Giving Semantics to Program-Counter Labels via Secure Effects

Andrew K. Hirsch
MPI-SWS
akhirsch@mpi-sws.org

Ethan Cecchetti
Cornell University
ethan@cs.cornell.edu

Abstract

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.

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:

\[
\text{if } x = 1 \text{ then write}(1) \text{ else write}(0)
\]

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 leaks nothing. For instance, using a type system with a program-counter label we 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 the type system. 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 $producer$ [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, noninterference (of the effects) is almost trivial, as expected. However, for a language to fit our framework its 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, and 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. It will form the basis of our examples in this paper. 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: \( \text{label}_\ell(e) \) and \( \text{unlabel} \ e \) as \( x \) in \( e \). 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.)

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 \( \ell_1 \) and \( \ell_2 \) 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 \( \ell_2 \) is at least as restrictive as \( \ell_1 \), 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 \( \text{label}_\ell(e) \) represents protecting the output of \( e \) at label \( \ell \). That is, \( e \) should only be used to compute information at levels at least as high as \( \ell \). Such computations are possible using the term unlabel \( e_1 \) as \( x \) 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 \( \tau \) being “of high enough level” to use information at label \( \ell \) is expressed in a relation \( \ell \triangleleft \tau \), which is read as “\( \ell \) is protected by \( \tau \)” or “\( \tau \) protects \( \ell \).” The formal rules defining this relation are in Figure 2. Intuitively, if \( \ell \triangleleft \tau \), then \( \tau \) information is at least as secret as \( \ell \).

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

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{label}_\ell(e) : L_\ell(\tau)} \quad \frac{\Gamma \vdash e_1 : L_\ell(\tau_1) \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \ell \triangleleft \tau_2}{\Gamma \vdash \text{unlabel} e_1 \text{ as } x \text{ in } e_2 : \tau_2}
\]

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 \( \lambda \)-calculus, so we only discuss the semantics of the terms \( \text{label}_\ell(e) \) and unlabel \( e_1 \) as \( \ell \) in \( e_2 \). We first note that \( \text{label}_\ell(E) \) and unlabel \( E \) as \( \ell \) in \( e_2 \) are evaluation contexts, where \( E \) stands for an arbitrary evaluation context. That is, computation can take place under both \( \text{label}_\ell(-) \) and unlabel \(-\) as \( x \) 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:

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

The full definition of the type system for, and operational semantics of, DCC can be found in Appendix B.

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 \( \ell \), representing what an attacker who can see values only up to label \( \ell \) can distinguish. The definition is contextual to allow for comparison of first-class functions.

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

Intuitively, expressions are \( \ell \)-equivalent if no well-typed decision procedure with output labeled \( \ell \) 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 machined-checked proof in Agda.

**Theorem 1** (Noninterference for DCC [Algehed and Bernardy, 2019; Bowman and Ahmed, 2015]). For expressions e₁ and e₂ and ℓ ∈ L, if Γ ⊢ e₁ : τ, Γ ⊢ e₂ : τ, and ℓ ≺ τ, then for all labels ℓ_{Atk} ∈ L, either ℓ ⊑ ℓ_{Atk} or e₁ ≈_{ℓ_{Atk}} e₂.

### 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:

- **Expressions**
  - e ::= ... | read | write(e) | throw | try {e₁} catch {e₂}
- **Evaluation Contexts**
  - E ::= ... | write(E) | try {E} catch {e₂}
- **Throw Contexts**
  - T ::= [] | T e | v T | (T, e) | (v, T) | proj₁(T) | proj₂(T)
  - | inl(T) | inr(T) | (match T with τ | inl(x) ⇒ e₁ | inr(y) ⇒ e₂ end)
  - | label_e(T) | unlabel T as x in e | write(T)

Here read returns the value of type σ currently stored in the one state cell, while write(e) replaces that value with e and returns unit. We include 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, try {e₁} catch {e₂} runs e₁ 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 e₂. We include try {E} catch {e₂} as an evaluation context, so it will evaluate e₁ 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:

- Γ ⊢ read : σ
- Γ ⊢ e : σ
- Γ ⊢ write(e) : unit
- Γ ⊢ throw : τ
- Γ ⊢ try {e₁} catch {e₂} : τ

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:

- ⟨read, s⟩ → ⟨s, s⟩
- ⟨write(v), s⟩ → ⟨(), v⟩
- ⟨T[throw], s⟩ → ⟨throw, s⟩

- ⟨try {v} catch {e}, s⟩ → ⟨v, s⟩
- ⟨try {throw} catch {e}, s⟩ → ⟨e, s⟩

¹Allowing higher-order state is possible, but it requires recursive types and complicates reasoning about termination.
We modify the rest of the operational semantics by including $s$ without modification in every other rule, as is standard for simply-typed $\lambda$-calculus with state [Pierce, 2002, Chapter 13.3].

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:

\[
\text{unlabel } h \text{ as } x \\
\text{in match } x \text{ with} \\
\begin{align*}
&h : L_\ell(\text{unit + unit}) \vdash \text{inl(\_)} \Rightarrow \text{throw} : L_\ell(\text{unit}) \\
&\quad \text{inr(\_)} \Rightarrow \text{label}_\ell(\_)
\end{align*}
\text{end}
\]

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 $pc \in \mathcal{L}$ with the typing judgment to track the sensitivity of the context. Thus, the typing judgment of this new type system takes the form $\Gamma \diamond pc \vdash e : \tau$ 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:

\[
\frac{\Gamma \diamond pc \vdash e_1 : L_\ell(\tau_1) \quad \Gamma, x : \tau_1 \diamond pc \sqcup \ell \vdash e_2 : \tau_2 \quad \ell \preceq \tau_2}{\Gamma \diamond pc \vdash \text{unlabel } e_1 \text{ as } x \text{ in } e_2 : \tau_2}
\]

