrading [Cecchetti et al. 2017; Zdancewic and Myers 2001], specifically *robust declassification*. *Declassification* is downgrading applied toward confidentiality, or dependencies regarding secrets. *Robust* declassification recognizes the sensitivity of declassification to manipulation by its environment, protecting against untrusted input. A stronger variant of this feature can be accessed in our system as a special case of the reasoning tools already reviewed, owing to their first-class treatment of robustness. As a motivating example, let us write a variant of the password checker in Figure 1 which uses (1) downgrading and (2) robust declassification to resist brute force guess attempts. We step through each point in turn via the program in Figure 11.

We begin on the left with a `module RobustDeclassify`. It introduces a `dependency rd` and a single method `declassify` which takes an argument with dependencies `[`α `rd]`, peeling off the `rd` dependency while introducing an anti-dependency against `untrusted` information. This can be understood as an existential $\exists$(`rd.[rd`α`] t -> [`α`] ![untrusted] t`). Its implementation proceeds, recalling T-Pack, by determining an implementation of `rd`—here the empty set of dependencies `[]`. We then write the body of `declassify` against `[]`, taking in an `arg` at type `[`α`] t` due to the chosen

```
module RobustDeclassify : sig
  dependency rd

  val declassify : [rd α] t ->
      [α] ![untrusted] t
end = struct
  dependency rd = []

  let declassify = fun arg ->
      { seal arg }
end

open RobustDeclassify
```

```
module PasswordChecker : sig
  dependency secret

  val password : [secret] string
  val check : string ->
       ![untrusted] bool
end = struct
  dependency secret = [rd]

  let password = { "takver" }
  let check = fun attempt ->
      declassify (attempt == password)
end
```

Fig. 11. Robust Password Checking

implementation of `rd`. Notably, `rd` is elided. The rest comes easily, simply enclosing `arg` inside both modalities and returning it. Finally, we **`open RobustDeclassify`** into the current scope.

Continuing to the right, we have the **`PasswordChecker`** interface. It introduces a **`dependency`** `secret` used to represent `password` data. The type of `password` is unchanged from Figure 1. However, the type of `check` differs, first in that its **`bool`** result no longer depends on `secret`, but also in that it is now anti-dependent on `untrusted`. The reason for this becomes clear as we look to the implementation for **`PasswordChecker`**. We implement `secret` as `[rd]`. We are then forced to invoke `declassify` in the implementation of `check` in order to elide the dependency of its body on `rd`—from `password`—rather than having it implicitly removed by virtue of being empty as in **`RobustDeclassify`**. So we get `![untrusted]` in the return type. This restriction prevents `check` from being affected by untrusted inputs. Summarizing, **`RobustDeclassify`** downgrades internally in order to export the `declassify` operation, while **`PasswordChecker`** uses it to robustly declassify.

Consider what Property 4.1 means for this example. When we use `check` in an `[untrusted]` setting we find ourselves in a situation analogous to that on lines 4-5 of Figure 9. As there, indistinguishability provides that any uses of the `![untrusted] bool` in the return value of `check` can be freely replaced with any other value of the same type. That is, under an `[untrusted]` environment, it is as if the declassification never occurred! The declassified data *itself* is made indistinguishable, or freely replaceable with any other; it is effectively un-leaked. Practically, robust declassifiers used with untrusted data cannot affect the rest of the program; they are isolated, as before.

```
let _ : [untrusted] string -> [untrusted] bool =
    fun user_input -> rate_limit { check user_input }
```

We realistically *do* want to be able to pass `[untrusted]` user input to this password checker, so we are not through with the example. It will not do to permanently carry around the `![untrusted]` restriction. Recalling that we specifically wanted to avoid *brute force guessing* resulting from untrusted input, the right move might be to connect downgrading—or removing—the `![untrusted]` restriction to a rate limiting operation. Downgrading anti-dependencies works analogously to dependencies, using a `[]` implementation of `untrusted`. Assume an **`Untrusted`** module exporting `untrusted` and a function `rate_limit : [α] ![untrusted] t -> [α] t`. The idea with `rate_limit` is that it runs the computation passed to it and does not allow itself to be invoked above some frequency. It is not dependency elided because it expects to take a suspended computation as argument, so

taking the argument at the open modality is convenient. Assuming the `PasswordChecker` module is `opened`, we have the above. Only the `[untrusted]` from `user_input` remains. Note with α = `untrusted`, `rate_limit` can downgrade data *in conflict* to data *out of conflict*. **Downgrading precludes us from always rejecting programs with conflicts.** Section 5 will say more.

We impose here the requirement that rate limiting take place while granting clients control over its granularity. This might be used to support some number of rapid attempts before a timed lock-out, or perhaps logging into multiple devices simultaneously. One must only ensure the argument to `rate_limit` does not already use brute force. A final modularity issue remains.

