Giving Semantics to Program-Counter Labels via Secure Effects

ANDREW K. HIRSCH, Max Planck Institute for Software Systems, Germany
ETHAN CECCHETTI, Cornell University, USA

Type systems designed for information-flow control commonly use a *program-counter label* to track the sensitivity of the context and rule out data leakage arising from effectful computation in a sensitive context. Currently, type-system designers reason about this label informally except in security proofs, where they use ad-hoc techniques. We develop a framework based on monadic semantics for effects to give semantics to program-counter labels. This framework leads to three results about program-counter labels. First, we develop a new proof technique for noninterference, the core security theorem for information-flow control in effectful languages. Second, we unify notions of security for different types of effects, including state, exceptions, and nontermination. Finally, we formalize the folklore that program-counter labels are a lower bound on effects. We show that, while not universally true, this folklore has a good semantic foundation.

Additional Key Words and Phrases: semantics of effects, information-flow control, noninterference

1 INTRODUCTION

Static information-flow control (IFC) assigns *information-flow labels* to data within a program. These labels describe the sensitivity of the data. For instance, data labeled secret is more sensitive than data labeled public. The type system then prevents more-sensitive inputs from influencing less-sensitive outputs. A flow of information can be *explicit*, if a program directly returns an input, or *implicit* if a program conditions on the input and returns a different value from each branch. In both cases, the type system can enforce *noninterference*—the powerful safety property that a program's sensitive inputs will not influence its less-sensitive outputs [Goguen and Meseguer 1982]—by checking that the output is at least as sensitive as the inputs used to compute it.

When we combine effects with implicit flows, however, this simple output checking becomes insufficient. Volpano et al. [1996] demonstrate this concern with the following simple program where the secret value x is either 0 or 1 and write modifies the state and returns the singleton value () of type unit:

$$\text{if } x = 1 \text{ then } \mathsf{write}(1) \text{ else } \mathsf{write}(0)$$

While this program does not directly write a secret and always returns the same thing, an attacker who can read the final state now learns the value of x.

Languages often rule out these effectful implicit flows by tracking the sensitivity of the current control-flow with a *program-counter label* (written pc) in the typing judgment [e.g., Milano and Myers 2018; Myers 1999; Pottier and Simonet 2002]. If the pc is private, then private data influenced which command is executing, so writing to public outputs may leak that data. If the pc is public, however, only public data has determined which program path was taken, so a decision to write

Authors' addresses: Andrew K. Hirsch, Max Planck Institute for Software Systems, Kaiserslautern and Saarbrücken, Germany, akhirsch@mpi-sws.org; Ethan Cecchetti, Cornell University, Ithaca, New York, USA, ethan@cs.cornell.edu.

2475-1421/2021/1-ART1
https://doi.org/

leaks nothing. For instance, a type system with a program-counter label can detect the leak above since we write to state after branching on secret data.

Type-system designers commonly use such intuitive reasoning when building their type system. Then they adjust the type system as needed to prove noninterference. Ideally, designers would instead use semantically- and mathematically-grounded design principles to design type systems. This approach would make the proof of noninterference almost trivial, since the mathematical grounding of the design principles would guarantee noninterference. Developing such design principles requires a semantic model of program-counter labels.

A piece of folklore gives a clue for how to develop these semantic models: the pc label is a lower bound on the effects that can occur in a well-typed program. Taken literally, this folklore does not even seem to type-check since effects are not labels. However, it suggests that we need a framework that relates effects and labels in a meaningful way.

To investigate this intuition, we employ a common semantic model for effects. We translate the earlier example to a monadic form, which returns a pair consisting of the original output and the state set by write.

$$\text{if } x = 1 \text{ then } ((), 0) \text{ else } ((), 1)$$

Indeed, after this translation, checking only the output is sufficient to detect any leaks. The pc label is no longer necessary, lending credence to the above-mentioned piece of folklore.

We formalize these intuitions by building a semantic model of program-counter labels based on monadic treatments of effects. We base our framework on a categorical construct called a *productor* [Tate 2013]. Productors provide the most-general-known framework for the semantics of *producer effects*—a generalization of monadic effects. (In fact, Tate [2013] argues that productors are the most general possible framework for producer effects.)

Since productors, like monads, are a categorical construct, naively applying them to a programming language would require that the programming language only have one variable in its context. We circumvent this weakness by following a suggestion from Tate's [2013] conclusion and developing *strong* productors, allowing us to apply our framework to realistic languages.

Our framework requires that the productors capture the sensitivity of the effects they encode. For instance, when translating the above example into monadic form, the left side of the output pair must capture the visibility of the old output, while the right side must capture the visibility of the heap. If the translated program were well-typed in a noninterfering language—which it was not in the insecure example above—we would therefore be sure that the original program did not leak data. We refer to effects captured by these security-typed productors as *secure effects*.

Our core theorem enables proofs of noninterference for effectful languages (with pc labels) that fit our framework while only proving it directly for the pure part of the languages (without pc labels). As far as we are aware, this is the first theorem proving noninterference for a large swath of languages. Moreover, this style of proof is nearly unknown in the literature. (Algehed and Russo [2017] mention that it is possible, but do not explore it in any depth.)

For languages that fit our framework, proving noninterference (of the effects) is almost trivial, as expected. However, to fit our framework, a language's effects must be secure, and showing that an effect is secure—that it has a productor that properly captures its visibility—requires reasoning about who can see the results of the effect. Luckily, for important examples, this reasoning is not difficult, so our proof technique leads to simpler proofs than previous techniques. As a result, we call this proof technique *Noninterference Half-Off*.

In addition to this new proof technique, we use our framework to unify different notions of noninterference from the information-flow literature. Some notions consider the termination behavior of programs (termination-sensitive), while others do not (termination-insensitive). Our

framework shows that these two notions of noninterference are distinguished by whether or not nontermination is considered a secure effect. This view is both intellectually satisfying and provides half-off proofs of termination-sensitive noninterference.

Finally, we formalize the folklore that pc labels serve as a lower bound on effects. We show that the aphorism is not always true for noninterfering languages that fit our framework, but it is true for fundamental reasons in every realistic information-flow language of which we are aware.

In Section 2, we review the Dependency Core Calculus (DCC) [Abadi et al. 1999], a simple, pure, noninterfering language. DCC serves as an introduction to necessary parts of IFC languages and as the basis of our example languages throughout the paper. We then add the following contributions:

- We demonstrate a productor-based translation for a language with state and exceptions. Beyond exploring the semantics of secure effects, this allows a simple proof of noninterference (Section 3).
- By treating possible nontermination as an effect—as is common in the effects literature—we obtain a simple proof of termination-sensitive noninterference (Section 4).
- We present our general semantic framework for effectful languages with IFC labels (Section 5), allowing us to prove properties about a wide class of IFC languages.
- We define and prove the *Noninterference Half-Off Theorem* (Theorem 5), allowing us to extend noninterference from pure languages to effectful languages in many settings (Section 6).
- We show that the folklore "the program-counter label is a lower bound on the effects in a program" need not hold in our framework. We also show that extending our framework with a few simple rules makes it hold (Section 7).
- We extend the theory of productors to include multiple-input languages like simply-typed λ-calculus and DCC (Appendix A).

2 AN INFORMATION-FLOW-CONTROL TYPE SYSTEM FOR A PURE LANGUAGE

We begin by reviewing Abadi et al.'s [1999] *Dependency Core Calculus* (DCC), a pure language with a simple noninterference property. DCC will form the basis of our examples in Sections 3 and 4. It also serves as a good language to introduce information-flow control (IFC) and noninterference, as well as the notation for this paper.

Figure 1 contains the syntax of DCC. The heart of DCC is the simply-typed λ-calculus with products and sums. The only additional terms are the security features that make DCC interesting from our perspective: $\mathsf{label}_\ell\,(e)$ and $\mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2$. We will also make free use of let notation, with its standard definition. (For simplicity, we omit the fixpoint operator present in the original language [Abadi et al. 1999], though we will add it back in Section 4.)

Labels	ℓ	$\in$	$\mathcal{L}$
Types	τ	$::=$	$\mathsf{unit} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid L_\ell\,(\tau)$
Values	v	$::=$	$() \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid (v_1, v_2) \mid \lambda x : \tau.\, e \mid \mathsf{label}_\ell\,(v)$
Expressions	e	$::=$	$x \mid () \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{proj}_1(e) \mid \mathsf{proj}_2(e)$
		$\mid$	$\mathsf{inl}(e) \mid \mathsf{inr}(e) \mid (\mathsf{match}\ e\ \mathsf{with} \mid \mathsf{inl}(x) \Rightarrow e_1 \mid \mathsf{inr}(y) \Rightarrow e_2\ \mathsf{end})$
		$\mid$	$\mathsf{label}_\ell\,(e) \mid \mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2$
Evaluation Contexts	E	$::=$	$[\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \mathsf{proj}_1(E) \mid \mathsf{proj}_2(E)$
		$\mid$	$\mathsf{inl}(E) \mid \mathsf{inr}(E) \mid (\mathsf{match}\ E\ \mathsf{with} \mid \mathsf{inl}(x) \Rightarrow e_1 \mid \mathsf{inr}(y) \Rightarrow e_2\ \mathsf{end})$
		$\mid$	$\mathsf{label}_\ell\,(E) \mid \mathsf{unlabel}\ E\ \mathsf{as}\ x\ \mathsf{in}\ e$

Fig. 1. Grammar for DCC Types and Terms

$$\frac{\ell \sqsubseteq \ell'}{\ell \triangleleft L_{\ell'}(\tau)} \qquad \frac{\ell \triangleleft \tau}{\ell \triangleleft L_{\ell'}(\tau)} \qquad \frac{\ell \triangleleft \tau_1 \qquad \ell \triangleleft \tau_2}{\ell \triangleleft \tau_1 \times \tau_2} \qquad \frac{\ell \triangleleft \tau_2}{\ell \triangleleft \tau_1 \to \tau_2}$$

Fig. 2. Protection Rules for DCC

The security terms use a set of *information-flow labels*, $\mathcal{L}$, over which DCC is parameterized, that represent restrictions on data use. For instance, if we have labels secret and public, then data labeled secret should not be used to compute data labeled public. We require that labels form a preorder. That is, there is a reflexive and transitive relation $\sqsubseteq$ (pronounced "flows to"). For presentation clarity, we also assume that $\mathcal{L}$ forms a join semilattice, meaning any two labels ℓ_1 and ℓ_2 have a join—a least upper bound—denoted $\ell_1 \sqcup \ell_2$, and there is a *top* element, denoted $\top$, such that $\ell \sqsubseteq \top$ for all $\ell \in \mathcal{L}$. We note again that this is only for clarity of presentation; we could replace every join with any upper bound, and disallow rules that use a join when no upper bound exists. Intuitively, if $\ell_1 \sqsubseteq \ell_2$, then ℓ_2 is at least as restrictive as ℓ_1, so $\top$ is the most-restrictive label. We note that most IFC work assumes that labels form a lattice, meaning labels also have greatest lower bounds and there is a least element $\bot$. We omit this additional structure as we do not find it helpful.

The term $\mathsf{label}_\ell\,(e)$ represents protecting the output of e at label ℓ. That is, e should only be used to compute information at levels at least as high as ℓ. Such computations are possible using the term $\mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2$, which requires the output type of e_2 to be at a high-enough level, and if it is, allows use of e_1 as if it were not labeled through the variable x.

The concept of a type τ being "of high enough level" to use information at label ℓ is expressed in a relation $\ell \triangleleft \tau$, which is read as "ℓ is protected by τ" or "τ protects ℓ." The formal rules defining this relation are in Figure 2. Intuitively, if $\ell \triangleleft \tau$, then τ information is at least as secret as ℓ.

The typing rules for $\mathsf{label}_\ell\,(e)$ and $\mathsf{unlabel}\ e_1\ \mathsf{as}\ \ell\ \mathsf{in}\ e_2$ are as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{label}_\ell\,(e) : L_\ell\,(\tau)} \qquad \frac{\Gamma \vdash e_1 : L_\ell\,(\tau_1) \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \ell \triangleleft \tau_2}{\Gamma \vdash \mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \tau_2}$$

Notice the use of the protection relation in the latter rule. Since unlabel allows a program to compute with labeled data, this check requires the output of that computation to be at least as sensitive as the input. This protection premise is the main security check in DCC's type system.

The operational semantics of DCC are mostly the standard semantics of call-by-value simply-typed λ-calculus, so we only discuss the semantics of the terms $\mathsf{label}_\ell\,(e)$ and $\mathsf{unlabel}\ e_1\ \mathsf{as}\ \ell\ \mathsf{in}\ e_2$. We first note that $\mathsf{label}_\ell\,(E)$ and $\mathsf{unlabel}\ E\ \mathsf{as}\ \ell\ \mathsf{in}\ e_2$ are evaluation contexts, where E stands for an arbitrary evaluation context. That is, computation can take place under both $\mathsf{label}_\ell\,(-)$ and $\mathsf{unlabel}\ -\ \mathsf{as}\ x\ \mathsf{in}\ e_2$. Note that computation *cannot* take place in the second expression of an unlabel operation, since this is expected to run after binding the variable x. The only remaining operational semantics rule is as follows:

$$\mathsf{unlabel}\ (\mathsf{label}_\ell\,(v))\ \mathsf{as}\ x\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]$$

DCC's main security theorem is its *noninterference* theorem. It formalizes the fact that programs do not compute, e.g., public information with secret data. The theorem requires a notion of equivalence at a label ℓ, representing what an attacker who can see values only up to label ℓ can distinguish. The definition is contextual to allow for comparison of first-class functions.

Definition 1 (ℓ-Equivalent Programs)**.** We say programs e_1 and e_2 are *ℓ-equivalent*, denoted $e_1 \approx_\ell e_2$, if for all expression contexts C such that $\vdash C[e_i] : L_\ell\,(\mathsf{unit} + \mathsf{unit})$ and $C[e_i] \longrightarrow^* v_i$ for both $i = 1, 2$, then $v_1 = v_2$.

Intuitively, expressions are ℓ-equivalent if no well-typed decision procedure with output labeled ℓ can distinguish them. Note that this definition only requires equivalent outputs when both programs terminate. Because we omitted DCC's fixpoint operator, the language is strongly normalizing so both terms always converge. We will address potential nontermination in Section 4.

We use this definition to say that two well-typed expressions must be ℓ-equivalent unless their labels allow them to influence ℓ. Formally, the protection relation defines the label of data, leading to the following theorem. Bowman and Ahmed [2015] proved the version we use here and Algehed and Bernardy [2019] provided a machine-checked proof in Agda.