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 $\lambda$-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 $\tau_1 \xrightarrow{pc} \tau_2$ is a function that takes an argument of type $\tau_1$, returns a value of type $\tau_2$, 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:

\[
\frac{\Gamma, x : \tau_1 \diamond pc_1 \vdash e : \tau_2}{\Gamma \diamond pc_2 \vdash \lambda x : \tau_1. e : \tau_1 \xrightarrow{pc_1} \tau_2}
\quad
\frac{\Gamma \diamond pc_1 \vdash e_1 : \tau_1 \xrightarrow{pc_2} \tau_2 \quad \Gamma \diamond pc_1 \vdash e_2 : \tau_1 \quad pc_1 \sqsubseteq pc_2}{\Gamma \diamond pc_1 \vdash e_1 e_2 : \tau_2}
\]
This change also necessitates adjusting the function protection rule from Figure 2. Applying a function with type $\tau_1 \xrightarrow{pc} \tau_2$ can still reveal data through its output at the level of $\tau_2$, 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:

$$\ell << \tau_2 \quad \ell \sqsubseteq pc$$

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 $\ell_{State}$ can see the value stored in the state cell. An attacker who can read $\ell_{State}$ can see writes to the 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 $\ell_{Exn}$ can distinguish between exceptions and other values. An attacker who cannot see information labeled $\ell_{Exn}$ 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 $\ell_{State}$.

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.

- **Read Rule:**
  $$\Gamma \vdash pc \vdash read : \sigma$$
  $$\Gamma \vdash pc \vdash write(e) : unit$$
  $$\Gamma \vdash pc \vdash throw : \tau$$

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: $\ell_{Exn}$.

$$\Gamma \vdash pc \vdash try\{e_1\} catch\{e_2\} : \tau$$

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:

Let $\_ = \text{unlabel}\ h$ as $x$

$$h : L_{\ell_{Exn}} (\text{unit} + \text{unit}), s : \sigma \vdash$$

- $\text{inl}(\_)$ $\Rightarrow$ $\text{throw}$
  $$\text{inr}(\_) \Rightarrow \text{label}_{\ell_{Exn}}() : L_{\ell_{Exn}} (\text{unit})$$

end

in $\text{write}(s)$

If $h = \text{label}_{\ell_{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 $\ell_{State}$. 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 $\ell_{Exn} \sqsubseteq \ell_{State}$.

Proving that we have successfully eliminated data leaks requires using a notion of $\ell$-equivalence that accounts for exceptions and state. Notably, to properly capture which attackers may view which values, our notion differs depending on how $\ell$ relates to $\ell_{State}$ and $\ell_{Exn}$.
Definition 2. We say programs $e_1$ and $e_2$ are state and exception $\ell$-equivalent, denoted $e_1 \cong^\mathcal{SE}_\ell e_2$, if for all values $s$ and expression contexts $C$ such that $\vdash s : \sigma \diamond \text{pc} \vdash C[e_i] : L_\ell (\text{unit } + \text{unit})$, and $\langle C[e_i], s \rangle \rightarrow^* \langle v_i, s_i \rangle$ for both $i = 1, 2$, then $v_1 = v_2$ if $\ell_{\text{Exn}} \sqsubseteq \ell$ and $s_1 = s_2$ if $\ell_{\text{State}} \sqsubseteq \ell$.

Intuitively, Definition 2 says that $e_1$ and $e_2$ are equivalent if no program will let an attacker at label $\ell$ distinguish them through either the program output or the state. Because the attacker can only see the program output if $\ell_{\text{Exn}} \sqsubseteq \ell$, we only check output equivalence in that case. Similarly, because the attacker can only see the state cell if $\ell_{\text{State}} \sqsubseteq \ell$, we only check equivalence of the state cells when the flow holds. Note that structural equality on state values is sufficient because we assumed $\sigma$ 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 to programs 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 $\epsilon$ from a set $E$ of possible effects. Therefore, we need to decide on the contents of $E$.

So far, we have described our language has having two effects: state and exceptions. We might therefore let $E = \{S, E\}$ 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 $\text{unit } + \text{unit}$):

\[
x : \text{unit } + \text{unit} \vdash \text{match } x \text{ with } \begin{array}{c}
| \text{inl}(\_ ) \Rightarrow \text{read} : \text{unit } + \text{unit} \\
| \text{inr}(\_ ) \Rightarrow \text{throw} \\
\end{array}
\]

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 $E = 2^{\{S, E\}}$. 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:

\[
\text{unlabel } h \text{ as } x \\
\text{in match } x \text{ with } \begin{array}{c}
h : L_\ell (\text{unit } + \text{unit}) \vdash \text{match } x \text{ with } \begin{array}{c}
| \text{inl}(\_ ) \Rightarrow \text{label}_\ell(\text{read}) : L_\ell (\text{unit } + \text{unit}) \\
| \text{inr}(\_ ) \Rightarrow \text{label}_\ell(\text{inl}()) \\
\end{array}
\end{array}
\]

An attacker who cannot distinguish values labeled $\ell$ 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 $E$ 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 $\Gamma \vdash e : \tau \circ \epsilon$, where $\epsilon \subseteq \{R, W, E, \}$. 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 $\Gamma \vdash e : L_\ell (\text{unit + unit}) \circ R, E$.