4.2.1 Delegating Conflicts. It is slightly odd for `Untrusted` to offer an operation relevant largely to the `PasswordChecker`. Avoiding brute force may not be the definition of robustness against untrusted input appropriate for other modules. To fix this, we introduce a new `dependency PasswordChecker.untrusted` implemented as `[]`, changing `check` to use it. `rate_limit` can then be made part of the interface of `PasswordChecker`, downgrading `PasswordChecker.untrusted`. So `rate_limit` can *only* be used to remove the restriction against `untrusted` from `check`.

But how is `PasswordChecker.untrusted` related to `Untrusted.untrusted`? We introduce a special operation `avoid : ![PasswordChecker.untrusted] t -> ![Untrusted.untrusted] t` to the former module. Recall that when leaving the scope of an existential, the third and fourth premises of T-Open from Figure 7 preclude mentioning the to-be-unbound dependency variable. This means that a program using a dependency from some module leaving scope must find a way to be rid of that dependency. Downgrading provides one path, and `avoid` another. The intuition behind the latter is to reify the meaning of a dependency going out of scope in terms of one still in scope. A practical implementation would introduce syntax for generating these `avoid` declarations, such as `dependency untrusted conflicts Untrusted.untrusted`. The generated declarations would be automatically called when leaving scope, and the type checker could further use them to proactively raise errors between conflicted dependencies. Alternate solutions to this issue, left as future work, fall under the purview of the *avoidance problem*; Crary [2020, §1] gives an overview.

Under this setup, each module controls the downgrading of its own restrictions while still referencing the global notion of that restriction. Existential scoping as highlighted here also prevents existentials from perpetually hiding conflicts, say, by abstracting the same dependency behind different variables. It will force either their declassification or their `avoid`ance.

4.2.2 Comparing to Standard Robust Downgrading, pt. 1. We mentioned our approach exhibits a stronger variant of the robustness reasoning explored previously by Zdancewic and Myers [2001]. Their treatment of the preceding password checking example reveals why. Under standard robust declassification, `declassify` would be a built-in operation of the language imposing its own ad-hoc robustness restrictions against `[untrusted]`. These restrictions would be satisfied by simply using `endorse` on `attempt` before passing `attempt == password` to `declassify` [Cecchetti et al. 2017, Fig. 1], where *endorsement* acts to downgrade `[untrusted]` dependencies. One might consider this implicit discharging of robustness restrictions for trusted input a convenience, but given the preceding example, it is in reality quite dangerous! Merely endorsing the attempt `string` does little to increase the safety of the ensuing declassification. That declassification is safe from manipulation by untrusted input must be *independently* and *explicitly* validated, as with `rate_limit`. Observe, however, that `rate_limit` does something quite distinct from endorsement: it downgrades the *robustness requirement itself*. This is possible because **our system possesses *internal* and therefore *programmable* robustness reasoning through the closed modality,** not specially attached to declassification behavior or the like. In turn, we need not connect its relaxation to inbuilt operations like `endorse`, instead deferring control to the interface- or specification-writer.

$$
\begin{aligned}
e \sim e' \in \mathsf{F}(A) \mid \phi;\Delta &\triangleq e \mapsto^* \mathtt{ret}(v),\ e' \mapsto^* \mathtt{ret}(v'),\ v \sim v' \in A \mid \phi;\Delta \\
e \sim e' \in A \to \underline{B} \mid \phi;\Delta &\triangleq \Delta \subseteq \Delta',\ \phi'_{\bigcirc} \vdash_{(\supseteq)} \phi_{\bigcirc},\ v \sim v' \in A \mid \phi';\Delta' \implies \\
&\quad \mathsf{ap}(e;v) \sim \mathsf{ap}(e';v') \in \underline{B} \mid \phi';\Delta' \\
e \sim e' \in \forall(\alpha.\underline{B}) \mid \phi;\Delta &\triangleq \Delta \subseteq \Delta',\ \Delta' \vdash \phi' \implies \\
&\quad e[\phi'] \sim e'[\phi'] \in [\phi'/\alpha]\underline{B} \mid \phi;\Delta' \\
e \sim e' \in \bigcirc_{\phi'}(\underline{B}) \mid \phi;\Delta &\triangleq \mathtt{produce}(e) \sim \mathtt{produce}(e') \in \underline{B} \mid \phi\,\phi';\Delta \\
v \sim v' \in \mathsf{U}(\underline{B}) \mid \phi;\Delta &\triangleq \mathtt{force}(v) \sim \mathtt{force}(v') \in \underline{B} \mid \phi;\Delta \\
\langle\rangle \sim \langle\rangle \in \mathtt{unit} \mid \phi;\Delta &\triangleq \mathtt{true} \\
\mathtt{l}\cdot v \sim \mathtt{l}\cdot v' \in A_1 + A_2 \mid \phi;\Delta &\triangleq v \sim v' \in A_1 \mid \phi;\Delta \\
\mathtt{r}\cdot v \sim \mathtt{r}\cdot v' \in A_1 + A_2 \mid \phi;\Delta &\triangleq v \sim v' \in A_2 \mid \phi;\Delta \\
\langle v_1, v_2\rangle \sim \langle v'_1, v'_2\rangle \in A_1 \otimes A_2 \mid \phi;\Delta &\triangleq v_1 \sim v'_1 \in A_1 \mid \phi;\Delta,\ v_2 \sim v'_2 \in A_2 \mid \phi;\Delta \\
\mathsf{pack}[\phi'](v) \sim \mathsf{pack}[\phi'](v') \in \exists(\alpha.A) \mid \phi;\Delta &\triangleq \Delta \vdash \phi',\ v \sim v' \in [\phi'/\alpha]A \mid \phi;\Delta \\
\mathtt{seal}[\phi'](v) \sim \mathtt{seal}[\phi'](v') \in \bullet_{\phi'}(A) \mid \phi;\Delta &\triangleq \phi_{\bigcirc} \vdash \phi'_{\bullet},\ v \sim v \in A \mid \phi;\Delta,\ v' \sim v' \in A \mid \phi;\Delta \text{ or} \\
&\quad v \sim v' \in A \mid \phi;\Delta
\end{aligned}
$$