Theorem 1 (Noninterference for DCC [Algehed and Bernardy 2019; Bowman and Ahmed 2015])**.** *For expressions e_1 and e_2 and $\ell \in \mathcal{L}$, if $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, and $\ell \triangleleft \tau$, then for all labels $\ell_{\mathsf{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ or $e_1 \approx_{\ell_{\mathsf{Atk}}} e_2$.*

3 EXAMPLE: NONINTERFERENCE IN A LANGUAGE WITH STATE AND EXCEPTIONS

We now extend our simple noninterfering pure language with two effects: state and exceptions. Both are simplified for space and ease of understanding. Specifically, we consider only one (typed) state cell and one type of exception. Moreover, the type of this state cell, σ, must not contain a function as a subterm in order to avoid concerns around higher-order state.[1] However, neither is difficult to broaden to more-realistic versions. We extend the language with the following syntax:

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & \cdots \mid \mathsf{read} \mid \mathsf{write}(e) \mid \mathsf{throw} \mid \mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\} \\
\text{Evaluation Contexts} & E & ::= & \cdots \mid \mathsf{write}(E) \mid \mathsf{try}\ \{E\}\ \mathsf{catch}\ \{e_2\} \\
\text{Throw Contexts} & T & ::= & [\cdot] \mid T\ e \mid v\ T \mid (T, e) \mid (v, T) \mid \mathsf{proj}_1(T) \mid \mathsf{proj}_2(T) \\
& & \mid & \mathsf{inl}(T) \mid \mathsf{inr}(T) \mid (\mathsf{match}\ T\ \mathsf{with} \mid \mathsf{inl}(x) \Rightarrow e_1 \mid \mathsf{inr}(y) \Rightarrow e_2\ \mathsf{end}) \\
& & \mid & \mathsf{label}_\ell(T) \mid \mathsf{unlabel}\ T\ \mathsf{as}\ x\ \mathsf{in}\ e \mid \mathsf{write}(T)
\end{array}
$$

Here read returns the value of type σ currently stored in the one state cell, while $\mathsf{write}(e)$ replaces that value with e and returns unit. We include $\mathsf{write}(E)$ as an evaluation context, ensuring that e is reduced to a value before being stored. The term throw throws an exception, which propagates through contexts until it either hits top-level or a try block. We use a throw context T to implement this propagation. T is identical to evaluation contexts except it does not include try blocks. Finally, $\mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\}$ runs e_1 until it returns either a value or an exception. If it returns a value v, then the try-catch returns v as well. If it throws an exception, however, the try-catch instead discards the exception and runs e_2. We include $\mathsf{try}\ \{E\}\ \mathsf{catch}\ \{e_2\}$ as an evaluation context, so it will evaluate e_1 to a value or exception, but not as a throw context so it can catch and discard an exception.

These considerations give rise to the following typing rules:

$$
\frac{}{\Gamma \vdash \mathsf{read} : \sigma} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathsf{write}(e) : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash \mathsf{throw} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\} : \tau}
$$

The resulting operational semantic rules are defined on pairs of an expression and a state cell s of type σ. The rules for our effectful operations are as follows:

$$
\langle \mathsf{read}, s\rangle \longrightarrow \langle s, s\rangle \qquad \langle \mathsf{write}(v), s\rangle \longrightarrow \langle (), v\rangle
$$

$$
\langle T[\mathsf{throw}], s\rangle \longrightarrow \langle \mathsf{throw}, s\rangle
$$

$$
\langle \mathsf{try}\ \{v\}\ \mathsf{catch}\ \{e\}, s\rangle \longrightarrow \langle v, s\rangle \qquad \langle \mathsf{try}\ \{\mathsf{throw}\}\ \mathsf{catch}\ \{e\}, s\rangle \longrightarrow \langle e, s\rangle
$$

We modify the rest of the operational semantics by including s without modification in every other rule, as is standard for simply-typed λ-calculus with state [Pierce 2002, Chapter 13.3].

[1] Allowing higher-order state is possible, but it requires recursive types and complicates reasoning about termination.

Technically, our previous noninterference theorem (Theorem 1) still holds, and by the same proof. However, the statement of this theorem is now very weak: it assumes that an attacker cannot see the state or distinguish throw from any other statement. This allows implicit flows. To see how exceptions allow implicit flows, consider the following example program that leaks its input to anyone who can distinguish throw from non-exceptional output:

$$h : L_\ell\,(\mathsf{unit} + \mathsf{unit}) \vdash \begin{array}{l}\mathsf{unlabel}\ h\ \mathsf{as}\ x\\ \mathsf{in\ match}\ x\ \mathsf{with}\\ \quad \mid \mathsf{inl}(\_) \Rightarrow \mathsf{throw}\\ \quad \mid \mathsf{inr}(\_) \Rightarrow \mathsf{label}_\ell\,()\\ \mathsf{end}\end{array} : L_\ell\,(\mathsf{unit})$$

3.1 Ruling Out Implicit Flows

We now aim to eliminate implicit flows and recover a strong notion of noninterference with realistic assumptions about an attacker's power. We achieve this result by changing our typing rules. We associate a *program-counter label* $\mathrm{pc} \in \mathcal{L}$ with the typing judgment to track the sensitivity of the context. Thus, the typing judgment of this new type system takes the form $\Gamma \diamond \mathrm{pc} \vdash e : \tau$ where pc is an information-flow label.

In the examples of implicit flows so far, vulnerabilities arose when we performed certain actions depending on the value of a secret expression. To prevent such problems, we might update pc so that it is at least as high as any value we conditioned on in a match statement. We could then ensure that actions that might leak information about those values cannot type-check in a sensitive environment. However, there is a problem with doing this in DCC: we never match on labeled data. Instead, we must first use unlabel, removing the label from the data before we can use that data in any way, including in a match expression. This is because DCC is a *coarse-grained* information-flow language. (In fact, it is the paradigmatic coarse-grained information-flow language.)

The fact that labeled data can only be used in an unlabel expression means that the unlabel rule is the only rule in which we can reasonably increase the pc. (This may seem like a significant restriction, since we are increasing the program-counter label even when we do not match on the data we are unlabeling. However, recent research has shown that coarse-grained information flow is equivalent to fine-grained information flow, which would increase the program-counter label in the match statement [Rajani and Garg 2018].) We increase the program counter label using the join operator on labels we discussed in Section 2. Thus, the rule for unlabel becomes the following:

$$\frac{\Gamma \diamond \mathrm{pc} \vdash e_1 : L_\ell\,(\tau_1) \qquad \Gamma, x : \tau_1 \diamond \mathrm{pc} \sqcup \ell \vdash e_2 : \tau_2 \qquad \ell \triangleleft \tau_2}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \tau_2}$$

We must also change the typing rules for functions. Intuitively, an expression $\lambda x : \tau.\, e$ will execute the actions of e when it is applied, not when it is defined. It is therefore safe to *construct* a λ-expression in any context, but it is only safe to *apply* one in a context where its effects do not leak information. Since we cannot, in general, know where a function will be used when it is constructed, we instead change the type of the function to restrict where it can be applied. The type $\tau_1 \xrightarrow{\mathrm{pc}} \tau_2$ is a function that takes an argument of type τ_1, returns a value of type τ_2, and can be safely run in contexts which have not discriminated on anything higher than pc. This gives rise to the following two typing rules:

$$\frac{\Gamma, x : \tau_1 \diamond \mathrm{pc}_1 \vdash e : \tau_2}{\Gamma \diamond \mathrm{pc}_2 \vdash \lambda x : \tau_1.\, e : \tau_1 \xrightarrow{\mathrm{pc}_1} \tau_2} \qquad \frac{\Gamma \diamond \mathrm{pc}_1 \vdash e_1 : \tau_1 \xrightarrow{\mathrm{pc}_2} \tau_2 \qquad \Gamma \diamond \mathrm{pc}_1 \vdash e_2 : \tau_1 \qquad \mathrm{pc}_1 \sqsubseteq \mathrm{pc}_2}{\Gamma \diamond \mathrm{pc}_1 \vdash e_1\ e_2 : \tau_2}$$

This change also necessitates adjusting the function protection rule from Figure 2. Applying a function with type $\tau_1 \xrightarrow{\text{pc}} \tau_2$ can still reveal data through its output at the level of τ_2, but it can also reveal information about control flow up to label pc. We therefore need to use the pc to ensure that effects will not leak information, leading to the following modified protection rule.

$$\frac{\ell \triangleleft \tau_2 \qquad \ell \sqsubseteq \text{pc}}{\ell \triangleleft \tau_1 \xrightarrow{\text{pc}} \tau_2}$$

To determine how the pc label should relate to effects, we need to be clear about what effects the attacker can and cannot see. We assume that anyone who can see things labeled ℓ_{State} can see the value stored in the state cell. An attacker who can read ℓ_{State} can see writes to state—since writes can change the stored value—but not reads—since reads leave the value unchanged. This is a reasonable assumption in many cases, but not all. For instance, it assumes the attacker cannot extract information through cache-based timing attacks [e.g., Kocher 1996]. We also assume that any attacker who can see information labeled ℓ_{Exn} can distinguish between exceptions and other values. An attacker who *cannot* see information labeled ℓ_{Exn} is therefore not privy to the success or error status of the program, meaning they also cannot see the result if it returns successfully. They may, however, still be able to observe the state cell if they can read ℓ_{State}.

We can use this model to determine the pc-based typing rules for our effectful operations. Three of the rules are fairly simple. The read rule allows any pc, while the write and throw rules must check that the context is not too sensitive to run this computation.

$$\frac{}{\Gamma \diamond \text{pc} \vdash \text{read} : \sigma} \qquad \frac{\Gamma \diamond \text{pc} \vdash e : \sigma \qquad \text{pc} \sqsubseteq \ell_{\text{State}}}{\Gamma \diamond \text{pc} \vdash \text{write}(e) : \text{unit}} \qquad \frac{\text{pc} \sqsubseteq \ell_{\text{Exn}}}{\Gamma \diamond \text{pc} \vdash \text{throw} : \tau}$$

The try-catch rule is slightly more complicated, since the catch block only executes if the try block throws an exception, and therefore may return different values depending on whether or not an exception occurs. To ensure this control flow does not leak data, the output must be at least as sensitive as the control flow: ℓ_{Exn}.

$$\frac{\Gamma \diamond \text{pc} \vdash e_1 : \tau \qquad \Gamma \diamond \text{pc} \vdash e_2 : \tau \qquad \ell_{\text{Exn}} \triangleleft \tau}{\Gamma \diamond \text{pc} \vdash \text{try } \{e_1\} \text{ catch } \{e_2\} : \tau}$$

None of the other rules change pc, since none of the other rules can leak information about the context or change the context based on a labeled value. These rules may appear to ensure security against the attacker we sketched above, but unfortunately this intuition misses the fact that exceptions can impact control flow outside just try-catch blocks. Consider the following program:

$$h : L_{\ell_{\text{Exn}}}(\text{unit} + \text{unit}), s : \sigma \vdash \begin{array}{l} \text{let } \_ = \text{unlabel } h \text{ as } x \\ \quad \text{in match } x \text{ with} \\ \qquad \mid \text{inl}(\_) \Rightarrow \text{throw} \\ \qquad \mid \text{inr}(\_) \Rightarrow \text{label}_{\ell_{\text{Exn}}}() \\ \quad \text{end} \\ \text{in write}(s) \end{array} : L_{\ell_{\text{Exn}}}(\text{unit}) \tag{1}$$

If $h = \text{label}_{\ell_{\text{Exn}}}(\text{inl}\ ())$, this will throw an exception and the write will never execute. Notably, this program leaks the value of h to anyone who can see ℓ_{State}. Pottier and Simonet [2002] eliminate this leak by constraining what effects can execute after an expression that may throw an exception. While our framework can handle this generality (see Section 5.1), we take a more restrictive but far simpler approach and require that $\ell_{\text{Exn}} \sqsubseteq \ell_{\text{State}}$.

Proving that we have successfully eliminated data leaks requires using a notion of ℓ-equivalence that accounts for exceptions and state. Notably, to properly capture which attackers may view which values, our notion differs depending on how ℓ relates to ℓ_{State} and ℓ_{Exn}.

Definition 2. We say programs e_1 and e_2 are *state and exception ℓ-equivalent*, denoted $e_1 \cong_\ell^{\mathrm{SE}} e_2$, if for all values s and expression contexts C such that $\vdash s : \sigma$, $\diamond\,\mathsf{pc} \vdash C[e_i] : L_\ell\,(\mathsf{unit} + \mathsf{unit})$, and $\langle C[e_i], s\rangle \longrightarrow^* \langle v_i, s_i\rangle$ for both $i = 1, 2$, then $v_1 = v_2$ if $\ell_{\mathsf{Exn}} \sqsubseteq \ell$ and $s_1 = s_2$ if $\ell_{\mathsf{State}} \sqsubseteq \ell$.

Intuitively, Definition 2 says that e_1 and e_2 are equivalent if no program will let an attacker at label ℓ distinguish them through either the program output or the state. Because the attacker can only see the program output if $\ell_{\mathsf{Exn}} \sqsubseteq \ell$, we only check output equivalence in that case. Similarly, because the attacker can only see the state cell if $\ell_{\mathsf{State}} \sqsubseteq \ell$, we only check equivalence of the state cells when the flow holds. Note that structural equality on state values is sufficient because we assumed σ contains no function types as subterms.

While it is possible to directly prove our type system enforces noninterference using such an equivalence relation [Russo et al. 2008; Tsai et al. 2007; Waye et al. 2015], we take a different approach. We formalize the view that the pc allows only secure effects to simplify the noninterference proof and help avoid the need for clever ad-hoc reasoning, such as the argument we made for try-catch.

3.2 Tracking Effects

To show that the pc label restricts well-typed programs to be those with secure effects, we need to track the effects in a program. To do so, we use a standard type-and-effect system [Lucassen and Gifford 1988; Marino and Milstein 2009; Nielson 1996; Nielson and Nielson 1999]. This type-and-effect system assigns each step of a typing proof an effect ε from a set $\mathcal{E}$ of possible effects. Therefore, we need to decide on the contents of $\mathcal{E}$.

So far, we have described our language has having two effects: state and exceptions. We might therefore let $\mathcal{E} = \{\mathrm{S}, \mathrm{E}\}$ where S represents state and E exceptions.