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

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 $\epsilon$, 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 $\lambda$-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:

$$
\begin{align*}
\Gamma, x : \tau_1 \vdash e : \tau_2 \circ \epsilon & \quad \Gamma \vdash e_1 : \tau_1 \xrightarrow{\epsilon_1} \tau_2 \circ \epsilon_2 \\
\Gamma \vdash \lambda x : \tau_1. e : \tau_2 \circ \epsilon & \quad \Gamma \vdash e_2 : \tau_1 \circ \epsilon_3 \\
\Gamma \vdash e_1 e_2 : \tau_2 \circ \epsilon_1 \cup \epsilon_2 \cup \epsilon_3 & \quad \ell \circ \tau_2 \quad \ell \subseteq \ell_\epsilon \\
\ell \vdash \tau_2 & \quad \ell \vdash \tau_1 \xrightarrow{\epsilon} \tau_2
\end{align*}
$$

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

$$
\begin{align*}
\Gamma \vdash \text{read} : \sigma \circ R & \quad \Gamma \vdash \text{write}(e) : \text{unit} \circ \epsilon \cup W \\
\Gamma \vdash \text{throw} : \tau \circ E
\end{align*}
$$

Reading a value gives an R effect, while $\text{write}(e)$ evaluates $e$ and therefore any effects $\epsilon$ 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 $\ell_{\text{Exn}} \wedge \tau$. The resulting rule is as follows.

$$
\begin{align*}
\Gamma \vdash e_1 : \tau \circ \epsilon_1 \cup E & \quad \Gamma \vdash e_2 : \tau \circ \epsilon_2 & \quad \ell_{\text{Exn}} \wedge \tau \\
\Gamma \vdash \text{try} \{e_1\} \text{ catch } \{e_2\} : \tau \circ \epsilon_1 \cup \epsilon_2
\end{align*}
$$

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

$$
\begin{align*}
\Gamma \vdash e_1 : L_\ell (\tau_1) \circ \epsilon_1 & \quad \Gamma, x : \tau_1 \vdash e_1 : \tau_2 \circ \epsilon_2 \quad \ell \circ \tau_2 \quad \ell \subseteq \ell_{e_2} \\
\Gamma \vdash \text{unlabel } e_1 \text{ as } x \text{ in } e_2 : \tau_2 \circ \epsilon_1 \cup \epsilon_2
\end{align*}
$$

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

$$
\begin{align*}
\Gamma \vdash e : \tau \circ \epsilon' & \quad \epsilon' \subseteq \epsilon \\
\Gamma \vdash e : \tau \circ \epsilon
\end{align*}
$$

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 \( \Gamma \vdash \text{pc} \vdash e : \tau \), then \( \Gamma \vdash e : \tau \circ \varepsilon \) for some \( \varepsilon \). Moreover, the \( \text{pc} \) controls which effects are possible. In particular:

- If \( \text{pc} \not\in \ell_{\text{State}} \) and \( \Gamma \vdash \text{pc} \vdash e : \tau \), then there exists an \( \varepsilon \) such that \( \Gamma \vdash e : \tau \circ \varepsilon \) and \( W \not\in \varepsilon \).
- If \( \text{pc} \not\in \ell_{\text{Exn}} \) and \( \Gamma \vdash \text{pc} \vdash e : \tau \), then there exists an \( \varepsilon \) such that \( \Gamma \vdash e : \tau \circ \varepsilon \) and \( E \not\in \varepsilon \).

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 \( \text{pc} \) 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 Appendix C.1 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 \( \{W\} \) and \( \{R, W\} \), as well as for \( \{W, E\} \) and \( \{R, W, E\} \).

We also assume for simplicity that \( \sigma \), the type of our state cell, already has at least \( \ell_{\text{State}} \) sensitivity—that is, \( \ell_{\text{State}} \triangleleft \sigma \)—allowing us to omit explicit use of \( \ell_{\text{State}} \). If this were not the case, we would use \( L_{\ell_{\text{State}}} (\sigma) \) instead of \( \sigma \).

<table>
<thead>
<tr>
<th>( \varepsilon )</th>
<th>( P_\varepsilon (\tau) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varnothing )</td>
<td>( \tau )</td>
</tr>
<tr>
<td>( {R} )</td>
<td>( \sigma \rightarrow \tau )</td>
</tr>
<tr>
<td>( {E} )</td>
<td>( L_{\ell_{\text{Exn}}} (\text{unit} + \tau) )</td>
</tr>
<tr>
<td>{W} and {R, W}</td>
<td>( \sigma \rightarrow (\tau \times \sigma) )</td>
</tr>
<tr>
<td>{R, E}</td>
<td>( \sigma \rightarrow L_{\ell_{\text{Exn}}} (\text{unit} + \tau) )</td>
</tr>
<tr>
<td>{W, E} and {R, W, E}</td>
<td>( \sigma \rightarrow (L_{\ell_{\text{Exn}}} (\text{unit} + \tau) \times \sigma) )</td>
</tr>
</tbody>
</table>

We refer to the monad for a (set of) effect(s) \( \varepsilon \) as \( P_\varepsilon \). We use this notation instead of the more-traditional \( M_\varepsilon \) as we will generalize to a producer 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 \( \{R, W, E\} \) is \( \sigma \rightarrow (\text{unit} + \sigma) \times \sigma \) instead of \( \text{unit} + (\sigma \rightarrow \tau) \times \sigma \) 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 \( \varepsilon \) and program \( \Gamma, x : \tau_1 \vdash p : P_\varepsilon (\tau_2) \), we can define \( \Gamma, x : P_{\varepsilon_1}(\tau_1) \vdash \text{bind}_\varepsilon (p) : P_{\varepsilon_2}(\tau_2) \).
- For any type \( \tau \) and set \( \varepsilon \), we can define \( \eta_\varepsilon : \tau \rightarrow P_\varepsilon (\tau) \).
- For any type \( \tau \) and pair of sets \( \varepsilon_1 \) and \( \varepsilon_2 \) such that \( \varepsilon_1 \subseteq \varepsilon_2 \), we can define a program \( \text{coerce}_{\varepsilon_1 \rightarrow \varepsilon_2} : P_{\varepsilon_1}(\tau) \rightarrow P_{\varepsilon_2}(\tau) \).