Fig. 12. Logical Relation for Robust Non-Interference

Beyond robust declassification, we can express robust *endorsement* in much the same way, producing restrictions on its output. One form of this is *transparent endorsement*, inducing `![secret]` restrictions. Another variant inducing `![untrusted]` has not been realized by prior work, despite its usefulness, because it risks breaking the confidentiality-integrity duality. We do not mind doing so, because **our duality is between dependencies and robustness against them.**

A few remarks follow. First, our specifications tend to simplify those of prior work, where each annotation must carry separate confidentiality and integrity components governed by opposing lattice operations [Arden and Myers 2016]. Second, we have used verbose syntax throughout these examples to relate them more closely to Section 3. A practical implementation could extend the `{...}` notation to the closed modality, automatically unsealing and resealing data under it at the bracket boundary. Bidirectional synthesis of the brackets would account for the syntax of the introduction. We leave this to future work. Third, we speak here in terms of dependencies and anti-dependencies since the call-by-value fragment, by blurring polar behavior, resists discussion in terms of information and flows. We now look to the metatheory of our duality which, among other developments, will allow us to substantiate Property 4.1. This will be significantly more straightforward than prior accounts of robustness.

5 Metatheory

Soundness for our system will take the form of an indistinguishability or *non-interference* property. We capture it through a binary logical relation, shown in Figure 12. Following the type system in Section 3, it is bifurcated into two mutually recursive sub-relations for values and computations. This relation will define program equivalence as used by Property 4.1. All results in this section have been mechanized in Lean; proof scripts are provided in the supplemental materials.

Starting with that for computations, the form of the relation is $e \sim e' \in \underline{B} \mid \phi;\Delta$, read as "$e$ is semantically related to e' at computation type $\underline{B}$ and ambient dependencies ϕ, all scoped under Δ."

The first case of the relation, at $\mathsf{F}(A)$, evaluates both sides to $\mathtt{ret}(v)$ and checks their membership in the value relation. The evaluation relation used is entirely standard, so we elide it. This case gives rise to the following two properties stating closure under forward and reverse evaluation.

Lemma 5.1 (Head Reduction). *If* $e_1 \sim e_2 \in \underline{B} \mid \phi; \Delta$ *and* $e_1 \longmapsto^* e_1'$ *then* $e_1' \sim e_2 \in \underline{B} \mid \phi; \Delta$.

Proof. By induction on $\underline{B}$ and use of evaluation in the case for $\mathsf{F}(A)$. □

Lemma 5.2 (Head Expansion). *If* $e_1 \sim e_2 \in \underline{B} \mid \phi; \Delta$ *and* $e_1' \longmapsto^* e_1$ *then* $e_1' \sim e_2 \in \underline{B} \mid \phi; \Delta$.

Proof. By induction on $\underline{B}$ and use of evaluation in the case for $\mathsf{F}(A)$. □

Beyond this case, negative types are defined by their elimination forms, so the relation for computations is defined by eliminating the expressions being related. For instance, the definition of relatedness at the open modality invokes $\mathtt{produce}(-)$ on each side, raising the ambient security level, and terms at function type are related extensionally. Note that when the latter assumes related arguments at A, it is at a raised ambient security level ϕ'. This is required to satisfy the following properties. t and $\mathbb{T}$ respectively denote terms and types generic over values and computations.

Lemma 5.3 (Dependency Monotonicity). *If* $t \sim t' \in \mathbb{T} \mid \phi; \Delta$ *and* $\phi_{\bigcirc}' \vdash_{(\sqsupseteq)} \phi_{\bigcirc}$ *then* $t \sim t' \in \mathbb{T} \mid \phi'; \Delta$.