This choice is undesirable for two reasons. First, consider the following program (where the state is of type unit + unit):

$$x : \mathsf{unit} + \mathsf{unit} \vdash \begin{array}{l}\mathsf{match}\ x\ \mathsf{with}\\ \quad \mid \mathsf{inl}(\_) \Rightarrow \mathsf{read}\\ \quad \mid \mathsf{inr}(\_) \Rightarrow \mathsf{throw}\\ \mathsf{end}\end{array} : \mathsf{unit} + \mathsf{unit}$$

This program could either read state or throw an exception. So which effect should we give it? We must note that both are possible, which we can do by having an effect that represents "can use state and/or throw an exception." In fact, we need an effect for each possible *collection* of our effects. Thus our possible effects come from the power set, so $\mathcal{E} = 2^{\{\mathrm{S},\mathrm{E}\}}$. Notably, this gives our effects a nice lattice structure, as is standard for a power set.

Second, considering state as a single effect does not allow us to represent that our attacker can only see writes. For instance, consider the following program in the same setting as above:

$$h : L_\ell\,(\mathsf{unit} + \mathsf{unit}) \vdash \begin{array}{l}\mathsf{unlabel}\ h\ \mathsf{as}\ x\\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\\ \quad \mid \mathsf{inl}(\_) \Rightarrow \mathsf{label}_\ell\,(\mathsf{read})\\ \quad \mid \mathsf{inr}(\_) \Rightarrow \mathsf{label}_\ell\,(\mathsf{inl}\ ())\\ \mathsf{end}\end{array} : L_\ell\,(\mathsf{unit} + \mathsf{unit})$$

An attacker who cannot distinguish values labeled ℓ cannot distinguish which branch the program takes. However, if we were to keep state as a single effect, we would have to label that branch

as having the state effect, and therefore disallow it. To resolve this problem, we separate state operations into read effects R and write effects W and change $\mathcal{E}$ to $2^{\{R,W,E\}}$. Note that since our attacker cannot see reads at all, we could consider read to be a pure operation. Though this would simplify a few technical details, we find it is more intuitive to include R as an effect.

Now we can develop our type-and-effect system. We again change the form of the typing judgment, this time to $\Gamma \vdash e : \tau \diamond \varepsilon$, where $\varepsilon \subseteq \{R, W, E\}$. For readability, we will often write this set without curly brackets. The example program above that could read state or throw an exception would therefore have the typing judgment $\Gamma \vdash e : L_\ell\,(\mathsf{unit} + \mathsf{unit}) \diamond R, E$.

Since we are trying to analyze the security of the program, we create a function $\ell\_$ from effects to labels, associating a label ℓ_ε with each effect ε. This label corresponds to our attacker model of who can observe the effect. For W and E we already have these labels—ℓ_{State} and ℓ_{Exn}, respectively. Because reads are not visible, we can set $\ell_R = \top$. For other effects, the label ℓ_ε should capture who might observe *any* component of ε, so it should be a lower bound on the components of ε. That is, $\ell_{W,E} \sqsubseteq \ell_{\mathsf{State}}$ and $\ell_{W,E} \sqsubseteq \ell_{\mathsf{Exn}}$.

Now we can build our type-and-effect system, modifying the rules of the pure typing system. As in the pc case, most of the typing rules do not change ε, since most terms do not change the effects a program may run. For the same reasons as before, functions and the four rules we added explicitly for effects do change. Since λ-expressions execute effects when applied but not when defined, we take the same approach as before and record the effects in the type. We also modify the function protection rule as we did previously. This gives rise to the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \diamond \varepsilon}{\Gamma \vdash \lambda x : \tau_1.\, e : \tau_1 \xrightarrow{\varepsilon} \tau_2 \diamond \varnothing} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varepsilon_1} \tau_2 \diamond \varepsilon_2 \qquad \Gamma \vdash e_2 : \tau_1 \diamond \varepsilon_3}{\Gamma \vdash e_1\ e_2 : \tau_2 \diamond \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \qquad \frac{\ell \triangleleft \tau_2 \qquad \ell \sqsubseteq \ell_\varepsilon}{\ell \triangleleft \tau_1 \xrightarrow{\varepsilon} \tau_2}$$

The rules for our extended expressions must incur appropriate effects. Again, three are fairly simple.

$$\frac{}{\Gamma \vdash \mathsf{read} : \sigma \diamond R} \qquad \frac{\Gamma \vdash e : \sigma \diamond \varepsilon}{\Gamma \vdash \mathsf{write}(e) : \mathsf{unit} \diamond \varepsilon \cup W} \qquad \frac{}{\Gamma \vdash \mathsf{throw} : \tau \diamond E}$$

Reading a value gives an R effect, while $\mathsf{write}(e)$ evaluates e and therefore any effects e runs, and then runs a write effect. Throwing an exception creates an E effect, but as before, catching an exception is more complicated. A try-catch expression does not generate any new effects, though it can combine effects from both the try and catch blocks. More importantly, since we are still aiming to enforce security through our type-and-effect system, the security concerns surrounding control flow still apply. We therefore again require $\ell_{\mathsf{Exn}} \triangleleft \tau$. The resulting rule is as follows.

$$\frac{\Gamma \vdash e_1 : \tau \diamond \varepsilon_1 \cup E \qquad \Gamma \vdash e_2 : \tau \diamond \varepsilon_2 \qquad \ell_{\mathsf{Exn}} \triangleleft \tau}{\Gamma \vdash \mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\} : \tau \diamond \varepsilon_1 \cup \varepsilon_2}$$

We also modify the unlabel rule to prevent effects from leaking data. Specifically, if a program e_2 relies on data with label ℓ, the effects of e_2 can only be visible at or above ℓ. Including the existing output restriction from Section 2 gives us the following rule:

$$\frac{\Gamma \vdash e_1 : L_\ell\,(\tau_1) \diamond \varepsilon_1 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau_2 \diamond \varepsilon_2 \qquad \ell \triangleleft \tau_2 \qquad \ell \sqsubseteq \ell_{\varepsilon_2}}{\Gamma \vdash \mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \tau_2 \diamond \varepsilon_1 \cup \varepsilon_2}$$

Finally, as several of these rules require precisely equal effects, we include a rule allowing judgments to overstate a program's effect:

$$\frac{\Gamma \vdash e : \tau \diamond \varepsilon' \qquad \varepsilon' \subseteq \varepsilon}{\Gamma \vdash e : \tau \diamond \varepsilon}$$

We now have the mechanics we need to formally state a connection between the program-counter label and effects. Intuitively, a program with a sufficiently restrictive pc cannot have certain effects. We capture this intuition with the following lemma:

Lemma 1 (Connection between Program-Counter Label and Effects)**.** *The program-counter label forces effects to be well typed, so if* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$*, then* $\Gamma \vdash e : \tau \diamond \varepsilon$ *for some* ε*. Moreover, the* pc *controls which effects are possible. In particular:*

- *If* $\mathrm{pc} \not\sqsubseteq \ell_{\mathsf{State}}$ *and* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$*, then there exists an* ε *such that* $\Gamma \vdash e : \tau \diamond \varepsilon$ *and* $\mathrm{W} \notin \varepsilon$*.*
- *If* $\mathrm{pc} \not\sqsubseteq \ell_{\mathsf{Exn}}$ *and* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$*, then there exists an* ε *such that* $\Gamma \vdash e : \tau \diamond \varepsilon$ *and* $\mathrm{E} \notin \varepsilon$*.*

Since our type-and-effect system restricts to secure effects, the first part of Lemma 1 is non-trivial. Also note that the two type systems use slightly different type constructors for functions—the pc system includes labels while the type-and-effect system includes effects. We implicitly convert between the two here, as the label associated with each effect makes the conversion simple. See the accompanying technical report for the full details of the conversion.

Lemma 1 provides a direct connection between program-counter labels and effects. From the point-of-view we have been advocating—that program-counter labels limit programs to secure effects—this is the semantics of a program-counter label. We will discuss this semantics for program-counter labels in more depth in Section 5.

3.3 Effectful Noninterference Half-Off

We now aim to use our effect tracking to prove noninterference for our extended language. We use a strategy from work on the semantics of type-and-effect systems and give meaning to effects via translation into pure programs. Importantly, if we do this in such a way that our notions of "low-equivalent" match up, we can get noninterference automatically.

Our translation uses monads to represent effects, as is common. In fact, we have one monad per (set of) effect(s), defined as follows. For technical reasons, we use the same monad for $\{\mathrm{W}\}$ and $\{\mathrm{R}, \mathrm{W}\}$, as well as for $\{\mathrm{W}, \mathrm{E}\}$ and $\{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$.[2] We also assume for simplicity that σ, the type of our state cell, already has at least ℓ_{State} sensitivity—that is, $\ell_{\mathsf{State}} \triangleleft \sigma$—allowing us to omit explicit use of ℓ_{State}. If this were not the case, we would use $L_{\ell_{\mathsf{State}}}(\sigma)$ instead of σ.

ε	$P_\varepsilon(\tau)$
$\varnothing$	τ
$\{\mathrm{R}\}$	$\sigma \to \tau$
$\{\mathrm{E}\}$	$L_{\ell_{\mathsf{Exn}}}(\mathsf{unit} + \tau)$
$\{\mathrm{W}\}$ and $\{\mathrm{R}, \mathrm{W}\}$	$\sigma \to (\tau \times \sigma)$
$\{\mathrm{R}, \mathrm{E}\}$	$\sigma \to L_{\ell_{\mathsf{Exn}}}(\mathsf{unit} + \tau)$
$\{\mathrm{W}, \mathrm{E}\}$ and $\{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$	$\sigma \to (L_{\ell_{\mathsf{Exn}}}(\mathsf{unit} + \tau) \times \sigma)$

We refer to the monad for a (set of) effect(s) ε as P_ε. We use this notation instead of the more-traditional $M\_$ as we will generalize to a productor in Section 5. These monads are standard, but they are not automatic. Instead, they reflect some choices about the semantics of programs. For instance, the fact that the monad for the set $\{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$ is $\sigma \to (\mathsf{unit} + \tau) \times \sigma$ instead of $\mathsf{unit} + (\sigma \to \tau \times \sigma)$ reflects the fact that state persists even when an exception is thrown.

Note that we can define three special kinds of programs:

- For any set ε and program $\Gamma, x : \tau_1 \vdash p : P_\varepsilon(\tau_2)$, we can define $\Gamma, x : P_\varepsilon(\tau_1) \vdash \mathsf{bind}_\varepsilon(p) : P_\varepsilon(\tau_2)$.
- For any type τ and set ε, we can define $\eta_\varepsilon : \tau \to P_\varepsilon(\tau)$.

[2] This is because the *writer* monad, for the write effect without read, is not a monad unless σ is a monoid. However, the *state* monad, for read and write effects, is a cartesian monad no matter the type of σ.

- For any type τ and pair of sets ε_1 and ε_2 such that $\varepsilon_1 \subseteq \varepsilon_2$, we can define a program $\text{coerce}_{\varepsilon_1 \mapsto \varepsilon_2} : P_{\varepsilon_1}(\tau) \to P_{\varepsilon_2}(\tau)$.

This (along with some easily-proven properties of these programs) makes $P_\_$ an *indexed monad* [Orchard et al. 2014; Wadler and Thiemann 1998], which is a mathematical object that gives semantics to systems of effects. Note that this requires $L_\_$ itself to be an indexed monad, which was proven by Abadi et al. [1999]. As the name suggests, indexed monads are a generalization of monads. Indexed monads also give a standard way to translate effectful programs into pure programs, transforming a derivation of $\Gamma \vdash e : \tau \diamond \varepsilon$ into a program e' such that $\Gamma \vdash e' : P_\varepsilon(\tau)$. See the accompanying technical report for the full translation.

This translation generates an important security result. Because all well-typed pure programs guarantee noninterference, we can extend that result to our effectful language if our translation has two specific properties. First, it must be sound. Indeed, if we omit any of the careful reasoning about exceptions from Section 3.2 our translation would be unsound. The point at which the soundness proof breaks down is, however, often informative. For example, when translating Program 1, our translation would need to return a value of type $L_{\ell_{\text{Exn}}}\,(\text{unit} + \text{unit}) \times \sigma$ after unlabeling a value of type $L_{\ell_{\text{Exn}}}\,(\text{unit} + \text{unit})$. Satisfying the premise of the unlabel rule that the removed label protects the output type forces exactly the $\ell_{\text{Exn}} \sqsubseteq \ell_{\text{State}}$ assumption we made above.

The second requirement to obtain effectful noninterference is that our monadic translation faithfully translates our effectful notion of equivalence to our pure one. In this case it does because monadic programs simulate effectful programs. Without labels, this is a well-known theorem of Wadler and Thiemann [1998], adding labels does not significantly change the proof. We can therefore use our type-and-effect system and monadic translation to make a strong claim of noninterference for the pc system.

Theorem 2 (Noninterference for State and Exceptions)**.** *For all expressions e_1 and e_2, and for all $\ell \in \mathcal{L}$, if $\Gamma \diamond \text{pc} \vdash e_1 : \tau$ and $\Gamma \diamond \text{pc} \vdash e_2 : \tau$, and $\ell \triangleleft \tau$ and $\ell \sqsubseteq \text{pc}$, then for all labels $\ell_{\text{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\text{Atk}}$ or $e_1 \cong^{\text{SE}}_{\ell_{\text{Atk}}} e_2$.*

Proof. This is a special case of the Noninterference Half-Off Theorem (Theorem 5) which uses the fact that effectful programs are equivalent if their monadic translations are equivalent. Theorem 5 also relies on the fact that pure programs are noninterfering (Theorem 1). □

The requirement that $\ell \sqsubseteq \text{pc}$ may appear odd next to classic definitions of noninterference. We require such a flow because our contextual notion of equivalence treats e_1 and e_2 as program inputs, but they may have effects. This requirement constrains those effects so that if $\ell \not\sqsubseteq \ell_{\text{Atk}}$, then ℓ_{Atk} will be unable to see them. In most classic noninterference statements, the inputs are values, meaning this restriction is unnecessary as any well-typed value type-checks with $\text{pc} = \top$.

4 EXAMPLE: TERMINATION-SENSITIVE NONINTERFERENCE

Type-and-effect systems can tell us more about programs than whether they access state or throw exceptions. One classic application is checking termination by considering possible nontermination to be an effect. We show that we can treat possible nontermination as a *secure* effect. This explains the role of program-counter labels in ruling out termination leaks. That is, we show how *termination-sensitive noninterference* falls out of our framework.