---

2This is because the writer monad, for the write effect without read, is not a monad unless \( \sigma \) is a monoid. However, the state monad, for read and write effects, is a cartesian monad no matter the type of \( \sigma \).
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 $\Gamma \vdash e : \tau \circ \varepsilon$ into a program $e'$ such that $\Gamma \vdash e' : P_\varepsilon(\tau)$. See Appendix C 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 $L_{\ell_{\text{Exn}}}\left((\text{unit} + \text{unit}) \times \sigma\right)$ after unlabeling a value of type $L_{\ell_{\text{Exn}}}\left((\text{unit} + \text{unit})\right)$. 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 $\Gamma \circ pc \vdash e_1 : \tau$ and $\Gamma \circ pc \vdash e_2 : \tau$, and $\ell \circ \tau$ and $\ell \sqsubseteq pc$, then for all labels $\ell_{\text{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\text{Atk}}$ or $e_1 \equiv_{\ell_{\text{Atk}}} e_2$.

**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 $\ell \sqsubseteq pc$ 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 $\ell_{\text{Atk}}$ 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 $pc = \top$.

### 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:

Expressions $e ::= \ldots | \text{fix } f : \tau. e$

$$
\frac{\Gamma \vdash f : \tau}{\Gamma \vdash \text{fix } f : \tau. e : \tau}
$$

Fix $\text{fix } f : \tau. e \rightarrow e[f \mapsto \text{fix } f : \tau. e]$
Because programs may not terminate in this extended language, the fact that our definition of $\ell$-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 $\ell$-equivalence, and use it to analyze security.

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

We follow the same approach as in Section 3 to ensure noninterference with respect to this stronger definition: we restrict effects with a $\text{pc}$ label, build a corresponding type-and-effect system, and prove security by translating to DCC with its termination-insensitive guarantee. Because nontermination from $\text{fix}$ 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 $\ell_{\text{PNT}} \in L$ such that any attacker who can read $\ell_{\text{PNT}}$ will eventually infer information from nontermination, but others will not. This gives rise to the following rule:

\[ \frac{\Gamma, f : \tau \circ \text{pc} \vdash e : \tau \quad \text{pc} \sqsubseteq \ell_{\text{PNT}}}{\Gamma \circ \text{pc} \vdash \text{fix} f : \tau.e : \tau} \]

The rule ensures that only data available at label $\ell_{\text{PNT}}$—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 $\ell_{\text{PNT}} = \top$, model traditional termination-sensitivity using a bottom label $\bot$ by setting $\ell_{\text{PNT}} = \bot$, 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: $\emptyset$ or $\text{PNT}$ with labels $\top$ and $\ell_{\text{PNT}}$, respectively. The typing rule for the fixed-point operator is then:

\[ \frac{\Gamma, f : \tau \vdash e : \tau \circ \varepsilon}{\Gamma \vdash \text{fix} f : \tau.e : \tau \circ \text{PNT}} \]

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. Pointed types are also used in defining the denotational semantics of DCC. Abadi et al. [1999] define the denotational semantics of DCC using Scott domains, and give pointed types semantics using domains with a bottom element. Then, fixed points of computations in a pointed types
can be found as usual.

| Types                | $\tau ::= \cdots | \tau_{\bot}$ |
|----------------------|------------------|
| Expressions          | $e ::= \cdots | \text{lift}(e) | \text{seq } x = e_1 \text{ in } e_2 | \text{fix } f : \tau. e$ |
| Evaluation Contexts  | $E ::= \cdots | \text{lift}(E) | \text{seq } x = E \text{ in } e$ |

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

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

\[
\Gamma \vdash e : \tau \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f : \tau \vdash e : \tau \quad \Gamma \vdash \text{seq } x = e_1 \text{ in } e_2 : \tau_2 \quad \Gamma \vdash \text{fix } f : \tau. e : \tau
\]

\[
\begin{align*}
\vdash \tau_{\bot} & \quad \vdash \tau_1 \times \tau_2 & \quad \vdash \text{L}_\ell(\tau) & \quad \vdash \tau_1 \rightarrow \tau_2
\end{align*}
\]

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

\[
\begin{align*}
\begin{array}{c}
\Gamma \vdash e : \tau \\
\Gamma \vdash \text{lift}(e) : \tau_{\bot}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{seq } x = e_1 \text{ in } e_2 : \tau_2
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\Gamma, f : \tau \vdash e : \tau \\
\Gamma \vdash \text{fix } f : \tau. e : \tau
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{seq } x = \text{lift}(v) \text{ in } e & \rightarrow e[x \mapsto v]
\end{align*}
\]

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 $\sigma \rightarrow (\tau \times \sigma)$ to be pure, but not a program which accesses state and returns a value of type $\tau$? 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 $\sigma$, and writes by returning an appropriate result of type $\sigma$. 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 easier, as noted above. Second, call-by-value DCC with pointed types enforces termination-insensitive noninterference, which is generally simpler to prove than the 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 $\ell_{\text{PNT}}$ 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:

\[
\ell \sqsubset \ell_{\text{PNT}} \quad \ell \sqsubset \tau
\]

\[
\ell \sqsubset \tau_{\bot}
\]
Note that, even when setting \( \ell_{\text{PNT}} = \bot \), this rule allows public data to influence termination behavior. We therefore differ slightly from Abadi et al.'s [1999] definition in systems where public and unlabeled data are different.

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 \( \text{pc} \) system to the type-and-effect system in an analogous way to Lemma 1. If the \( \text{pc} \) cannot influence \( \ell_{\text{PNT}} \), then the program must terminate. We can formalize this as follows:\(^3\)

**Lemma 2.** If \( \Gamma \vdash \text{pc} \vdash e : \tau \), then \( \Gamma \vdash e : \tau \circ \text{PNT} \). Moreover, if \( \text{pc} \not\succeq \ell_{\text{PNT}} \), then \( \Gamma \vdash e : \tau \circ \varnothing \).