Proof. By induction on $\mathbb{T}$, using the further assumption $\phi_{\bigcirc}' \vdash \phi_{\bigcirc}$ in the case for $A \to \underline{B}$. □

Lemma 5.4 (Type Monotonicity). *If* $t \sim t' \in \mathbb{T} \mid \phi; \Delta$ *and* $\mathbb{T} \subseteq \mathbb{T}'$ *then* $t \sim t' \in \mathbb{T}' \mid \phi; \Delta$.

Proof. By induction on $\mathbb{T}$ and application of Lemma 5.3 in the case for $\bigcirc_{\phi'}(\underline{B})$. □

This means we can freely upgrade the relation along the subtyping relationship from Section 3. Inspecting the relation, the cases for the computation types $\bigcirc_{\phi'}(\underline{B})$ and $A \to \underline{B}$ are the only ones modifying the ambient dependencies, as promised in Section 3.3. A further monotonicity property arises from the cases for functions and universal quantifiers extending the environment Δ.

Lemma 5.5 (Kripke Monotonicity). *If* $t \sim t' \in \mathbb{T} \mid \phi; \Delta$ *and* $\Delta \subseteq \Delta'$ *then* $t \sim t' \in \mathbb{T} \mid \phi; \Delta'$.

Proof. By induction on $\mathbb{T}$, using the assumption $\Delta \subseteq \Delta'$ in the cases for $A \to \underline{B}$ and $\forall(\alpha.\underline{B})$. □

The above properties establish ϕ and Δ as *Kripke worlds* within the relation, with $\phi_{\bigcirc}' \vdash \phi_{\bigcirc}$ and $\Delta \subseteq \Delta'$ known as *Kripke quantification*. Importantly, the dependencies α contained in Δ are exactly *included middles* from a semantic perspective, as noted in Section 3.3.2. They will be those relevant to indistinguishability as deployed by the relation.

Moving to the value relation $v \sim v' \in A \mid \phi; \Delta$, it is read as "$v$ is semantically related to v' at value type A and ambient dependencies ϕ, all scoped under Δ." Positive types are defined by their introduction forms, so the value relation—but for $\mathsf{U}(\underline{B})$—is defined by pattern matching the values to be related with the introduction forms expected of their types. Values not of the expected form are considered unrelated. Any values inside matching introduction forms are further required to be related. For instance, the case for pairs $A_1 \otimes A_2$ requires both sides match the pattern $\langle \_, \_ \rangle$ and that their components be pointwise related at A_1 and A_2.

The exception to the requirement that inner values be related occurs in the all-important case for the quasi-closed modality $\bullet_{\phi'}(A)$. There are *two* paths for values v, v' to be related under it: (1) to satisfy $\phi_{\bigcirc} \vdash \phi_{\bullet}'$ with v and v' *reflexively related* at the inner type, or (2) for v and v' to simply be related at the inner type as usual. The former uses the entailment $\phi_{\bigcirc} \vdash \phi_{\bullet}'$, mentioned in Section 3.3 as the indistinguishability condition, to relate values under the modality nearly vacuously. Reflexive relatedness merely ensures that each value still individually behaves according to its type. The property resulting from this case relies on certain auxiliary lemmas.

Lemma 5.6 (Symmetry). *If* $t \sim t' \in \mathbb{T} \mid \phi; \Delta$ *then* $t' \sim t \in \mathbb{T} \mid \phi; \Delta$.

Proof. By induction on $\mathbb{T}$. □

Lemma 5.7 (Transitivity). *If* $t \sim t' \in \mathbb{T} \mid \phi; \Delta$ *and* $t' \sim t'' \in \mathbb{T} \mid \phi; \Delta$ *then* $t \sim t'' \in \mathbb{T} \mid \phi; \Delta$.

Proof. By induction on $\mathbb{T}$, using Lemma 5.6 for reflexivity in the $A \to \underline{B}$ case. □

We use these to state the *sealing lemma*, which captures the essence of indistinguishability.

Lemma 5.8 (Seal). *If* $t \sim t \in \mathbb{T} \mid \phi; \Delta$, $t' \sim t' \in \mathbb{T} \mid \phi; \Delta$, $\mathbb{T} \geq \phi'$, *and* $\phi_{\bigcirc} \vdash \phi'_{\bullet}$ *then* $t \sim t' \in \mathbb{T} \mid \phi; \Delta$.

Proof. By induction on $\mathbb{T}$, using Lemma 5.6 and Lemma 5.7 for reflexivity in the $A \to \underline{B}$ case. □

This lemma relates two arbitrary terms t and t' which **(1)** satisfy the relation reflexively **(2)** at a type $\mathbb{T}$ sealed at ϕ' **(3)** under ambient dependencies ϕ where $\phi_{\bigcirc} \vdash \phi'_{\bullet}$. In effect, we can freely equate terms where all the information in them is guaranteed to be sealed at some ϕ' conflicting with the ambient dependencies ϕ. Relatedness being trivialized for terms satisfying certain conditions prevents terms not satisfying those conditions from depending on them while remaining in the relation. The method by which the former are related is unavailable to the latter. In turn, terms satisfying the antecedents of Lemma 5.8 are isolated from the rest and cannot affect their behavior. This isolation property is known as *non-interference* [Goguen and Meseguer 1982]. That the isolated terms are those violating their robustness requirements[4] means that **non-interference is defined by robustness within our system.** Misbehaving terms cannot interfere with those well-behaved.