The fragment of DCC we use in Section 2 is strongly-normalizing; thus, all programs terminate. Even the extensions in Section 3 did not allow for nonterminating behavior, since we require that the type of the state cell is first order. We can, however, easily add a standard fixpoint operator:

$$\text{Expressions} \quad e \quad ::= \quad \cdots \mid \text{fix}\, f : \tau.\, e$$

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, f : \tau. e : \tau} \qquad \mathsf{fix}\, f : \tau. e \longrightarrow e[f \mapsto \mathsf{fix}\, f : \tau. e]$$

Because programs may not terminate in this extended language, the fact that our definition of ℓ-equivalence (Definition 1) allows for different termination behavior matters. In particular, our previous noninterference theorem (Theorem 1) now says that if both programs terminate, they must produce the same value. If *either* program diverges, however, it makes no guarantees.

This guarantee models an attacker who cannot tell if a program has failed to terminate, or if it will produce an output on the next step. That is, the attacker is *insensitive* to termination behavior. This notion of noninterference is therefore called *termination-insensitive* noninterference.

While DCC with the call-by-value semantics we are using enforces termination-insensitive noninterference [Abadi et al. 1999; Algehed and Bernardy 2019; Bowman and Ahmed 2015; Heintze and Riecke 1998], termination channels can leak arbitrary amounts of data [Askarov et al. 2008]. We would therefore like to remove the strong assumption that attackers cannot use those channels. To do so, we define a stronger form of equivalence, *termination-sensitive ℓ-equivalence*, and use it to analyze security.

Definition 3 (Termination-Sensitive ℓ-Equivalence)**.** We say e_1 is *termination-sensitive ℓ-equivalent* to e_2, denoted $e_1 \cong_\ell^{\mathrm{TS}} e_2$, if for all expression contexts C such that $\vdash C[e_i] : L_\ell\,(\mathsf{unit} + \mathsf{unit})$ for both $i = 1, 2$, then $C[e_1] \longrightarrow^* v$ if and only if $C[e_2] \longrightarrow^* v$.

We follow the same approach as in Section 3 to ensure noninterference with respect to this stronger definition: we restrict effects with a pc label, build a corresponding type-and-effect system, and prove security by translating to DCC with its termination-insensitive guarantee. Because nontermination from fix is the only effect, this process is considerably simpler than in Section 3.

Traditionally, termination-sensitive noninterference assumes all attackers can see whether or not a program terminates. We take a more general approach and assume that some attackers are termination-sensitive, while others may not be. Specifically, we imagine there is a label $\ell_{\mathsf{PNT}} \in \mathcal{L}$ such that any attacker who can read ℓ_{PNT} will eventually infer information from nontermination, but others will not. This gives rise to the following rule:

$$\frac{\Gamma, f : \tau \diamond \mathrm{pc} \vdash e : \tau \qquad \mathrm{pc} \sqsubseteq \ell_{\mathsf{PNT}}}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{fix}\, f : \tau. e : \tau}$$

The rule ensures that only data available at label ℓ_{PNT}—visible to any termination-sensitive attacker—can influence the program's termination behavior. This single label-based rule allows us to model termination-insensitivity by setting $\ell_{\mathsf{PNT}} = \top$, model traditional termination-sensitivity using a bottom label $\bot$ by setting $\ell_{\mathsf{PNT}} = \bot$, or express policies about other levels of termination visibility.

We can now move on to the type-and-effect system. This time it has only two possible effects: $\varnothing$ or PNT with labels $\top$ and ℓ_{PNT}, respectively. The typing rule for the fixed-point operator is then:

$$\frac{\Gamma, f : \tau \vdash e : \tau \diamond \varepsilon}{\Gamma \vdash \mathsf{fix}\, f : \tau. e : \tau \diamond \mathrm{PNT}}$$

The unlabel rule constrains effects in the same way as in Section 3, and no other typing rules change or constrain effects. We also change function types as in Section 3.

Finally, we translate this type-and-effect system into pure DCC using a monadic translation. Unfortunately, this time there is no such monad definable in the language of Section 2. Luckily, the original definition of DCC [Abadi et al. 1999] allowed for nontermination with a fixed-point operator, but only if the result was a *pointed* type. We thus add fixed points and pointed types—the

parts of DCC we omitted from Section 2—and use pointed types to define our monad.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \cdots \mid \tau_\bot \\ \text{Expressions} & e & ::= & \cdots \mid \mathsf{lift}(e) \mid \mathsf{seq}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}\ f : \tau.\, e \\ \text{Evaluation Contexts} & E & ::= & \cdots \mid \mathsf{lift}(E) \mid \mathsf{seq}\ x = E\ \mathsf{in}\ e \end{array}$$

The type $\tau_\bot$ represents a version of τ that supports fixed points, while the expression $\mathsf{lift}(e)$ lifts the expression e from type τ to $\tau_\bot$. The expression $\mathsf{seq}\ x = e_1\ \mathsf{in}\ e_2$ waits for e_1 to terminate and, if it does, binds the result to x in e_2. Finally, the term $\mathsf{fix}\ f : \tau.\, e$ defines a fixpoint.

We then define a judgment determining when a type is a pointed type as follows:

$$\frac{}{\vdash \tau_\bot\ \mathsf{ptd}} \qquad \frac{\vdash \tau_1\ \mathsf{ptd} \qquad \vdash \tau_2\ \mathsf{ptd}}{\vdash \tau_1 \times \tau_2\ \mathsf{ptd}} \qquad \frac{\vdash \tau\ \mathsf{ptd}}{\vdash L_\ell(\tau)\ \mathsf{ptd}} \qquad \frac{\vdash \tau_2\ \mathsf{ptd}}{\vdash \tau_1 \to \tau_2\ \mathsf{ptd}}$$

This allows us to state the following typing and semantic rules for the newly-added terms:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{lift}(e) : \tau_\bot} \qquad \frac{\Gamma \vdash e_1 : \tau_{1\bot} \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \vdash \tau_2\ \mathsf{ptd}}{\Gamma \vdash \mathsf{seq}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\Gamma, f : \tau \vdash e : \tau \qquad \vdash \tau\ \mathsf{ptd}}{\Gamma \vdash \mathsf{fix}\ f : \tau.\, e : \tau}$$

$$\mathsf{seq}\ x = \mathsf{lift}(v)\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]$$

Because the possibility of nontermination is limited to pointed types, we consider the full DCC to be pure for our purposes. That is, we consider programs which are nonterminating, but which have a pointed type, pure. To see why this is justified, we have to ask what we consider an effect. We can probe this by considering the example from Section 3: why do we consider a program which returns a value of type $\sigma \to (\tau \times \sigma)$ to be pure, but not a program which accesses state and returns a value of type τ? After all, they can encode the same computations. Intuitively, though, we have translated accesses to state to operations provided by the more-complex type; namely, reads have been replaced by usage of the parameter of type σ, and writes by returning an appropriate result of type σ. In the current case, we have translated fixpoint computations which can take place anywhere with a fixpoint operation provided by a more-complex type. This eases reasoning in many settings. For instance, when attempting a proof by logical relations, all reasoning about nontermination can now be located in pointed types.

Some readers may remain skeptical of the application of the word "pure" to DCC extended with pointed types. It is therefore worth noting that to use noninterference half-off, we only need a monad which can represent the effect that we want to reason about in a language where it is easier to prove noninterference. Proving noninterference in full (call-by-value) DCC with pointed types is easier than in our pc system for two reasons. First, possible nontermination is located by types, which makes many proof techniques easer, as noted above. Second, call-by-value DCC with pointed types enforces *termination-insensitive* noninterference, which is generally simpler to prove than termination-sensitive noninterference, which our pc system enforces.

While pointed types give access to fixpoint operators, the translation of the unlabel rule fails whenever a program returns a $\tau_\bot$, because $\tau_\bot$ protects no labels. This was Abadi et al.'s [1999] original design, to ensure that labeled data can never determine the termination behavior of a program. This is too restrictive for us, since we want to allow data up to label ℓ_{PNT} to influence a program's termination behavior. Of course, the label must also be allowed to influence any output the program produces if it does terminate. This leads to the following rule:

$$\frac{\ell \sqsubseteq \ell_{\mathsf{PNT}} \qquad \ell \lhd \tau}{\ell \lhd \tau_\bot}$$

Note that, even when setting $\ell_{\mathsf{PNT}} = \bot$, this rule allows *public* data to influence termination behavior. We therefore differ slightly from Abadi et al.'s [1999] definition in systems that distinguish public data from unlabeled data.

With this rule, the traditional monadic translation works, which tells us that the type-and-effect system indeed enforces noninterference. Now we need to connect the pc system to the type-and-effect system in an analogous way to Lemma 1. If the pc cannot influence ℓ_{PNT}, then the program *must* terminate. We can formalize this as follows:[3]

Lemma 2. *If* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$, *then* $\Gamma \vdash e : \tau \diamond \mathrm{PNT}$. *Moreover, if* $\mathrm{pc} \not\sqsubseteq \ell_{\mathsf{PNT}}$, *then* $\Gamma \vdash e : \tau \diamond \varnothing$.

The first statment is again non-trivial since our type-and-effect system restricts to secure effects.

Lemma 2 is a very powerful guarantee. Combined with a monadic translation based on pointed types (which can be found in the accompanying technical report), it tells us that if the pc is too high, e will terminate. Thus, if some data determines whether or not a program terminates, it must be visible to any attacker who can see termination behavior. Formalizing this, we get the following guarantee:

Theorem 3 (Termination-Sensitive Noninterference)**.** *For* $\ell \in \mathcal{L}$ *and expressions* e_1 *and* e_2, *if* $\Gamma \diamond \mathrm{pc} \vdash e_1 : \tau$ *and* $\Gamma \diamond \mathrm{pc} \vdash e_2 : \tau$, *and* $\ell \lhd \tau$ *and* $\ell \sqsubseteq \mathrm{pc}$, *then for all labels* $\ell_{\mathsf{Atk}} \in \mathcal{L}$ *where* $\ell_{\mathsf{PNT}} \sqsubseteq \ell_{\mathsf{Atk}}$, *either* $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ *or* $e_1 \cong^{\mathrm{TS}}_{\ell_{\mathsf{Atk}}} e_2$.

This is a special case of Theorem 5. As with both previous noninterference theorems (Theorems 1 and 2), this theorem says data at label ℓ cannot leak to an attacker who cannot distinguish values at level ℓ. This time, however, the attacker can glean information from nontermination. Moreover, if we set $\ell_{\mathsf{PNT}} = \bot$, then $\bot \sqsubseteq \ell_{\mathsf{Atk}}$ for all ℓ_{Atk}, so if $\ell \neq \bot$ we get classic termination-sensitive noninterference.

Note that if either e_1 or e_2 may diverge, the theorem always allows a termination-sensitive attacker to distinguish them. In this case, Lemma 2 ensures $\mathrm{pc} \sqsubseteq \ell_{\mathsf{PNT}}$, which then guarantees $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ by transitivity.

5 A FRAMEWORK FOR EFFECTFUL LABELED LANGUAGES

We have now twice given semantics to pc systems using type-and-effect systems and monadic translations. The ability to give semantics to not just traditional effects like state, but combinations of effects and more unusual effects, like nontermination, demonstrates the power of this technique. We now generalize these ideas by moving to a semantic framework that does not lock us into a single language. Instead, we provide a set of typing rules and equations specifying the language features that our semantics require.

This approach allows us to describe the semantics of a large class of languages at once. By making our framework as general as possible, we learn about what features a language needs to make our semantics work. More importantly, we can also be sure that we do not rely on a *lack* of other language features. Thus, we can make strong semantic and security guarantees about any language that admits the rules of our framework, regardless of what other features may be present. We develop our framework by looking at the commonalities in our examples and determining which properties are necessary to obtain the security results.

In our examples in Sections 3 and 4, we developed a semantics for pc labels via two layers of translation. We first translated from a pc system to a type-and-effect system, and then to a pure language via monadic translation. Our general framework makes the same division. We begin by focusing on the monadic translation and discuss the first layer in Section 6.

[3] As with Lemma 1, we implicitly assume a simple translation between the two sets of type constructors (see the accompanying technical report).

The monadic translation required two languages, one effectful and one pure, a set of effects $\mathcal{E}$, and a translation capturing effectful programs as monadic pure ones. The effectful language used a type-and-effect system with judgements of the form $\Gamma \vdash e : \tau \diamond \varepsilon$ with $\varepsilon \in \mathcal{E}$, while the pure language had judgements of the form $\Gamma \vdash e : \tau$. We used an indexed monad to provide a type transformer $P_\varepsilon(-)$ for each effect $\varepsilon \in \mathcal{E}$, so that $P_\varepsilon(\tau)$ was the pure type resulting from from translating an effectful program with type τ and effect ε.

Our framework generalizes this approach. We again have two languages, one effectful and one pure. Here the type-and-effect judgements in the effectful language take the form $t \vdash_{\mathrm{E}} p \dashv t' \diamond \varepsilon$ and the typing judgements of the pure language take the form $\tau \vdash_{\mathrm{P}} \rho \dashv \tau'$. For clarity, we annotate the turnstyles and color judgments in the two languages differently. We also use Roman letters to refer to types and programs in the effectful language and Greek letters in the pure language.

An eagle-eyed reader will have noticed that our examples have a context on the left, while our generalization has only a single type t or τ. This is due to the categorical nature of monadic semantics, which makes it most natural to talk about *single-input, single-output* systems. A common trick, which we will use here, is to have t and τ represent *contexts*, rather than types. However, the multiple-input, single-output nature of our example languages can actually give more complex structure. That structure allows us to interpret the categorical operations as applying to a single input of programs, which we do freely in our examples. We formalize the technical details of this transformation in Appendix A.

In both of our examples, our set of effects $\mathcal{E}$ were the power set of the individual effects in the language. The result was a lattice structure that we leveraged to define the effect of sequentially composing effectful programs. Sequential composition took the form of function application. If we wrap e_2 in a lambda and provide e_1 as an argument, we first execute e_1 and then e_2. In particular, the abstraction, application, and effect variance rules in Section 3.2 combined prove that the following rule is admissible.

$$\frac{\Gamma \vdash e_1 : \tau_1 \diamond \varepsilon_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \diamond \varepsilon_2 \qquad \varepsilon_1 \cup \varepsilon_2 \subseteq \varepsilon}{\Gamma \vdash (\lambda x.\, e_1)\ e_2 : \tau_2 \diamond \varepsilon}$$

For our general framework, we also require both a set of effects $\mathcal{E}$ and a sequential composition operation that we denote $p_1 \,;\, p_2$. In our examples, the lattice structure of $\mathcal{E}$ allowed us to compose any pair of effectful programs. In general, however, this requirement is not only unnecessary, it is overly restrictive. It is sometimes useful for the composition operation to be *partial*, allowing only certain sequences of effects—and certain sequences of effectful programs—to compose. In particular, we discuss in Section 5.1 how using partiality, we can lift a seemingly-arbitrary restriction in our treatment of exceptions and state. For our framework, we therefore turn to Tate's [2013] *effector*, which was designed as the minimal structure required to give meaning to such compositions.