The first statement 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 Appendix C), it tells us that if the \( \text{pc} \) 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 \( \Gamma \vdash \text{pc} \vdash e_1 : \tau \) and \( \Gamma \vdash \text{pc} \vdash e_2 : \tau \), and \( \ell \triangleleft \tau \) and \( \ell \not\succeq \text{pc} \), then for all labels \( \ell_{\text{Atk}} \in \mathcal{L} \) where \( \ell_{\text{PNT}} \not\succeq \ell_{\text{Atk}} \), either \( \ell \not\succeq \ell_{\text{Atk}} \) or \( e_1 \mathrel{\equiv}^\mathcal{E} \text{pc} e_2 \).

This is a special case of Theorem 5. As with both previous noninterference theorems (Theorems 1 and 2), this theorem says data at label \( \ell \) cannot leak to an attacker who cannot distinguish values at level \( \ell \). This time, however, the attacker can glean information from nontermination. Moreover, if we set \( \ell_{\text{PNT}} = \bot \), then \( \bot \not\succeq \ell_{\text{Atk}} \) for all \( \ell_{\text{Atk}} \), 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 \( \text{pc} \not\succeq \ell_{\text{PNT}} \), which then guarantees \( \ell \not\succeq \ell_{\text{Atk}} \) by transitivity.

### 5 A Framework for Effectful Labeled Languages

We have now twice given semantics to \( \text{pc} \) systems using type-and-effect systems and monadic translations. The ability to analyze 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 results by examining what language properties are necessary to obtain the security results from Sections 3 and 4.

We therefore move 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 are required for our analysis. This approach allows us to make strong guarantees about any language that has those properties, regardless of what other features may be present.

As with our examples, we have two languages, one effectful and one pure. The effectful language is defined with a type-and-effect system and judgments take the form \( t \vdash \text{pc} p \vdash t' \circ \varepsilon \). Typing judgments for the pure language are simply \( \tau \vdash p \vdash t' \circ \varepsilon \). 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. Note that both judgments are single-input, single-output. This restriction follows Tate [2013], who invented *productors*, the main structure we are using in this section. In Appendix A, we show how to remove this restriction.

To define the requisite structures on the effectful language we need a collection of effects \( \mathcal{E} \). In Sections 3 and 4, \( \mathcal{E} \) formed a lattice, allowing us to give meaning to sequential composition of programs with different effects. A lattice, however, is far more structure than is necessary. Instead we use an *effector* [Tate, 2013], which was designed as the minimal structure required to give meaning to such compositions.

\(^3\)As with Lemma 1, we implicitly assume a simple translation between the two sets of type constructors (see Appendix C.1).
An effector is a set $E$ 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 $\varepsilon_1$ through $\varepsilon_n$, respectively, can result in a program with effect $\varepsilon$. Note that $n$ may in particular be zero or one, where $[\varepsilon] \leq \varepsilon$ means that a program judged to have effect $\varepsilon$ may be pure, and $[\varepsilon] \leq \varepsilon'$ means that a program judged to have effect $\varepsilon'$ may also have effect $\varepsilon$. The relation must follow appropriate versions of identity and associativity laws, reflecting these intuitions [Tate, 2013, Section 5].

Using effectors, we can define the requisite typing rules on each language (Figure 3). To begin, $\text{SEQ}_{\leq}$ and $\text{SEQ}$ require both languages to have a notion of $n$-way sequential composition. (When $n = 0$, we have an identity program which changes nothing: $\tau \vdash_p - \tau$.) In the effectful language, composition is allowed when the effects of the programs compose. In the pure language, composition is always allowed.

We also require a rule, $\text{CAPTURE}$, that allows us to convert a program from the effectful language to the pure one.\footnote{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.} We denote this translation $[-]_\varepsilon$. Note also that the pure and effectful languages may use different types. For instance, in our example languages, functions in the type-and-effect system were annotated by an effect, whereas pure functions were not. We therefore need a translation $\langle - \rangle$ to translate effectful types to pure ones. This handles the translation of the annotated function type $\tau_1 \leq \tau_2$ to the pure, unannotated function type $\tau_1 \rightarrow P_\varepsilon(\tau_2)$ in our example languages.

Moreover, we need to ensure that $[p]_\varepsilon$ properly represents the effect $\varepsilon$ of the program $p$. This is done by changing the output type to represent not only the outputs of $p$ but also $\varepsilon$. This change must be a mathematical function of effect and type, which we write $P_\varepsilon(\cdot)$.

This second translation $P_\varepsilon(\cdot)$ must have two additional pieces of structure to allow us to faithfully translate sequential composition of effectful programs. First, we require a pure program transformer $\text{map}_\varepsilon$ for every effect $\varepsilon$ with the typing rule $\text{MAP}$: it takes a program from $\tau$ to $\tau'$ and transforms it into a program from $P_\varepsilon(\tau)$ to $P_\varepsilon(\tau')$. Second, for any list of effects $\varepsilon_1,\ldots,\varepsilon_n$ and $\varepsilon$ such that $[\varepsilon_1,\ldots,\varepsilon_n] \leq \varepsilon$, we require a pure program $\text{join}_{[\varepsilon_1,\ldots,\varepsilon_n]}$ with typing rule $\text{JOIN}$: it takes an input of type $P_{\varepsilon_1}(\cdots P_{\varepsilon_n}(\tau) \cdots)$ and produces an output of type $P_\varepsilon(\tau)$.
by capturing each program individually and using pure composition, map, and join appropriately. Here \( \rho_1 \xrightarrow{\tau_1, \tau_2} \rho_2 \) means “\( \rho_1 \) and \( \rho_2 \) are equal as pure programs from \( \tau_1 \) to \( \tau_2 \).”