Being derived from the semantics of the open and closed modalities, Lemma 5.8 relies crucially on the case for closed being aware of the ambient dependencies—specifically the value-level *flows-to-be*—as noted in Section 3.2. Importantly, this case is the *only* one making any concessions to indistinguishability; all other types have a typical treatment of equality. **Our definition of program equality does not hold indistinguishability in special regard. It derives from the definition of equality at the closed modality under the ambient dependencies of the open modality.** The isolation behavior we exploit in this work arises inevitably from here.

Finally, observe that Lemma 5.8 forces the faithful tracking of both $\bigcirc_{\phi}(\underline{B})$ and $\bullet_{\phi'}(A)$ within the semantics. Failing to do so would allow for terms vacuously related by it which do not actively satisfy its antecedents. However, the relation would then require such terms to be truly related according to their type, which is impossible in general. As an aside, note that Lemma 5.8 embodies the equation from Section 3.3. And through another lens, it states that if $\mathbb{T}$ is $\bullet_{\phi'}$-modal then it is $\bigcirc_{\phi}$-connected, corresponding to the forward direction of Sterling and Harper [2021a, Lem. 3.6].

A minor limitation is that the case for existentials requires both sides to have the same implementing dependencies ϕ'. We might relax this restriction by interpreting dependency variables as relations to be substituted into the type A, following the interpretation of type variables under relational parametricity [Reynolds 1983]. Existentials could then be included in the sealing judgement $A \geq \phi$, as forecast in Section 3.1. However, this induces a steep increase in complexity and departs from all other accounts of non-interference. We leave such a generalization to the future.

The relation so far has been defined on terms closed in term variables x. The type system from Section 3 works on open terms. To bridge the divide, we must generalize the former to open terms. Assume contexts Δ, Δ', and Γ, and assume ϕ, ϕ' such that $\Delta \vdash \phi$ and $\Delta' \vdash \phi'$. Define a map δ from dependency variables $\alpha \in \Delta$ to $\Delta' \vdash \phi$, and a map γ from term variables $x : A \in \Gamma$ to values v closed in term variables and scoped under Δ'. Then define $\gamma \sim \gamma'$ such that $\forall x : A \in \Gamma \,.\, \gamma(x) \sim \gamma'(x) \in \delta(A) \mid \phi'; \Delta'$, that is, mapping the same variable to related values. When δ and γ are passed types and terms, assume they recurse through their structure, substituting

[4] Excepting those vacuously satisfying $A \geq \phi'$ by containing no information, like `unit`.

for all variables in their domain. The definition of the open relation is that under the preceding assumptions, the following must hold: $\delta(\gamma(t)) \sim \delta(\gamma'(t')) \in \delta(\mathbb{T}) \mid \delta(\phi) \cup \phi'; \Delta'$. We write this concisely as $\Delta; \Gamma; \phi \gg t \sim t' \in \mathbb{T} \mid \phi'; \Delta'$. We can now state the fundamental theorem.

Theorem 5.9 (Fundamental Theorem)**.** *If* $\Delta; \Gamma; \phi \vdash t : \mathbb{T}$ *then* $\Delta; \Gamma; \phi \gg t \sim t \in \mathbb{T} \mid \phi'; \Delta'$.

Proof. By induction on a derivation of $\Delta; \Gamma; \phi \vdash t : \mathbb{T}$, using Lemma 5.8 in the T-Unseal case. □

This property connects static well-typedness to membership in the semantics. It states that any well-typed program is reflexively related under the open relation. This is nearly the same as being reflexively related under the closed relation, except that each side receives *related* rather than *equivalent* substitutions for its term variables. This is important for the final corollary.

Corollary 5.10 (Robust Non-Interference)**.** *If* $\Delta; x : A; \phi \vdash t : \mathbb{T}$ *and* $A \geq \phi$ *and* $\Delta; \varnothing; \phi \vdash v_1, v_2 : A$ *with* ϕ *nonempty then* $[v_1/x]t \sim [v_2/x]t \in \mathbb{T} \mid \phi; \Delta$.

Proof. Immediate from Theorem 5.9, using Lemma 5.8 to obtain that v_1 and v_2 are related. □

Set $\Delta = \alpha$, $\phi = \alpha$, $A = \bullet_\alpha(\ldots)$, and $\mathbb{T} = \mathsf{U}(\bigcirc_\alpha(\mathsf{F}(\ldots)))$ to obtain Property 4.1, using Figure 12 as the definition of equivalence. We elucidate this result in two parts. First, we briefly recap the reconciliation of declassification and non-interference by Gouni et al. [2025] and review how this result extends it. We then compare to the standard definition of robust downgrading.