An effector is a set $\mathcal{E}$ with a relation $[-, \ldots, -] \leq -$ defining how the effects can compose. Intuitively, $[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon$ means that sequentially composing n programs with effects ε_1 through ε_n, respectively, can result in a program with effect ε. Note that n may in particular be zero or one, where $[] \leq \varepsilon$ means that a program judged to have effect ε may be pure, and $[\varepsilon] \leq \varepsilon'$ means that a program judged to have effect ε' may also have effect ε. The relation must follow appropriate versions of identity and associativity laws, reflecting these intuitions [Tate 2013, Section 5]. For the power-set lattices from Sections 3 and 4, we can simply define $[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon$ as $\varepsilon_1 \cup \cdots \cup \varepsilon_n \subseteq \varepsilon$.

An effector allows us to state that composition of effectful programs is only required when their types match and their effects compose. We formalize this rule as $\textsc{Seq}_{\leq}$ in Figure 3, which also requires the larger program's effects to be a valid composition of the individual effects. Note that, as with composing effects, $\textsc{Seq}_{\leq}$ requires composition of zero or more programs. The above typing rule only demonstrates composition in our example languages for pairs of programs. Using pairwise

$$\textsc{Seq}_{\leq}\ \frac{t_0 \vdash_{\mathrm{E}} p_1 \dashv t_1 \diamond \varepsilon_1 \quad \cdots \quad t_{n-1} \vdash_{\mathrm{E}} p_n \dashv t_n \diamond \varepsilon_n \qquad [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{t_0 \vdash_{\mathrm{E}} p_1 \,;\cdots;\, p_n \dashv t_n \diamond \varepsilon} \qquad \textsc{Seq}\ \frac{\tau_0 \vdash_{\mathrm{P}} \rho_1 \dashv \tau_1 \quad \cdots \quad \tau_{n-1} \vdash_{\mathrm{P}} \rho_n \dashv \tau_n}{\tau_0 \vdash_{\mathrm{P}} \rho_1 \,;\cdots;\, \rho_n \dashv \tau_n}$$

$$\textsc{Capture}\ \frac{t \vdash_{\mathrm{E}} p \dashv t' \diamond \varepsilon}{(\!|t|\!) \vdash_{\mathrm{P}} \lfloor p \rfloor_\varepsilon \dashv P_\varepsilon(\!|t'|\!)}$$

$$\textsc{Map}\ \frac{\tau \vdash_{\mathrm{P}} \rho \dashv \tau'}{P_\varepsilon(\tau) \vdash_{\mathrm{P}} \mathsf{map}_\varepsilon(\rho) \dashv P_\varepsilon(\tau')} \qquad \textsc{Join}\ \frac{[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{P_{\varepsilon_1}(\cdots P_{\varepsilon_n}(\tau)\cdots) \vdash_{\mathrm{P}} \mathsf{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon} \dashv P_\varepsilon(\tau)}$$

$$\textsc{CapturedSeq}\ \frac{t_0 \vdash_{\mathrm{E}} p_1 \dashv t_1 \diamond \varepsilon_1 \quad \cdots \quad t_{n-1} \vdash_{\mathrm{E}} p_n \dashv t_n \diamond \varepsilon_n \qquad [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{\lfloor p_1 \,;\cdots;\, p_n \rfloor_\varepsilon \;\overset{(\!|t_0|\!), P_\varepsilon(\!|t_n|\!)}{=\!=\!=}\; \lfloor p_1 \rfloor_{\varepsilon_1} \,;\, \mathsf{map}_{\varepsilon_1}(\lfloor p_2 \rfloor_{\varepsilon_2} \,;\, \mathsf{map}_{\varepsilon_2}(\cdots(\lfloor p_n \rfloor_{\varepsilon_n}))) \,;\, \mathsf{join}_{[\varepsilon_1,\cdots,\varepsilon_n],\varepsilon}}$$

$$\textsc{ProtectC}\ \frac{\ell \sqsubseteq \ell_\varepsilon \qquad \ell \lhd t}{\ell \lhd P_\varepsilon(\!|t|\!)} \qquad \textsc{EquivCap}\ \frac{\lfloor p \rfloor_\varepsilon \approx_\ell \lfloor q \rfloor_\varepsilon}{p \cong_\ell^\varepsilon q}$$

Fig. 3. Main Rules for Semantic Framework of Labeled Pure and Effectful Programming Languages

composition, we can inductively define composition of any larger number of programs. Composing a single program is just that program unmodified, and nullary composition is the identity $\lambda x.\, x$.

Note that our pure language, DCC, also used function application for sequential composition. Our general framework requires sequential composition for the pure language in the Seq rule. However, because the language is pure, there is no concern about when effects may compose, so we require pure programs to compose whenever the output type of the one matches the input type of the next.

We now turn to the monadic translation itself. In both examples, we handled multiple possible sets of effects by using a monad $P_\varepsilon(-)$ indexed on $\varepsilon \in \mathcal{E}$. We then required a translation that took a well-typed effectful program e where $\Gamma \vdash e : \tau \diamond \varepsilon$ to a well-typed pure program e' where $\Gamma \vdash e' : P_\varepsilon(\tau)$. The Capture rule incorporates this translation, which we denote $\lfloor - \rfloor_-$, into our framework.[4] Note that the pure and effectful languages may use different type constructors. For instance, our example effectful languages annotated function types with effects, but DCC does not. While we implicitly translated $\tau_1 \xrightarrow{\varepsilon} \tau_2$ to $\tau_1 \to P_\varepsilon(\tau_2)$ in Sections 3 and 4, our general framework makes this translation explicit and denotes it $(\!|-|\!)$.

The lattice structure of $\mathcal{E}$ coupled with the η, bind, and coerce operations meant that $P_\varepsilon(-)$ formed an indexed monad [Orchard et al. 2014; Wadler and Thiemann 1998]. As our framework generalizes $\mathcal{E}$ to an effector, it correspondingly generalizes P_ε to a *productor* [Tate 2013], a generalization of both indexed monads and graded monads [Fujii et al. 2016; Katsumata 2014], which require $\mathcal{E}$ to be an ordered monoid.

Unsurprisingly, productors require structure similar to η, bind, and coerce, which we specify in the Map and Join rules. Map requires, for each effect ε, a pure program transformer map_ε that takes a program from τ to τ' and produces a program from $P_\varepsilon(\tau)$ to $P_\varepsilon(\tau')$. Join requires a program $\mathsf{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon}$ whenever $[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon$ that translates a pure program capturing effects ε_1 through ε_n in order into a pure program that captures effect ε. Notably, every indexed monad is also a productor and any productor whose effector $\mathcal{E}$ forms a lattice is also an indexed monad [Tate 2013].

[4] Tate [2013] calls capture *thunking*, to bring attention to the similarity with the familiar concept in functional languages. We use the word "capture" here because it better-fits how we use the concept.

CapturedSeq says that we can define the capture of the sequential composition of effectful programs by capturing each program individually and using pure composition, map, and join appropriately. Here $\rho_1 \underset{\tau_1,\tau_2}{=\!=} \rho_2$ means "ρ_1 and ρ_2 are equal as pure programs from τ_1 to τ_2." Those familiar with monads will recognize this as a generalization of monadic bind. Indeed, a monad (or indexed monad) requires $[\varepsilon, \varepsilon] \leq \varepsilon$ for all $\varepsilon \in \mathcal{E}$, allowing us to define $\mathsf{bind}_\varepsilon(\rho) = \mathsf{map}_\varepsilon(\rho);\mathsf{join}_{[\varepsilon,\varepsilon],\varepsilon}$. Those familiar with category theory will also recognize this as Klesili composition.

Our framework up to this point is developed directly from Tate [2013] and gives rules defining the semantics of a type-and-effect system using a productor. However, giving semantics to program-counter labels also requires our effects to be *secure*. For that, we need two more laws.

In Sections 3 and 4 we went to great pains to ensure that the labels on our pure monads properly represented the visibility of our effects. In Section 4 we did this directly with the protection relation on a pointed type $\tau_\bot$. In Section 3 it was a bit more complicated. We required that $\ell_{\mathsf{State}} \triangleleft \sigma$, the type of the state cell, and $P_{\mathbb{E}}(-)$ explicitly wrapped its output with ℓ_{Exn} to constrain visibility. The end goal was that, if both the output and effects of the original program had sensitivity at least ℓ, then the output of the translated pure program must also have sensitivity at least ℓ. In order to codify that goal in our framework, we assume both the pure and effectful languages have a protection relation defining sensitivity and relate them in exactly this way. We then require ProtectC to hold in our framework, which is exactly the rule that we used in Section 4. We can furthermore show that ProtectC holds in the example from Section 3 by case analysis on the effects ε, using the fact that $\ell_{\mathsf{Exn}} \sqsubseteq \ell_{\mathsf{State}}$ and that $\ell_{\mathsf{State}} \triangleleft \sigma$.

Note that, while we assume that both languages define a protection relation, it does not have to be the main security mechanism of the language. Instead, protection can be defined in terms of that security mechanism. For instance, a fine-grained system can define a protection relation using the fact that every type has an associated label. Rajani and Garg [2018] give an example of this sort of definition in their translation of a fine-grained system into a coarse-grained one.

In Section 3 we used a well-known monadic translation that creates a simulation. That is, if $\langle p, s\rangle \longrightarrow \langle p', s'\rangle$, then $(\lfloor p \rfloor_\varepsilon\ s) \longrightarrow^* (\lfloor p' \rfloor_\varepsilon\ s')$. This property tells us that any context that distinguishes p from q translates to a context that distinguishes $\lfloor p \rfloor_\varepsilon$ from $\lfloor q \rfloor_\varepsilon$. In other words, effectful programs must be indistinguishable whenever their captured pure counterparts are. Because noninterference specifies that an attacker *cannot* distinguish programs whose outputs (including effects) are highly sensitive, this property allows us to lift noninterference from the pure language to the effectful one. While encoding operational semantics, contextual equivalence, and simulation into our framework would require a lot of work, we can directly encode this last insight. We do so with EquivCap. We require two equivalence relations parameterized on a label ℓ, one for pure programs and one for effectful ones. Intuitively, two programs are equivalent at ℓ if they are indistinguishable to attacker who can read values only up to level ℓ. The equivalence of effectful programs also takes an effect parameter, allowing the definition to account for visible effects. EquivCap demands only that they capture the correspondence we relied on above.

EquivCap is also the rule that requires the monadic translation to capture the semantics of the effect. For instance, if we replace the monadic translation in Section 3 with a faulty one, we will not enjoy EquivCap with contextual equivalences. Imagine in particular a translation that treated the state as a constant and discarded writes. The effectful programs write(3) and write(4) would then be distinguishable by a context that reads the state and compared it to 3, but they would translate to identical—and therefore indistinguishable—pure programs. Of course, we could change the notion of effectful equivalence as well, essentially changing the semantics of our effectful language. However, as we will see in Section 6.1, that equivalence defines the meaning of noninterference

in our framework, so the guarantees and semantics given by our framework will reflect this new equational semantics.

With these rules in place, the framework allows us to prove a strong and general security theorem for effectful programming languages. Moreover, it connects to the category theory, giving us powerful tools for reasoning about effects.

5.1 Better State and Exceptions via Partialiaty

In Section 3 we avoided a concern about state and exceptions combining to leak data by assuming that $\ell_{\mathsf{Exn}} \sqsubseteq \ell_{\mathsf{State}}$. Recall that when we compose a program p that may throw an exception with a program q that may write state, the write—or lack thereof—can leak whether q executed and thus whether p threw an exception. We therefore restricted our system to require that anyone who could observe the state could also see any exceptions.

We now modify our rules from Section 3.2 to remove this restriction while still ensuring that state and exceptions interact securely. We use the translation from Section 3.3 as a guide. When composing programs p and q where p may throw an exception and q has effect ε, $\lfloor p \rfloor$ wraps its output in ℓ_{Exn}. This represents the fact that, to read that data, an attacker must be able to see whether an exception has been thrown (since if one had, that data would not exist). To compose these programs, then, anyone who can see the results of any effects in q must be able to see whether an exception occurred. In other words, for the translation to be well-typed, we require $\ell_{\mathsf{Exn}} \sqsubseteq \ell_{\varepsilon}$.

By assuming $\ell_{\mathsf{Exn}} \sqsubseteq \ell_{\mathsf{State}}$, we assumed an adversary who could see the results of any effect could see if an exception was thrown. This made the requirement above trivial, but it prevented the language from representing e.g., writes that were more secret than exceptions. We could alternatively enforce noninterference by restricting composition of effects as follows:

$$\frac{\varepsilon_1 \cup \varepsilon_2 \subseteq \varepsilon_3 \qquad E \notin \varepsilon_1}{[\varepsilon_1, \varepsilon_2] \preceq \varepsilon_3} \qquad\qquad \frac{\varepsilon_1 \cup \varepsilon_2 \subseteq \varepsilon_3 \qquad \ell_{\mathsf{Exn}} \sqsubseteq \ell_{\varepsilon_2}}{[\varepsilon_1, \varepsilon_2] \preceq \varepsilon_3}$$

Not that this makes the composition relation partial, since not all pairs of effects compose. This means that $\mathcal{E}$ is no longer a lattice or even an ordered monoid. As a result, the translation we defined in Section 3.3 is no longer an indexed monad, or even a graded monad. This solution relies on the generality of effectors and productors.

This precise modification to the composition rules—and the partiality that results—is critically important for more realistic languages. For example, FlowCaml [Pottier and Simonet 2002] includes multiple exception types and general mutable reference cells with different types. In that setting, a program p that may throw an exception with label ℓ can safely compose with a program q that writes data exclusively at or above label ℓ.

6 THE NONINTERFERENCE HALF-OFF THEOREM

Our main aim is to show how, given a pc system, we can give semantics to the pc label and prove noninterference for that system. We do this by translation to a type-and-effect system, which we give semantics using the framework from Section 5. This is a powerful and general result, but it requires us to first demonstrate the security of the type-and-effect system itself.

6.1 Type-and-Effect Noninterference

In Sections 3 and 4 we leveraged the ability to translate effects into a pure language to simplify reasoning about noninterference. This allowed us to prove noninterference for the effectful language while only proving it directly for the pure part of the language. Since the framework of Section 5 specifies when this is possible, we would like to prove noninterference for any languages which admit the rules in Figure 3.