Map and Join must also follow two simple rules, similar to the rules for effectors,\(^5\) to ensure that the monad-like structure \( P_{\_}(-) \) forms a productor [Tate, 2013]. Note that productors are a generalization of monads. If we have only one effect \( \varepsilon \) such that \( [\varepsilon, \varepsilon] \leq \varepsilon \) and \( [] \leq \varepsilon \), then the above defines a monad. Productors also generalize indexed monads [Orchard et al., 2014; Wadler and Thiemann, 1998] and graded monads [Fujii et al., 2016; Katsumata, 2014]. An indexed monad is a productor for a lattice, the structure we had in Sections 3 and 4, while a graded monad is a productor for an ordered monoid; both are examples of effectors.

Our framework up until this point is a slight variant of Tate’s [2013] framework for languages with a producer effect. While Tate’s framework assumed a single language with both pure and effectful programs, we have assumed two languages, one pure and one effectful. Moreover, Tate did not assume Map and Join directly, but rather assumed an effectful program which would execute the effects stored in a captured type. He then used this program to prove Map and Join were admissible. We work in this more-general framework, which allows the language of Section 4 to fit.

We also make a few assumptions about labels. We assume both the effectful and pure languages define protection relations, and we relate them with the ProtectC rule. This rule says that if both an effectful type \( t \) and an effect \( \varepsilon \) have sensitivity at least \( \ell \), then \( P_{\varepsilon}(\{t\}) \) must as well. Our examples admitted this rule by composing two simpler rules: protection transfers directly from the effectful to the pure type system, and if a pure type \( \tau \) and an effect \( \varepsilon \) have sensitivity at least \( \ell \), then \( \ell \leq P_{\varepsilon}(\tau) \). Our results, however, only require the single combined rule.

Note that, while we assume that both languages define a protection relation, this does not have to be the main security mechanism of the language. Instead, it 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 a label associated with that type. 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.

Finally, in our examples we defined noninterference using notions of equivalence that reflected what an attacker could see. To transfer a noninterference result from the pure language to the effectful one, both languages need such notions of equivalence, and those notions must correspond to each other. In Section 3, for example, effectful equivalence considered both the program output and the state cell. By making the state cell part of the term when translating to the pure language, we ensured that the pure equivalence gave rise to the effectful equivalence. In general, the EQUIVCAP rule requires that the pure notion of equivalence, \( \approx_{\_} \), be at least as precise on captured programs as the effectful equivalence, \( \approx_{\varepsilon} \), is on uncaptured ones.

A categorically-minded reader may have noticed that our pure and effectful languages each define categories where the category of effectful programs is essentially the Kleisli category for a monad-like structure on the category of pure programs. Indeed, productors are the most general structure known for studying producer effects, which are effects that have a Kleisli-like structure [Tate, 2013].

This framework will allow us to prove a strong and general security theorem for effectful programming languages. Moreover, it connects us 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 \( \ell_{\text{State}} \subseteq \ell_{\text{Exn}} \). Recall that when we compose a program \( p \) that may throw an exception with a program \( q \) that may

\(^5\)Essentially, (1) naturality of \( \text{join}_{[-],\_} \), (2) \( \text{join}_{[\varepsilon],\_} \) is the identity, and (3) when \( [\varepsilon_i] \leq \varepsilon_i \) and \( [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon \), the two evident ways of using \( \text{join}_{[-],\_} \) to get from \( P_{\varepsilon_1}(\_ \cdots \_ \cdots P_{\varepsilon_n}(\_ \cdots)) \) to \( P_{\varepsilon}(\_) \) are equal.
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 \( \varepsilon \), \( [p] \) wraps its output in \( \ell_{\text{Exn}} \). 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 \( \ell_{\text{Exn}} \subseteq \ell_{\varepsilon} \).

By assuming \( \ell_{\text{Exn}} \subseteq \ell_{\text{State}} \), 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:

\[
\begin{align*}
\varepsilon_1 \cup \varepsilon_2 & \subseteq \varepsilon_3 \quad E \notin \varepsilon_1 \\
\ell_{\text{Exn}} & \subseteq \ell_{\varepsilon_2} \\
[\varepsilon_1, \varepsilon_2] & \leq \varepsilon_3
\end{align*}
\]

Not that this makes the composition relation partial, since not all pairs of effects compose. This means that \( E \) 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 \( \ell \) can safely compose with a program \( q \) that writes data exclusively at or above label \( \ell \).

6 The Noninterference Half-Off Theorem

Our main aim is to show how, given a \( \text{pc} \) system, we can give semantics to the \( \text{pc} \) 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 compile 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 \( \ell_{\text{Atk}} \) can only distinguish data and effects at or below label \( \ell_{\text{Atk}} \). 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 \( \ell_{\varepsilon} \) to the effect \( \varepsilon \) to represent \( \varepsilon \)'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 \( t_1 \vdash_E r \models t_2 \circ \varepsilon_1 \). 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
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 \( \equiv _{\ell } \).

Proof. Unfolding Definition 4, we have programs \( p, q \), and \( r \) and a label \( \ell \) such that

1. \( t_3 \vdash _{\ell } p \triangleright t_1 \circ \varepsilon _2 \) and \( t_3 \vdash _{\ell } q \triangleright t_1 \circ \varepsilon _2 \) with \( [\varepsilon _2, \varepsilon _1] \leq \varepsilon \),
2. \( \ell \circ t_1 \) and \( \ell \subseteq \ell _{\text{Atk}} \)

then for all labels \( \ell _{\text{Atk}} \in \mathcal{L} \), either \( \ell \subseteq \ell _{\text{Atk}} \) 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) \( \varepsilon _2 \). 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, \( \varepsilon _2 \), of the input programs be at least \( \ell \). 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.