5.1 Downgrading for Cheap

Where do we account for downgrading behavior? Notice in the definition of the open relation for Theorem 5.9 that there are *two* dependency variable environments Δ and Δ' at hand. The first is the environment coming from the static typing, and the second is that intended for use in the closed relation. When we close the relation under Δ' as there, we fix the dependencies relevant to the condition $\phi_\bigcirc \vdash \phi'_\bullet$, and therefore to non-interference. The definition of the open relation then assumes related substitutions $\gamma \sim \gamma'$ for the term variables in Γ, which has the effect of assuming non-interference-preserving implementations of all in-scope downgraders.

For instance, we may pick `untrusted` to be in Δ' with δ mapping `untrusted` to `untrusted`; ϵ, making it able to be used to trigger indistinguishability, and elide all other currently in-scope dependency variables by mapping them to ϵ. And with `rate_limit : [α] ![untrusted] t -> [α] t` in scope we assume related implementations for it at its type. When downgrading `![untrusted]`, one might think we risk bringing the input data out of conflict, which would be dangerous for the reasons previously outlined. Since we have *assumed* related implementations of `rate_limit`, however, we can be guaranteed related outputs from related inputs. That is, `rate_limit` and *only* `rate_limit` is permitted to downgrade `![untrusted]` data while preserving relatedness.

The idea of non-interference we have conveyed is thus *already* stated up to downgrading, without extra effort on our part. The shape of downgrading is determined by which dependencies δ maps into Δ', and the operations for which $\gamma \sim \gamma'$ provides related implementations. Our work recognizes that this pattern continues to function when both the quasi-open and quasi-closed modalities are in play, rather than just the former as in Gouni et al. [2025]. We defer to them for further exposition. Note that when we transform a dependency to ϵ, it is downgraded because $\epsilon_\bigcirc \not\vdash \epsilon_\bullet$ by Figure 8; it can no longer be used to trigger indistinguishability, or as a justification to trivialize equality. The non-emptiness assumption in Corollary 5.10 is due to this. Further, when we fix the dependencies in Δ' these can be thought of as the *included middle* truth values interpreted in Section 3.3.2.

5.2 Comparing to Standard Robust Downgrading, pt. 2

Let us review what this all means for the conception of robustness from prior work. Previous accounts of robustness have cast it as a *4-hyperproperty*, or a matter of 4 related traces of a program. Our binary logical relation, however, only concerns 2 traces. The reason for the simplification is due to our *explicit* and *typed* treatment of robustness. We offer justification through the words of prior work in robust declassification.

> If we could precisely identify the release events, this would allow us to specify robust declassification as a 2-safety property on those release events. Cecchetti et al. [2017]

Conventional work integrating robustness into information flow specifically focuses on robust *declassification* and its release of secrets. Declassifiers are primitive to the language, and they are the focus of robustness. As such, stating robust declassification requires first detecting the use of declassification, and subsequently ensuring the robustness of its uses. However, as shown in Section 4.2, we defer control over robustness to authors of module interfaces. Specifically, we do this by reifying robustness as its own connective, the closed modality. So the type of the program directly states its robustness requirements, allowing our semantics to be formulated around it.

Beyond robustness being made explicit in the type system, the *typed* nature of our semantics is the second key piece of the simplification. Prior approaches to the semantics of robust declassification attempt to do so in an *untyped* way, purely reasoning over traces of programs. We take the perspective that **the semantics of robustness is driven by types,** giving our semantic definition access to the type of the computation over which it reasons. Taking explicit specifications and typed semantics together, robustness may be defined by interpreting the closed modality. This results in a system which makes strides in expressiveness while presenting a significantly simpler metatheory. Corollary 5.10 provides that data with robustness restrictions against certain contexts is unavailable to computations in those contexts, which is analogous to the separation properties pursued by the more complex formulations. The above argument also applies to transparent endorsement.

5.3 Mechanization Notes

Proofs of the theorems just stated can be found in `Mechanize/Semantics.lean`, in the attached artifact. We occasionally use set-theoretic subsetting and intersection operations in place of entailments on $\phi_{\bigcirc}, \phi_{\bullet}$ for ease of mechanization, where the two are proven equivalent. We used free large language models to search the Lean and Mathlib documentation. They were also used to complete routine cases and fill holes in auxiliary lemmas while mechanizing the paper proofs of Gouni et al. [2025], which ultimately provided helpful structure for this development. We otherwise avoided the use of such tools in order to extract the maximal metatheoretic insights.

6 Related Work

Related work has structured our exposition throughout. We review here the few remaining points.