As in our examples, noninterference formalizes the intuition that adversary at label ℓ_{Atk} can only distinguish data and effects at or below label ℓ_{Atk}. We again define the label of data using a protection relation, though this time we leave the details of that relation abstract. We also assign a label ℓ_ε to the effect ε to represent ε's sensitivity. Finally, as each language has a different notion of equivalence, use an abstract notion of equivalence $\equiv_{\_}$ parameterized on labels.

Definition 4 (Abstract Noninterference)**.** Let r be a program such that $t_1 \vdash_{\text{E}} r \dashv t_2 \diamond \varepsilon_1$. We say that r is *noninterfering with respect to* $\equiv_{\_}$ if, for all labels $\ell \in \mathcal{L}$ and programs p and q such that

(1) $t_3 \vdash_{\text{E}} p \dashv t_1 \diamond \varepsilon_2$ and $t_3 \vdash_{\text{E}} q \dashv t_1 \diamond \varepsilon_2$ with $[\varepsilon_2, \varepsilon_1] \leq \varepsilon$
(2) $\ell \triangleleft t_1$ and $\ell \sqsubseteq \ell_{\varepsilon_2}$

then for all labels $\ell_{\text{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\text{Atk}}$ or $p \,;\, r \equiv_{\ell_{\text{Atk}}} q \,;\, r$.

In the above definition, condition 1 requires the sequential compositions $p \,;\, r$ and $q \,;\, r$ to be well-typed, while p and q each produce effects (at most) ε_2. This sequential composition represents providing two different inputs to the program r, abstracting the contextual equivalence we used in Sections 2, 3, and 4. Condition 2 requires that the sensitivity of both the type, t_1, and the effects, ε_2, of the input programs be at least ℓ. Intuitively, the conclusion says that an attacker can only use r to distinguish between p and q if they could already see the effects or outputs of p and q, and thus distinguish them without r.

We also allow the definition to apply to pure programs, replacing type-and-effect judgements with pure judgements and disregarding other references to effects. Our framework's rules are then sufficient to transfer a noninterference result from pure programs to effectful ones.

Theorem 4 (Type-and-Effect Noninterference)**.** *For any system satisfying all rules in Figure 3 where every well-typed pure program is noninterfering with respect to* $\approx_{\_}$*, then every program well-typed in the type-and-effect system is noninterfering with respect to* $\cong^{\varepsilon}_{\_}$*.*

Proof. Unfolding Definition 4, we have programs p, q, and r and a label ℓ such that

(1) $t_1 \vdash_{\text{E}} r \dashv t_2 \diamond \varepsilon_1$,
(2) $t_3 \vdash_{\text{E}} p \dashv t_1 \diamond \varepsilon_2$ and $t_3 \vdash_{\text{E}} q \dashv t_1 \diamond \varepsilon_2$ with $[\varepsilon_2, \varepsilon_1] \leq \varepsilon$,
(3) $\ell \triangleleft t_1$ and $\ell \sqsubseteq \ell_{\varepsilon_2}$,

and we aim to show that for all labels $\ell_{\text{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\text{Atk}}$ or $p \,;\, r \cong^{\varepsilon}_{\ell_{\text{Atk}}} q \,;\, r$.

Let $\rho = \text{map}_{\varepsilon_2}(\lfloor r \rfloor_{\varepsilon_1}) \,;\, \text{join}_{[\varepsilon_2,\varepsilon_1],\varepsilon}$. The rules in Figure 3 guarantee $P_{\varepsilon_2}(\!|t_1|\!) \vdash_{\text{P}} \rho \dashv P_\varepsilon(\!|t_2|\!)$ and CapturedSeq requires $\lfloor p \,;\, r \rfloor_\varepsilon \overset{(\!|t_3|\!),P_\varepsilon(\!|t_2|\!)}{=\!=} \lfloor p \rfloor_{\varepsilon_2} \,;\, \rho$, and similarly for q. All well-typed pure programs are noninterfering with respect to $\approx_{\_}$, ρ is pure, and $\ell \triangleleft P_{\varepsilon_2}(\!|t_1|\!)$ by ProtectC. Thus, for any label ℓ_{Atk}, either $\ell \sqsubseteq \ell_{\text{Atk}}$ or $\lfloor p \rfloor_{\varepsilon_2} \,;\, \rho \approx_{\ell_{\text{Atk}}} \lfloor q \rfloor_{\varepsilon_2} \,;\, \rho$. In the first case, we are finished. In the second case, the program equality above and EquivCap give us

$$\lfloor p \rfloor_{\varepsilon_2} \,;\, \rho \approx_{\ell_{\text{Atk}}} \lfloor q \rfloor_{\varepsilon_2} \,;\, \rho \iff \lfloor p \,;\, r \rfloor_\varepsilon \approx_{\ell_{\text{Atk}}} \lfloor q \,;\, r \rfloor_\varepsilon \implies p \,;\, r \cong^{\varepsilon}_{\ell_{\text{Atk}}} q \,;\, r. \qquad \square$$

Theorem 4 lifts noninterference of pure programs to effectful programs when the corresponding notions equivalence satisfy EquivCap. We can now see what happens if these equivalences do not match expectations. Recall our example from Section 5: we translate state by discarding writes and returning a constant for all reads. EquivCap no longer holds for contextual equivalence, but we can change the equivalence so that it does. There may be many such equivalences, but one simple option is to use the trivial effectful equivalence that is always true. With this equivalence, our example now admits all rules in Figure 3, so Theorem 4 applies. However, we are now giving trivial semantics to the type-and-effect system. Abstract noninterference with respect to this semantics merely says that an attacker who cannot distinguish anything cannot distinguish sensitive programs. This

$$\textsc{PcEff}\ \frac{t \diamond \mathrm{pc} \vdash_{\mathrm{Pc}} p \dashv t'}{\exists \varepsilon. [\![t]\!] \vdash_{\mathrm{E}} p \dashv [\![t']\!] \diamond \varepsilon} \qquad \textsc{ProtectTrans}\ \frac{\ell \triangleleft t}{\ell \triangleleft [\![t]\!]}$$

Fig. 4. Additional Rules for pc systems

result is both intuitively and technically trivial. The instantiation of the framework, while allowed, is therefore probably uninteresting.

6.2 Semantics of Program Counter Labels

We can now use the semantic framework we have developed for effectful labeled programs and noninterference for type-and-effect systems to talk about the semantics and security of the pc label. We extend the framework to include a pc system with judgments of the form $t \diamond \mathrm{pc} \vdash_{\mathrm{Pc}} p \dashv t'$. (We still use Roman letters for types and programs in the pc system, but we color them in green.) Figure 4 shows the rules we require for this extended framework.

We give semantics to the pc by formalizing the intuition that it constrains programs to only use secure effects. Specifically, we define the semantics by requiring a translation of typing proofs in the pc system to typing proofs in the type-and-effect system, which guarantees security by Theorem 4. PcEff formalizes this requirement.

For this semantics to make sense, we would like it to preserve types. Unfortunately, in our examples, the pc systems and type-and-effect systems had different function types. The pc system included a label on its functions ($\tau_1 \xrightarrow{\mathrm{pc}} \tau_2$), while the type-and-effect system included an effect ($\tau_1 \xrightarrow{\varepsilon} \tau_2$). We therefore allow the pc system to have different types, but the same programs, and require there to be a translation $[\![-]\!]$ from the pc types to the type-and-effect types. This translation must preserve the sensitivity of the data, represented as the protection level, a requirement we formalize as rule ProtectTrans.

These rules complete the requirements for our core theorem.

Theorem 5 (The Noninterference Half-Off Theorem)**.** *For any system satisfying all rules in Figures 3 and 4 where every well-typed pure program is noninterfering with respect to $\approx_{\_}$, then every program well-typed in the* pc *system is noninterfering with respect to $\cong^{\varepsilon}_{\_}$.*

Proof. This follows directly from PcEff and Theorem 4. We note that this always instantiates the ε in $\cong^{\varepsilon}_{\_}$ with the same ε used to type check r. □

7 DEEPENING THE PC-EFFECT CONNECTION

So far we have kept the connections between effects and program-counter labels lightweight: we only required a function $\ell_{\_}$ from effects to labels and the two rules from Figure 4. This means that our framework can give semantics to many systems. This generality, however, prevents us from proving some interesting theorems which we would like to prove. In this section, we strengthen the connection between program-counter labels and effects, allowing us to prove stronger results.

In particular, we formalize the aphorism that the pc is a lower bound on effects. Interestingly, not every language that fits our framework treats the pc as a lower bound on effects, despite the fact that they are secure by Thoerem 5. Indeed, a rather simple counterexample shows that the pc can always be incomparable to the label of an effect.

Still, all of our realistic examples do treat the pc as a lower bound on effects. We show that this is because they admit a few simple rules on top of the framework we have developed so far. Moreover, in all of our examples so far, we can extend the function $\ell_{\_}$ into a *Galois connection* between labels

and effects. Beyond being intrinsically interesting, it also allows us to refine our formalization of the folklore above, producing a more-concrete result.

7.1 Is the PC a Lower Bound on Effects?

We start by formalizing the folklore statement that "the pc is a lower bound on effects." As mentioned in Section 1, taken literally this aphorism does not even seem to type-check, since we are trying to bound an effect by a label. However, we can use our function $\ell\_$ to formalize the statement by saying that the pc bounds the *label* of the effect.

Definition 5 (pc-Bounded Effects)**.** We say effects are pc-*bounded* if whenever $t \diamond \mathsf{pc} \vdash_{\mathrm{PC}} p : t'$, there is some ε such that $[\![t]\!] \vdash_{\mathrm{E}} p \dashv [\![t']\!] \diamond \varepsilon$ where $\mathsf{pc} \sqsubseteq \ell_\varepsilon$.

Note that Definition 5 only requires that the pc bound *some* ε. We might at first think that any ε with which p type-checks in the type-and-effect system should be bounded below by pc, but effect variance prevents that definition from applying to most languages. To see why, imagine a program p in the system from Section 3 that reads some data from state, but can never write data or throw an exception. It can type-check with a pc of $\top$, which flows to ℓ_{R}, since $\ell_{\mathrm{R}} = \top$. However, by variance p can also type-check with effect $\{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$, and $\top \not\sqsubseteq \ell_{\{\mathrm{R},\mathrm{W},\mathrm{E}\}}$. The existential quantifier in Definition 5 thus provides a meaningful statement while allowing imprecision due to variance.

Common wisdom suggests that any language that uses a pc to enforce noninterference should have pc-bounded effects. However, this is not the case, as we can show using our framework.

Consider a language with state and exceptions, based on that from Section 3. In the original language, any preorder could serve as the set of information-flow labels (though we used a join semilattice for convenience). However, in the new language we will use a join semilattice of a special form. Intuitively, we want two equivalent but unrelated spaces of labels, one for effects and one for program-counter labels. Hence, we use a *semilattice coproduct*: given a semilattice of labels $\mathcal{L}$ sufficient to represent our effects, we construct a new semilattice with two disjoint copies of $\mathcal{L}$. We cannot work directly over the disjoint union $\mathcal{L} + \mathcal{L}$, since this is not a semilattice—there is no join of two labels $\mathsf{inl}(\ell_1)$ and $\mathsf{inr}(\ell_2)$. However, if we add a new distinguished top element, the result is a semilattice. In fact, it is the smallest semilattice that contains two disjoint copies of $\mathcal{L}$. Thus, we use a semilattice of this form for this example.

The modified language differs from the original in three ways: (*i*) the typing rule for label, (*ii*) the typing rules for effectful operations, and (*iii*) the function $\ell\_$. First, the label rule now forces all labels into the left-hand side of the lattice. That is, the rule is split into three cases:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{label}_{\mathsf{inl}(\ell)}\,(e) : L_{\mathsf{inl}(\ell)}\,(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{label}_{\mathsf{inr}(\ell)}\,(e) : L_{\mathsf{inl}(\ell)}\,(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{label}_{\top}\,(e) : L_{\mathsf{inl}(\top)}\,(\tau)}$$

Second, we also consider functions $\ell\_$ of a special form. Intuitively, the lattice of labels has a label space on the left for data, and a label space on the right for effects. We thus need ℓ_ε to always be of the form $\mathsf{inr}(\ell)$ for some $\ell \in \mathcal{L}$. To do this, we pick a function $\hat{\ell}\_ : \mathcal{E} \to \mathcal{L}$ connecting effects to the original semilattice $\mathcal{L}$, such as the effect-to-label function we used in Section 3. We then lift $\hat{\ell}\_$ to the full label space by defining $\ell_\varepsilon = \mathsf{inr}(\hat{\ell}_\varepsilon)$.

Finally, we modify the rules that compare the pc and effect labels by explicitly comparing the pc to the data-label analogue of the effect's label. Formally, we use the following rules:

$$\frac{\Gamma \diamond \mathsf{pc} \vdash e : \sigma \qquad \mathsf{pc} \sqsubseteq \mathsf{inl}(\hat{\ell}_{\mathrm{W}})}{\Gamma \diamond \mathsf{pc} \vdash \mathsf{write}(e) : \mathsf{unit}} \qquad \frac{\mathsf{pc} \sqsubseteq \mathsf{inl}(\hat{\ell}_{\mathrm{E}})}{\Gamma \diamond \mathsf{pc} \vdash \mathsf{throw} : \tau}$$

All of the other rules remain unchanged from those in Section 3.

$$\textsc{MapPC}\ \frac{t \diamond \mathrm{pc} \vdash_{\mathrm{PC}} p \dashv t'}{L_{\mathrm{pc}}(t) \diamond \mathrm{pc} \vdash_{\mathrm{PC}} \mathsf{map}_{\mathrm{pc}}(p) : L_{\mathrm{pc}}(t')} \qquad \textsc{MapEffInv}\ \frac{L_\ell(t) \vdash_{\mathrm{E}} \mathsf{map}_\ell(p) \dashv L_\ell(t') \diamond \varepsilon}{\exists \varepsilon'.\, t \vdash_{\mathrm{E}} p \dashv t' \diamond \varepsilon' \wedge \ell \sqsubseteq \ell_{\varepsilon'}}$$

$$\textsc{LabTrans}\ [\![L_\ell(t)]\!] = L_\ell([\![t]\!])$$

Fig. 5. Additional Rules for pc as a Lower Bound

This fits our framework and in fact admits exactly the same programs as the original pc-based system did. However, if a program type-checks with some pc, $\mathrm{pc} = \mathsf{inl}(\ell)$ for some ℓ, while the effect label will be $\ell_\varepsilon = \mathsf{inr}(\ell')$ for some ℓ'. By construction, we cannot have $\mathsf{inl}(\ell) \sqsubseteq \mathsf{inr}(\ell')$ for any labels ℓ and ℓ'.