Let \( \rho = \text{map}_{\varepsilon _2}(\varepsilon _2, t_1) \); \( \text{join}_{\varepsilon _2, \varepsilon _1} \); \( \varepsilon _1 \). The rules in Figure 3 guarantee \( P_{\varepsilon _2}(t_1) \vdash _{p} p \triangleleft P_{\varepsilon _2}(t_2) \) and \( \text{CAPTURESEQ} \) requires \( [p ; r]_{\varepsilon _2} \); \( [q]_{\varepsilon _2} ; \rho \), and similarly for \( q \). All well-typed pure programs are noninterfering with respect to \( \approx _{\ell } \), \( \rho \) is pure, and \( \ell \triangleleft P_{\varepsilon _2}(t_1) \) by \( \text{PROTECTC} \). Thus, for any label \( \ell _{\text{Atk}} \), either \( \ell \subseteq \ell _{\text{Atk}} \) or \( [p]_{\varepsilon _2} ; \rho \approx _{\ell _{\text{Atk}}} [q]_{\varepsilon _2} ; \rho \). In the first case, we are finished. In the second case, the program equality above and \( \text{EQUIVCAP} \) give us

\[
[p]_{\varepsilon _2} ; \rho \approx _{\ell _{\text{Atk}}} [q]_{\varepsilon _2} ; \rho \iff [p ; r]_{\varepsilon _2} \approx _{\ell _{\text{Atk}}} [q ; r]_{\varepsilon } \Rightarrow p ; r \approx _{\ell _{\text{Atk}}} q ; r. \]

\( \square \)

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 \( \text{pc} \) label. We extend the framework to include a \( \text{pc} \) system with judgments of the form \( t \circ \text{pc} \vdash _{\ell } p \triangleleft t' \). (We still use Roman letters for types and programs in the \( \text{pc} \) system, but we color them in green.) Figure 4 shows the rules we require for this extended framework.

We give semantics to the \( \text{pc} \) 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 \( \text{pc} \) 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 \( \tau_1 \xrightarrow{\text{pc}} \tau_2 \), while the type-and-effect system included an effect \( \tau_1 \xrightarrow{\text{e}} \tau_2 \). We therefore allow the pc system to have different types, but the same programs, and require there to be a translation \([\cdot]\) 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 \textsc{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 \( \sim \).

**Proof.** This follows directly from PCEFF and Theorem 4. We note that this always instantiates the \( \varepsilon \) in \( \approx \) with the same \( \varepsilon \) 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 Theorem 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 \( t \circ \text{pc} \vdash \text{pc} \ p : t' \), there is some \( \varepsilon \) such that \( [t] \vdash_E p \vdash [t'] \circ \varepsilon \) where \( \text{pc} \sqsubseteq \ell_\varepsilon \).

Note that Definition 5 only requires that the pc bound some \( \varepsilon \). We might at first think that any \( \varepsilon \) 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 \( \ell_\top \), since \( \ell_\top = \top \). However, by variance \( p \) can also type-check with effect \( \{R, W, E\} \), and
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 \(\mathit{pc}\) to enforce noninterference should have \(\mathit{pc}\)-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 \textit{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 \(\text{inl}(\ell_1)\) and \(\text{inr}(\ell_2)\). 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:

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \implies \quad \Gamma \vdash \text{label}_{\text{inl}(\ell)}(e) : L_{\text{inl}(\ell)}(\tau) \\
\Gamma \vdash e : \tau & \quad \implies \quad \Gamma \vdash \text{label}_{\text{inr}(\ell)}(e) : L_{\text{inr}(\ell)}(\tau) \\
\Gamma \vdash e : \tau & \quad \implies \quad \Gamma \vdash \text{label}_{\top}(e) : L_{\text{inl}(\top)}(\tau)
\end{align*}
\]

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 \(\ell_e\) to always be of the form \(\text{inr}(\ell)\) 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 \(\ell\_\) to the full label space by defining \(\ell_e = \text{inr}(\hat{\ell}_e)\).

Finally, we modify the rules that compare the \(\mathit{pc}\) and effect labels by explicitly comparing the \(\mathit{pc}\) to the data-label analogue of the effect’s label. Formally, we use the following rules:

\[
\begin{align*}
\Gamma \odot \mathit{pc} \vdash e : \sigma & \quad \implies \quad \mathit{pc} \sqsubseteq \text{inl}(\hat{\ell}_W) \\
\Gamma \odot \mathit{pc} \vdash \text{write}(e) : \text{unit} & \quad \implies \quad \mathit{pc} \sqsubseteq \text{inl}(\hat{\ell}_E) \\
\Gamma \odot \mathit{pc} \vdash \text{throw} : \tau & \quad \implies \quad \mathit{pc} \sqsubseteq \text{inl}(\hat{\ell}_E)
\end{align*}
\]

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

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

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

Consider a program \(p\) in one of our example type-and-effect systems such that \(\Gamma, x : \tau \vdash p : \tau' \odot e\). 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 \(\text{map}_\ell\) that we can type-check as \(\Gamma, x : L_\ell(\tau) \vdash \text{map}_\ell(p) : L_\ell(\tau') \odot e\). Notably, this does not change the effect. To retain security, we must ensure that \(\ell \sqsubseteq \ell_e\), since \(p\) may otherwise leak data about the \(\ell\)-labeled input in its effects.