Graded Effects. Our framework may be understood as one of *graded monads* or *effects*, grading $\bigcirc_\phi(A)$ and $\bullet_\phi(A)$ with the free semilattice ϕ. Of note, Torczon et al. [2024] sets up graded effects in a Call-By-Push-Value setting. There, the judgemental effect ϕ grades the $\mathsf{U}(\underline{B})$ connective and resides only within computations. This would be the wrong move in our case because the open modality is *idempotent*, meaning it is of no object how many time the open 'effect' runs. So we need not restrict its ϕ to the computation judgement; indeed, this would break the closed modality.

Open and Closed. We mentioned the forward direction of Sterling and Harper [2021a, Lem. 3.6] corresponds to Lemma 5.8. What about the reverse? This does not hold in our case, due to the choices made in Section 3.3.2 to diverge from the standard modalities. Fortunately, it is also not

needed for our current goals. A *singleton* dependency context, in the style of the enriched effect calculus [Egger et al. 2014], seems to be able to recover the full theorem; we consider it future work.

Other work with the open and closed modalities has exploited the idea of *included middles* towards declassification-like behavior, if unacknowledged. The *abstraction function* of Grodin et al. [2026] appears to operate distantly similarly. More work is needed to determine the relationship.

Confidentiality and Integrity. The best-known duality within information flow is, of course, that between confidentiality and integrity, introduced by Biba [1977]. Cecchetti et al. [2017] maintains it in the presence of advanced mechanisms like robust downgrading and transparent endorsement, though at the cost of metatheoretic complexity. Future work may explore a translation of this perspective into our system, solidifying the relationship between the two.

Substructurality. Gouni et al. [2026] works in a system which extends Gouni et al. [2025] to be able to speak about ϕs which are not sets. Specifically, they are able to speak about the order and quantity of dependencies within the quasi-open modality. Whether the quasi-closed modality can be made substructural in the same sense as done for the quasi-open modality there is future work.

7 Conclusion

We have offered a number of different perspectives on the duality at the heart of information flow, identifying with it several surprising and useful phenomena spanning theory and practice. The type-theoretic consonance of our analysis will likely continue to bear fruit beyond what has been discussed. Shall secure information flow focus on tracking *information* or *flows*? We say both.

Acknowledgements

We thank Corinthia Aberlé, Harrison Grodin, and Jonathan Sterling for their guidance in grasping the background material for this work. This research was supported by the Department of Defense under Grant No. H98230-23-C-0275 and by seed funding from the CyLab Security and Privacy Institute at Carnegie Mellon University. The lead author was supported by a Sansom Endowed Presidential Fellowship. The writing and insights in this paper were produced exclusively by the minds of the authors, in the hope that others might benefit from them.

References

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A core calculus of dependency. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/292540.292555

Owen Arden and Andrew C Myers. 2016. A Calculus for Flow-Limited Authorization. In *29th Computer Security Foundations Symposium (CSF)*. doi:10.1109/CSF.2016.17

Kenneth J Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report. https://apps.dtic.mil/sti/citations/ADA039324

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. In *20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. doi:10.1145/2784731.2784733

Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. 2011. Resource-Aware Authorization Policies for Statically Typed Cryptographic Protocols. In *24th Computer Security Foundations Symposium (CSF)*. doi:10.1109/CSF.2011.13

Ethan Cecchetti, Andrew C Myers, and Owen Arden. 2017. Nonmalleable Information Flow Control. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. doi:10.1145/3133956.3134054

Pritam Choudhury. 2022. Monadic and Comonadic Aspects of Dependency Analysis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1320–1348. doi:10.1145/3563335

Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. A Dependent Dependency Calculus. In *European Symposium on Programming (ESOP)*. doi:10.1007/978-3-030-99336-8_15

Karl Crary. 2020. A focused solution to the avoidance problem. *Journal of Functional Programming* 30, Robert Harper Festschrift Collection (2020). doi:10.1017/S0956796820000222

Jeff Egger, Rasmus Ejlers, Alex Simpson, et al. 2014. The enriched effect calculus: syntax and semantics. *Journal of Logic and Computation* 24, 3 (2014), 615–654. doi:10.1093/logcom/exs025

Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and Types*. Cambridge University Press.

Joseph A Goguen and José Meseguer. 1982. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy (S&P)*. doi:10.1109/SP.1982.10014

A.D. Gordon and A. Jeffrey. 2001. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW)*. doi:10.1109/CSFW.2001.930143

Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2025. Structural Information Flow: A Fresh Look at Types for Non-interference. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 3954–3980. doi:10.1145/3764116

Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2026. Security Reasoning via Substructural Dependency Tracking. *Proceedings of the ACM on Programming Languages* 10, POPL (2026), 777–805. doi:10.1145/3776669

Harrison Grodin, Runming Li, and Robert Harper. 2026. Abstraction Functions as Types: Modular Verification of Cost and Behavior in Dependent Type Theory. *Proceedings of the ACM on Programming Languages* 10, POPL (2026), 895–922. doi:10.1145/3776673

Andrew K Hirsch and Ethan Cecchetti. 2021. Giving semantics to program-counter labels via secure effects. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434316