This example shows that it is possible to have a secure language in our framework where the pc and the label of the effect are incomparable. The language is noninterfering, yet its effects are not pc-bounded. However, we only need a few simple additions to our framework to ensure that a language's effects are pc-bounded.

Consider a program p in one of our example type-and-effect systems such that $\Gamma, x : \tau \vdash p : \tau' \diamond \varepsilon$. We can transform this into a program on labeled data by unlabeling the input, running p, and labeling its output. That is, we can build a program transformer map_ℓ that we can type-check as $\Gamma, x : L_\ell(\tau) \vdash \mathsf{map}_\ell(p) : L_\ell(\tau') \diamond \varepsilon$. Notably, this does not change the effect. To retain security, we must ensure that $\ell \sqsubseteq \ell_\varepsilon$, since p may otherwise leak data about the ℓ-labeled input in its effects.

When a program of the form $\mathsf{map}_\ell(p)$ has effect ε—that is, $\Gamma, x : L_\ell(\tau) \vdash \mathsf{map}_\ell(p) : L_\ell(\tau') \diamond \varepsilon$—we know that the effect ε must come from p. Moreover, we know that p must have type-checked with some effect ε' such that $\ell \sqsubseteq \ell_{\varepsilon'}$. However, this need not be ε, due to similar reasoning about variance that we saw in the design of Definition 5. Again, this leads to an existential quantifier.

The program transformer $\mathsf{map}_\ell(p)$ also has a similar action in the pc system as it did in the type-and-effect system. If $\Gamma, x : \tau \diamond \mathrm{pc} \vdash p : \tau'$, then we want $\Gamma, x : L_\ell(\tau) \diamond \mathrm{pc} \vdash \mathsf{map}_\ell(p) : L_\ell(\tau')$. However, now the limiter is the pc rather than the effect. That is, this only type-checks if $\ell \sqsubseteq \mathrm{pc}$. We also note that map_ℓ type-checks in both the pc and type-and-effect systems because the type translation between them leaves labels alone: $[\![L_\ell(\tau)]\!] = L_\ell([\![\tau]\!])$.

We provide version of these rules in Figure 5 using the single-input, single-output judgments from Sections 5 and 6. These versions are, in fact, slightly more general. First, the MapPC rule only requires that we be able to map the current pc, rather than any label ℓ where $\ell \sqsubseteq \mathrm{pc}$. Second, we have no requirement that all effects of $\mathsf{map}_\ell(p)$ come from p, only that the label of the effects that do come from p are bounded below by ℓ.

Note also that we do not have a rule corresponding to map_ℓ in the type-and-effect system. PcEff will ensure that map_ℓ has the right type, which is all we need for the applications in this paper. However the rule MapEffInv is very suggestive, and most any language that admits MapEffInv will also admit an appropriate rule for map_ℓ for effects.

Added to our framework, these extra assumptions are sufficient to prove that a language is pc-bounded.

Theorem 6. *The effects of any language admitting the rules in Figures 3, 4, and 5 are* pc*-bounded.*

Proof.

$$\dfrac{\dfrac{\dfrac{t \diamond \mathrm{pc} \vdash_{\mathrm{PC}} p \dashv t'}{L_{\mathrm{pc}}(t) \diamond \mathrm{pc} \vdash_{\mathrm{PC}} \mathrm{map}_{\mathrm{pc}}(p) \dashv L_{\mathrm{pc}}(t')}\ \textsc{MapPC}}{\exists \varepsilon.\, L_{\mathrm{pc}}([\![t]\!]) \vdash_{\mathrm{E}} \mathrm{map}_{\mathrm{pc}}(p) \dashv L_{\mathrm{pc}}([\![t']\!]) \diamond \varepsilon}\ \textsc{PcEff and LabTrans}}{\exists \varepsilon'.\, [\![t]\!] \vdash_{\mathrm{E}} p \dashv [\![t']\!] \diamond \varepsilon' \wedge \mathrm{pc} \sqsubseteq \ell_{\varepsilon'}}\ \textsc{MapEffInv}$$

□

7.2 Computing PC Bounds via Galois Connections

Our example languages have even more structure: we know what effect a program has based on its pc. That is, we can build a function γ from labels to effects such that if a program type-checks with program-counter label pc then it type-checks with effect $\gamma(\mathrm{pc})$. Let us examine this in detail for the example language from Section 3. We used a function $\ell\_$ on effects such that $\ell_{\{E\}} = \ell_{\mathsf{Exn}}$, $\ell_{\{R\}} = \top$, and $\ell_{\{W\}} = \ell_{\mathsf{State}}$ and on arbitrary sets it acts as a lower bound. Note that because of this $\ell\_$ must be *antitone*: if $\varepsilon \subseteq \varepsilon'$, then $\ell_{\varepsilon'} \sqsubseteq \ell_{\varepsilon}$.

Given a particular $\ell\_$, we can then define the function γ from labels to effects as follows:

$$\gamma(\ell) = \{R\} \cup \{W \mid \ell \sqsubseteq \ell_{\mathsf{State}}\} \cup \{E \mid \ell \sqsubseteq \ell_{\mathsf{Exn}}\}$$

The functions $\ell\_$ and γ form a *Galois connection*. Galois connections are well-known for their uses in abstract interpretation [Cousot and Cousot 1977]. However, we will see that they can be used here to strengthen Theorem 6 by providing a witness for the existential in Definition 5.

Since ℓ is antitone, it seems like it cannot be part of a Galois connection, as Galois connections are defined on *monotone* functions. However, it turns out $\ell\_$ and γ more precisely form an *antitone* Galois connection [see e.g., Galatos 2007]. An antitone Galois connection between lattices A and B is equivalent to a monotone Galois connection between A and B^{op}, the order dual of B.

Lemma 3 (Antitone Galois Connection). *The functions $\ell\_$ and γ form an antitone Galois connection. That is, for any label ℓ and effect ε, $\ell \sqsubseteq \ell_{\varepsilon}$ if and only if $\varepsilon \subseteq \gamma(\ell)$.*

Proof. By examining all of the (eight) possible values of ε. □

A similar construction and proof can be done for the example of Section 4, and for realistic languages like Jif [Magrino et al. 2016], Fabric [Liu et al. 2017], and FlowCaml [Pottier and Simonet 2002].

With this structure, it becomes relatively easy to strengthen Theorem 6. Here we use $[\varepsilon] \preceq \varepsilon'$ as the general ordering relation on effects

Theorem 7. *If a language admits the rules in Figures 3, 4, and 5, and there is a function γ such that $\ell\_$ and γ form an antitone Galois connection, then whenever $t \diamond \mathrm{pc} \vdash_{PC} p \dashv t'$, then $[\![t]\!] \vdash_{E} p \dashv [\![t']\!] \diamond \gamma(\mathrm{pc})$.*

Proof. By Theorem 6, we know that there is some ε such that $[\![t]\!] \vdash_{\mathrm{E}} p \dashv [\![t']\!] \diamond \varepsilon$ and $\mathrm{pc} \sqsubseteq \ell_{\varepsilon}$. Because $\mathrm{pc} \sqsubseteq \ell_{\varepsilon}$ and $\ell\_$ and γ form an antitone Galois connection, $[\varepsilon] \preceq \gamma(\mathrm{pc})$. This means we can further apply $\textsc{Seq}_{\preceq}$ to get $[\![t]\!] \vdash_{\mathrm{E}} p \dashv [\![t']\!] \diamond \gamma(\mathrm{pc})$. □

8 RELATED WORK

This work pulls mostly from two distinct areas: static IFC and the theory of effects. We discuss related work from each of these areas in turn.

8.1 Static Information-Flow Control

Noninterference, originally introduced by Goguen and Meseguer [1982], is the foundational security property of information-flow control systems. While originally proposed to avoid confidentiality leaks, noninterference can apply to any security policy expressible as a preorder of labels. Since Volpano et al.'s [1996] seminal work enforcing noninterference with a type system, numerous others

have used type systems to guarantee noninterference for functional and imperative languages, with and without effects, where security policies represent confidentiality, integrity, or even distributed consistency [Abadi et al. 1999; Heintze and Riecke 1998; Milano and Myers 2018; Pottier and Simonet 2002; Rafnsson and Sabelfeld 2014; Sabelfeld and Myers 2003; Tsai et al. 2007; Vassena et al. 2018; Zdancewic and Myers 2002].

Termination is one channel many type-based enforcement mechanisms ignore [e.g., Liu et al. 2017; Magrino et al. 2016; Pottier and Simonet 2002; Volpano et al. 1996]. As Volpano and Smith [1997] showed and we observed in Section 4, enforcing termination-sensitive noninterference with a type system is possible, but highly restrictive. Unfortunately, Askarov et al. [2008] argue that termination channels can leak an arbitrary amount of data, making it dangerous to ignore them. We hope that our framework's ability to unify possible nontermination with other effects will connect to recent work on precisely specifying and constraining leakage through termination in more permissive languages [Bay and Askarov 2020; Moore et al. 2012].

Prior work uses a wide variety of techniques to prove noninterference. The first proofs of static noninterference [Heintze and Riecke 1998; Volpano and Smith 1997; Volpano et al. 1996] relied on structural induction with careful manual reasoning. Pottier and Simonet [2002] used bracketed pairs of terms to simulate two program executions with different high inputs and compare the outputs. This technique makes combining state and exceptions tractable, but provides no means to reason about termination. Other proofs rely on semantics using partial equivalence relations [Abadi et al. 1999; Sabelfeld and Sands 2001; Tse and Zdancewic 2004] or logical relations [Rajani and Garg 2018; Shikuma and Igarashi 2008]. The complexity of all of these approaches lies in reasoning about effects, demonstrating the value of noninterference half-off.

We base all of the example languages in this paper on DCC [Abadi et al. 1999]. DCC was originally designed to explore dependency, with information flow as an interesting special case. Interestingly, DCC was not given an operational semantics or a noninterference theorem in the original paper. Instead, Abadi et al. [1999] described a domain-theoretic semantics, and used it to prove a semantic security theorem closely related to noninterference. Tse and Zdancewic [2004] later developed an operational semantics for DCC, and claimed to prove a noninterference theorem analogous to the one we used in Section 2 by translating DCC into System F and using parametric reasoning. Shikuma and Igarashi [2008], however, found a flaw in Tse and Zdancewic's proof, which Bowman and Ahmed [2015] later repaired. Algehed and Bernardy [2019] extended and simplified the proof technique, leading to a verified version of the proof written in Agda.

DCC is the paradigmatic *coarse-grained* IFC language, a style that is characterized by labeling and unlabeling data. The various other results mentioned above all employ *fine-grained* IFC systems, where each type includes a label. Though the two approaches may appear substantially different, Rajani and Garg [2018] proved them equivalent.

Both DCC_{pc} [Tse and Zdancewic 2004] and the Sealing Calculus [Shikuma and Igarashi 2008] include *protection context labels* that look very similar to our pc labels. Both languages, however, are pure, and the labels serve only to securely include a more permissive typing rule for unlabel. While our examples could also employ this technique, it would increase the complexity of the type systems, particularly the type-and-effect systems which would need to include both protection context labels and effects.

Other work has implemented coarse-grained IFC as monadic libraries, mostly in Haskell [Algehed and Russo 2017; Arden 2017; Russo et al. 2008; Stefan et al. 2011; Tsai et al. 2007; Vassena et al. 2018]. Both Algehed and Russo [2017] and MAC [Vassena et al. 2018], moreover, handle effectful computation via monadic reasoning. Algehed and Russo [2017] in particular advocate building noninterfering pure languages, and using monads to define effects on top of them. They do not,

however, explore the connections to pc systems. MAC [Vassena et al. 2018] combines the monads for effects and the monad for labels, and therefore still requires a pc label.

8.2 The Theory of Effects

Type-and-effect systems originated as a program analysis technique [Lucassen and Gifford 1988; Nielson 1996; Nielson and Nielson 1999]. This technique allowed compilers to leverage the type system of their source language to track other properties of programs, enabling optimizations like dead-code elimination that may behave differently depending on effects.

Wadler and Thiemann [1998] gave type-and-effect systems semantics via monads by recognizing the correspondence between type-and-effect systems and Moggi's [1989; 1991] notions of computation. This result gave rise to a long line of work describing generalizations of monads which could be used to give semantics to as many type-and-effect systems as possible. The most relevant for this work are Wadler and Thiemann's (and Orchard et al.'s [2014]) indexed monads, which work on a lattice of effects [Wadler and Thiemann 1998] and Tate's [2013] productors, which work on an arbitrary effector.

9 CONCLUSION

We have developed a framework that gives semantics to program-counter labels based on the semantics of producer effects. This choice supports an abstract perspective, allowing us to reason about a language *feature* without being tied to a specific language. The Noninterference Half-Off Theorem (Theorem 5) thus proved noninterference for a large swath of languages—any language admitting the simple rules in Figures 3 and 4. Moreover, the proof technique the theorem suggests provides simple proofs of noninterference for important effects: state, exceptions, and nontermination. It even applies to languages with multiple types of effects, as we saw in Section 3.

By viewing possible nontermination as an effect, we both achieved a half-off proof of termination-sensitive noninterference and unified the treatment of termination sensitivity with that of other effects. Previously, these had only been considered separately. Hopefully, this new understanding of termination sensitivity will allow us to build better termination-sensitive type systems.

We also demonstrated the power of our framework by using it to formalize the folklore belief that the pc is a lower bound on the effects in secure programs. Surprisingly, such pc-boundedness is *not* a theorem of our semantics. It is, however, a theorem of a slightly expanded version of our semantics. Moreover, this extension is suggestive of a categorical construct called a *distributive law*. Exploring this connection would be interesting future work.

In fact, a categorical perspective infuses this entire work. The semantics of effects are usually given categorically, so perhaps this is unsurprising. Further formalizing this work categorically would require a categorical models of noninterference, perhaps developing a connection between our semantics and Kavvos's [2019] semantics of pure noninterference given via modal types.

We believe there are two other important directions for future work. First, our framework could influence the design of, and facilitate noninterference proofs for, a language with secure algebraic effect handlers. Algebraic effect handlers [Bauer and Pretnar 2015; Leijen 2016; Plotkin and Power 2003; Plotkin and Pretnar 2009; Pretnar 2010] allow programmers to specify their own effectful operations while retaining the fundamental properties of a pure language, hopefully making secure programming considerably easier. Second, one may be able to expand our framework to other security guarantees by replacing preservation of equivalences in Figure 3 with preservation of arbitrary *hyperproperties* [Clarkson and Schneider 2010]. Such an extension might have important applications in the design of secure programming languages in general.

The results we have developed in this paper and the future work we suggest all require the generality of our semantic framework. We hope that future work also adopts abstract perspectives to similarly prove highly general results.

ACKNOWLEDGMENTS

This work originated from ideas we had while designing the First Order Logic for Flow-Limited Authorization along with Pedro de Amorim, Owen Arden, and Ross Tate. Deepak Garg and Andrew C. Myers helped us focus the work and, along with Deian Stefan, provided guidance on how to explain our results to a broader audience. Maximilian Algehed pointed us to important related work. Coşku Acay, Jonathan DiLorenzo, Matthew Milano, and Isaac Sheff helped with editing. Finally, our shepherd, William Bowman, and our anonymous reviewers provided tremendously insightful comments and helpful suggestions.

This project was supported in part by a fellowship awarded through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR), and the Army Research Office (ARO). Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and may not reflect those of these sponsors.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. 1999. A Core Calculus of Dependency. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/292540.292555

Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple Noninterference from Parametricity. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3341693

Maximilian Algehed and Alejandro Russo. 2017. Encoding DCC in Haskell. In *Programming Languages and Analysis for Security (PLAS)*. https://doi.org/10.1145/3139337.3139338

Owen Arden. 2017. *Flow-Limited Authorization*. Ph.D. Dissertation. Cornell University. https://users.soe.ucsc.edu/~owen/publications/pdfs/FLA_OwenArden.pdf

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 333–348. https://doi.org/10.1007/978-3-540-88313-5_22

Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming (JLAMP)* 84, 1 (2015). https://doi.org/10.1016/j.jlamp.2014.02.001

Johan Bay and Aslan Askarov. 2020. Reconciling Progress-Insensitive Noninterference and Declassification. In *Computer Security Foundations (CSF)*. 95–106. https://doi.org/10.1109/CSF49147.2020.00015

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

Michael Clarkson and Fred Schneider. 2010. Hyperproperties. *Journal of Computer Security (JCS)* 18, 6 (2010). https://doi.org/10.3233/JCS-2009-0393

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. https://doi.org/10.1145/512950.512973

Soichiro Fujii, Shin-ya Katsumata, and Paul-André Mellisès. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computational Structures (FOSSACS)*. https://doi.org/10.1007/978-3-662-49630-5_30

Nikolaos Galatos. 2007. *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*. Elsevier Sience.

Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *Symposium on Security and Privacy (SSP) (Oakland)*. https://doi.org/10.1109/SP.1982.10014

Nevin Heintze and John G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/268946.268976

Alan Jeffrey. 1997. Premonoidal Categoies and a Graphical View of Programs. http://fpl.cs.depaul.edu/ajeffrey/premon/paper.html

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2535838.2535846

G. A. Kavvos. 2019. Modalities, Cohesion, and Information Flow. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3290333

Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *International Cryptology Conference (CRYPTO)*. Springer, 104–113.

Daan Leijen. 2016. *Type Directed Compilation of Row-Typed Algebraic Effects*. Technical Report. Microsoft. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/08/algeff-tr-2016-1.pdf

Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. 2017. Fabric: Building Open Distributed Systems Securely by Construction. *Journal of Computer Security (JCS)* 25 (2017). https://doi.org/10.323/JCS-15805

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/73560.73564

Tom Magrino, Jed Liu, Owen Arden, Chinawat Isradisaikul, and Andrew C. Myers. 2016. Jif 3.5: Java Information Flow. (June 2016). https://www.cs.cornell.edu/jif Software release.

Daniel Marino and Todd Milstein. 2009. A Generic Type-and-Effect System. In *Types in Language Design and Implementation (TLDI)*. https://doi.org/10.1145/1481861.1481868

Matthew P. Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Programming Languages Design and Implementation (PLDI)*. https://doi.org/10.1145/3192366.3192375

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Logic in Computer Science (LICS)*. https://doi.org/10.1109/LICS.1989.39155

Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (1991). https://doi.org/10.1016/0890-5401(91)90052-4

Scott Moore, Aslan Askarov, and Stephen Chong. 2012. Precise Enforcement of Progress-Sensitive Security. In *Computer Security Foundations (CSF)*. 881–893. https://doi.org/10.1145/2382196.2382289

Andrew C. Myers. 1999. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages (POPL)*. 228–241.

Flemming Nielson. 1996. Annotated Type and Effect Systems. *ACM Computing Surveys (CSUR)* 28, 2 (1996). https://doi.org/10.1145/234528.234745

Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*. Springer. https://doi.org/10.1007/3-540-48092-7_6

Dominic Orchard, Tomas Petricek, and Alan Mycroft. 2014. The Semantic Marriage of Effects and Monads. (2014). https://arxiv.org/abs/1401.5391

Benjamin C Pierce. 2002. *Types and Programming Languages*. MIT press.

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003). https://doi.org/10.1023/A:1023064908962

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-00590-9_7

François Pottier and Vincent Simonet. 2002. Information Flow Inference for ML. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/503272.503302

Matija Pretnar. 2010. *The Logic and Handling of Algebraic Effects*. Ph.D. Dissertation. School of Informatics, The University of Edinburgh. http://hdl.handle.net/1842/4611

Willard Rafnsson and Andrei Sabelfeld. 2014. Compositional Information-Flow Security for Interactive Systems. In *Computer Security Foundations (CSF)*. https://doi.org/10.1109/CSF.2013.8

Vineet Rajani and Deepak Garg. 2018. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *Computer Security Foundations (CSF)*. https://doi.org/10.1109/CSF.2018.00024

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-Weight Information-Flow Security in Haskell. In *Haskell Symposium (HASKELL)*. https://doi.org/10.1145/1411286.1411289

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications (JSAC)* 21, 1 (2003). https://doi.org/10.1109/JSAC.2002.806121

Andrei Sabelfeld and David Sands. 2001. A PER Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation* 14, 1 (2001). https://doi.org/10.1023/A:1011553200337

Naokata Shikuma and Atsushi Igarashi. 2008. Proving Noninterference by a Fully Complete Translation to the Simply Typed λ-calculus. *Logical Methods in Computer Science (LMCS)* 4, 3 (September 2008). https://doi.org/10.2168/LMCS-4(3:10)2008

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Haskell Symposium (HASKELL)*. https://doi.org/10.1145/2034675.2034688

Ross Tate. 2013. The Sequential Semantics of Producer Effect Systems. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2429069.2429074

Tsa-ching Tsai, Alejandro Russo, and John Hughes. 2007. A Library for Secure Multi-Threaded Information Flow in Haskell. In *Computer Security Foundations (CSF)*. https://doi.org/10.1109/CSF.2007.6

Stephen Tse and Steve Zdancewic. 2004. Translating Dependency into Parametricity. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1016850.1016868

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. MAC: A Verified Static Information-Flow Control Library. *Journal of Logical and Algebraic Methods in Programming (JLAMP)* 95 (2018). https://doi.org/10.1016/j.jlamp.2017.12.003

Dennis Volpano and Geoffrey Smith. 1997. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop (CSFW)*. IEEE.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security (JCS)* 4, 3 (1996). https://doi.org/10.3233/JCS-1996-42-304

Philip Wadler and Peter Thiemann. 1998. The Marriage of Effects and Monads. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/289423.289429

Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. 2015. It's My Privilege: Controlling Downgrading in DC-Labels. In *Security and Trust Management (STM)*. https://doi.org/10.1007/978-3-319-24858-5_13

Steve Zdancewic and Andrew C Myers. 2002. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation* 15, 2-3 (2002).

APPENDICES

A PROGRAMS WITH MULTIPLE INPUTS

In Section 5, we developed a theory of effects in languages with information-flow-control types. This was designed to be extremely general. However, we developed our theory for single-input, single-output programs. In this appendix, we consider expanding our theory to multiple-input programs, such as the simply-typed λ-calculus.

Tate [2013, see Section 9], mentions that a productor can be viewed as a 2-functor from an effector to the category of categories. Thus, each effect ε is mapped to a functor $\langle P_\varepsilon(-), \mathsf{map}_\varepsilon \rangle$, and each inequality $[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon$ is mapped to a natural transformation $P_{\varepsilon_1}(-) \circ \cdots \circ P_{\varepsilon_n}(-) \Rightarrow P_\varepsilon(-)$. He then suggests that we can move to multiple-input languages by defining changing the base to premonoidal categories, defining a *strong* productor.

To understand this, let us define a premonoidal category C [Jeffrey 1997]:

Definition 6 (Premonoidal Category)**.** A *premonoidal category* is a category C along with

- A binary operation on objects, written $- \otimes -$
- For every object Γ, two functors $\Gamma \ltimes -$ and $- \rtimes \Gamma$, such that their action on objects is $- \otimes -$

This is enough to define a notion of propagating context. We have suggestively written objects of our category as Γ, and we can define Γ_1, Γ_2 as $\Gamma_1 \otimes \Gamma_2$. Then, for any morphism $\rho : \Gamma_1 \longrightarrow \Gamma_2$, we can think of ρ as a program that operates in an environment Γ_1, and then finishes having changed the environment to Γ_2. Then, $\Gamma \ltimes \rho : \Gamma \otimes \Gamma_1 \longrightarrow \Gamma \otimes \Gamma_2$, so we have propagated the context Γ.

We can put this in perhaps-more-familiar programming-languages terms. A premonoidal category is one where the following rules are admissible:

$$\frac{\Gamma_1 \vdash \rho \dashv \Gamma_2}{\Gamma_1, \Gamma \vdash \rho \rtimes \Gamma \dashv \Gamma_2, \Gamma} \qquad \frac{\Gamma_1 \vdash \rho \dashv \Gamma_2}{\Gamma, \Gamma_1 \vdash \Gamma \ltimes \rho \dashv \Gamma, \Gamma_2}$$

Here, we write $\Gamma_1 \vdash \rho \dashv \Gamma_2$ as the typing judgment representing a program $\rho : \Gamma_1 \longrightarrow \Gamma_2$.

Then, a strong productor is simply a productor for which every functor $P_\varepsilon(-)$ is a premonoidal functor, that is $P_\varepsilon(\Gamma_1 \otimes \Gamma_2) = P_\varepsilon(\Gamma_1) \otimes P_\varepsilon(\Gamma_2)$. In programming-language terms, the following rules

are admissible:

$$\frac{\Gamma_1 \vdash \rho \dashv \Gamma_2}{\mathsf{map}_\varepsilon(\rho \rtimes \Gamma) \underset{(P_\varepsilon(\Gamma_1),P_\varepsilon(\Gamma)),\,(P_\varepsilon(\Gamma_2),P_\varepsilon(\Gamma))}{=\!=\!=} \mathsf{map}_\varepsilon(\rho) \rtimes P_\varepsilon(\Gamma)} \qquad \frac{\Gamma_1 \vdash \rho \dashv \Gamma_2}{\mathsf{map}_\varepsilon(\Gamma \ltimes \rho) \underset{(P_\varepsilon(\Gamma),P_\varepsilon(\Gamma_1)),\,(P_\varepsilon(\Gamma),P_\varepsilon(\Gamma_2))}{=\!=\!=} P_\varepsilon(\Gamma) \ltimes \mathsf{map}_\varepsilon(\rho)}$$

For a linear language, this is enough. But we often want to deal with non-linear languages with the following notion of sequencing:

$$\frac{\Gamma \vdash p_1 : t_1 \diamond \varepsilon_1 \qquad \Gamma, x_1 : t_1 \vdash p_2 : t_2 \diamond \varepsilon_2 \qquad \cdots \qquad \Gamma, x_1 : t_1, \ldots, x_{n-1} : t_{n-1} \vdash p_n : t_n \diamond \varepsilon_n \qquad [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{\Gamma \vdash \mathsf{let}\ x_1 = p_1\ \mathsf{in}\ \mathsf{let}\ x_2 = p_2\ \mathsf{in}\ \cdots \mathsf{let}\ x_{n-1} = p_{n-1}\ \mathsf{in}\ p_n : t_n \diamond \varepsilon}$$

In order to give semantics to this rule, we need an extra few assumptions. In particular, we need a doubling natural transformation $\Delta_\Gamma : \Gamma \to \Gamma \otimes \Gamma$, which is preserved by map_ε. That is, we need the following rules to be admissible:

$$\frac{}{\Gamma \vdash \Delta_\Gamma \dashv \Gamma, \Gamma} \qquad \frac{}{\mathsf{map}_\varepsilon(\Delta_\Gamma) \underset{P_\varepsilon(\Gamma),\,(P_\varepsilon(\Gamma),P_\varepsilon(\Gamma))}{=\!=\!=} \Delta_{P_\varepsilon(\Gamma)}}$$

This allows us to prove the following:

Theorem 8 (Semantics of Effectful Composition with Let)**.**

$$\frac{\Gamma \vdash p_1 : t_1 \diamond \varepsilon_1 \qquad \cdots \qquad \Gamma, x_1 : t_1, \ldots, x_{n-1} : t_{n-1} \vdash p_n : t_n \diamond \varepsilon_n \qquad [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{\left\lfloor \begin{array}{l} \mathsf{let}\ x_1 = p_1 \\ \mathsf{in}\ \ldots \mathsf{let}\ x_{n-1} = p_{n-1} \\ \qquad \mathsf{in}\ p_n \end{array} \right\rfloor_\varepsilon \underset{\Gamma,\,(\Gamma, x_1 : t_1, \ldots, x_n : t_n)}{=\!=\!=} \begin{array}{l} \Delta_\Gamma\ ;\ \Gamma \ltimes \lfloor p_1 \rfloor_{\varepsilon_1}\ ;\ \mathsf{join}_{[],\varepsilon_1} \rtimes P_\varepsilon(x_1 : t_1) \\ \qquad ;\ \mathsf{map}_{\varepsilon_1}(\Delta_{\Gamma, x_1 : t_1}\ ;\ \cdots \\ \qquad\qquad ;\ \mathsf{map}_{\varepsilon_n}(\lfloor p_n \rfloor_{\varepsilon_n})) \\ \qquad ;\ \mathsf{join}_{[\varepsilon_1, \ldots \varepsilon_n],\varepsilon} \end{array}}$$

In order to extend to information-flow-control typed languages, we need only assume that labeling is a *strong premonoidal* functor. That is, the following rules are admissible:

$$L_\ell(\Gamma_1, \Gamma_2) = L_\ell(\Gamma_1), L_\ell(\Gamma_2) \qquad \frac{}{\Gamma_1, L_\ell(\Gamma_2) \vdash \mathsf{str}_{\ell,\Gamma_1,\Gamma_2} \dashv L_\ell(\Gamma_1, \Gamma_2)}$$

Then, extending Theorem 8 to unlabel is not difficult.