IEEE Standards Board. 1990. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* (1990), 1–84. doi:10.1109/IEEESTD.1990.101064

Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer Dordrecht. doi:10.1007/978-94-007-0954-6

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

Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. Necessity Specifications for Robustness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 811–840. doi:10.1145/3563317

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. doi:10.1017/S0956796807006326

Kenji Miyamoto and Atsushi Igarashi. 2004. A Modal Foundation for Secure Information Flow. In *Workshop on Foundations of Computer Security (FCS)*. https://www.fos.kuis.kyoto-u.ac.jp/~igarashi/papers/pdf/modal-FCS04.pdf

Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *4th Annual Symposium on Logic in Computer Science (LICS)*. doi:10.1109/LICS.1989.39155

Andrew C Myers. 1999. JFlow: practical mostly-static information flow control. In *26th ACM SIGPLAN-SIGACT symposium on Principles of programming language (POPL)*. doi:10.1145/292540.292561

Peter Ørbæk and Jens Palsberg. 1997. Trust in the λ-calculus. *Journal of Functional Programming* 7, 6 (1997), 557–591. doi:10.1017/S0956796897002906

Vineet Rajani, Alex Coleman, and Hrutvik Kanabar. 2025. A Graded Modal Approach to Relaxed Semantic Declassification. In *38th Computer Security Foundations Symposium (CSF)*. doi:10.1109/CSF64896.2025.00032

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83 (IFIP Congress Series)*. https://people.mpi-sws.org/~dreyer/tor/papers/reynolds.pdf

Egbert Rijke, Michael Shulman, and Bas Spitters. 2020. Modalities in homotopy type theory. *Logical Methods in Computer Science* 16, 1 (2020), 1–79. doi:10.23638/LMCS-16(1:2)2020

Patrick Schultz and David I Spivak. 2019. *Temporal Type Theory: A Topos-Theoretic Approach to Systems and Behavior*. Birkhäuser Cham. doi:10.1007/978-3-030-00704-1

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, 3 (2008), 1–31. doi:10.2168/LMCS-4(3:10)2008

Jonathan Sterling and Robert Harper. 2021a. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 68, 6 (2021), 1–47. doi:10.1145/3474834

Jonathan Sterling and Robert Harper. 2021b. A metalanguage for multi-phase modularity. In *ML Workshop (MLW)*. https://www.cs.cmu.edu/~rwh/papers/multiphase/mlw.pdf

Jonathan Sterling and Robert Harper. 2022. Sheaf Semantics of Termination-Insensitive Noninterference. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD)*. doi:10.4230/LIPIcs.FSCD.2022.5

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26. doi:10.1145/3133913

Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024. Effects and Coeffects in Call-by-Push-Value. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1108–1134. doi:10.1145/3689750

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Steve Zdancewic and Andrew C Myers. 2001. Robust Declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*. doi:10.5555/872752.873524

$$\textsc{Mark}\ \frac{\Gamma; \phi' \cup \phi \vdash e : A}{\Gamma; \phi' \vdash \mathsf{box}\ e : T^{\phi} A} \qquad \textsc{LetMark}\ \frac{\Gamma; \phi' \vdash e_1 : T^{\phi}(A_1) \qquad \Gamma, x :^{\phi} A_1; \phi' \vdash e_2 : A_2}{\Gamma; \phi' \vdash \mathsf{unbox}\ e_1\ \mathsf{as}\ x\ \mathtt{in}\ e_2 : A_2} \qquad \textsc{Var}\ \frac{x :^{\phi} A \in \Gamma \qquad \phi \subseteq \phi'}{\Gamma; \phi' \vdash x : A}$$

Fig. 13. Hybrid Modality

A The Hybrid Modality

Figure 13 shows a third kind of information flow modality appearing in the literature [Choudhury et al. 2022; Miyamoto and Igarashi 2004]. The introduction rule Mark is structurally identical to that of the open modality, reifying dependencies in the typing judgement within the modality $T^{\phi} A$. However, LetMark is structurally closer to the elimination rule for the closed modality; it inspects its first operand and passes it to the second. Unlike Unseal, it does not operate by way of a sealing judgement $A \geq \phi$, but by marking the bound variable x with the dependencies ϕ from the eliminated modality. Using x will require the judgemental dependencies ϕ' to be greater than the ϕ marking the variable, in similar fashion to the eliminator Produce for the open modality.

We are not aware of a particular justification for this formulation, and indeed its metatheoretic standing is unclear. One might wonder, for instance, what judgemental structure it internalizes. If the ambient dependencies ϕ, one would expect it to be formulated structurally as a function, as in the case of the open modality, but it is not: LetMark is a positive, rather than negative, elimination rule. Furthermore, there do not seem to be any expressiveness benefits of this setup over the others. We suspect this 'hybrid' modality to simply be an accident of formalization, drafted when the precise character of dependency tracking modalities was not as critical as it is here, and set it aside.