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

\textbf{Theorem 6.} The effects of any language admitting the rules in Figures 3, 4, and 5 are

\textbf{Proof.}

monotone

strengthen Theorem 6 by providing a witness for the existential in Definition 5. However, we will see that they can be used here to

arbitrary sets it acts as a lower bound. Note that because of this

\textbf{Section 3.} We used a function

\textbf{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 \( \gamma \) from labels to effects such that if a program type-checks with program-counter

label \( \text{pc} \) then it type-checks with effect \( \gamma(\text{pc}) \). Let us examine this in detail for the example language from Section 3. We used a function \( \ell_- \) on effects such that \( \ell_\{E\} = \ell_{\text{Exn}}, \ell_\{R\} = \top, \) and \( \ell_\{W\} = \ell_{\text{State}} \) and on arbitrary sets it acts as a lower bound. Note that because of this \( \ell_- \) must be antitone: if \( \varepsilon \subseteq \varepsilon' \), then \( \ell_{\varepsilon'} \subseteq \ell_\varepsilon \).

Given a particular \( \ell_- \), we can then define the function \( \gamma \) from labels to effects as follows:

\[ \gamma(\ell) = \{ R \} \cup \{ W \mid \ell \subseteq \ell_{\text{State}} \} \cup \{ E \mid \ell \subseteq \ell_{\text{Exn}} \} \]

The functions \( \ell_- \) and \( \gamma \) form a \textit{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 \( \ell \) is antitone, it seems like it cannot be part of a Galois connection, as Galois connections are defined on \textit{monotone} functions. However, it turns out \( \ell_- \) and \( \gamma \) more precisely form an \textit{antitone} Galois
connection [see e.g., Galatos, 2007]. An antitone Galois connection between lattices \( \mathcal{A} \) and \( \mathcal{B} \) is equivalent to a monotone Galois connection between \( \mathcal{A} \) and \( \mathcal{B}^{\text{op}} \), the order dual of \( \mathcal{B} \).

**Lemma 3 (Antitone Galois Connection).** The functions \( \ell \) and \( \gamma \) form an antitone Galois connection. That is, for any label \( \ell \) and effect \( \varepsilon \), \( \ell \sqsubseteq \varepsilon \) if and only if \( \varepsilon \subseteq \gamma(\ell) \).

**Proof.** By examining all of the (eight) possible values of \( \varepsilon \). \( \square \)

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 \( [\varepsilon] \leq \varepsilon' \) 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 \( \gamma \) such that \( \ell \) and \( \gamma \) form an antitone Galois connection, then whenever \( t \diamond pc \vdash p \Rightarrow t' \), then \( [t] \vdash_E p \Rightarrow [t'] \diamond (pc) \).

**Proof.** By Theorem 6, we know that there is some \( \varepsilon \) such that \( [t] \vdash_E p \Rightarrow [t'] \diamond \varepsilon \) and \( pc \subseteq \ell_\varepsilon \). Because \( pc \subseteq \ell_\varepsilon \) and \( \ell \) and \( \gamma \) form an antitone Galois connection, \( [\varepsilon] \leq \gamma(pc) \). This means we can further apply \( \text{SEQ} \leq \) to get \( [t] \vdash_E p \Rightarrow [t'] \diamond \gamma(pc) \). \( \square \)

## 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 DCCpc [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 anonymous reviewers provided tremendously insightful comments and helpful suggestions.

This work was funded in part by a National Defense Science and Engineering Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and may not reflect those of these sponsors.

References


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 producer can be viewed as a 2-functor from an effector to the category of categories. Thus, each effect \(\varepsilon\) is mapped to a functor \((P_\varepsilon(-), \text{map}_\varepsilon)\), 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 producer.

To understand this, let us define a premonoidal category \(\mathcal{C}\) [Jeffrey, 1997]:

**Definition 6** (Premonoidal Category). A premonoidal category is a category \(\mathcal{C}\) along with
- A binary operation on objects, written \(- \otimes -\)
- For every object \(\Gamma\), two functors \(\Gamma \times -\) and \(- \times \Gamma\), 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 \(\Gamma\), and we can define \(\Gamma_1, \Gamma_2\) as \(\Gamma_1 \otimes \Gamma_2\). Then, for any morphism \(\rho : \Gamma_1 \rightarrow \Gamma_2\), we can think of \(\rho\) as a program that operates in an environment \(\Gamma_1\), and then finishes having changed the environment to \(\Gamma_2\). Then, \(\Gamma \times \rho : \Gamma \times \Gamma_1 \rightarrow \Gamma \times \Gamma_2\), so we have propagated the context \(\Gamma\).

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

\[
\Gamma_1 \vdash \rho \rightarrow \Gamma_2 \\
\rightarrow \Gamma_1, \Gamma \vdash \rho \times \Gamma \rightarrow \Gamma_2, \Gamma
\]

Here, we write \(\Gamma_1 \vdash \rho \rightarrow \Gamma_2\) as the typing judgment representing a program \(\rho : \Gamma_1 \rightarrow \Gamma_2\).

Then, a strong producer is simply a producer 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:

\[
\Gamma_1 \vdash \rho \rightarrow \Gamma_2 \\
\rightarrow \Gamma_1, \Gamma \vdash \rho \times \Gamma \rightarrow \Gamma_2, \Gamma
\]

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

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

In order to give semantics to this rule, we need an extra few assumptions. In particular, we need a doubling natural transformation \(\Delta_\Gamma : \Gamma \rightarrow \Gamma \otimes \Gamma\), which is preserved by \(\text{map}_\varepsilon\). That is, we need the following rules to
be admissible:

\[
\Gamma \vdash \Delta \Gamma \Rightarrow \Gamma, \Gamma' \quad \text{map}_\varepsilon(\Delta \Gamma) \xmapsto{P_\varepsilon(\Gamma), (P_\varepsilon(\Gamma'), P_\varepsilon(\Gamma))} \Delta P_\varepsilon(\Gamma)
\]

This allows us to prove the following:

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

\[
\Gamma \vdash p_1 : t_1 \circ \varepsilon_1 
\cdots 
\Gamma, x_1 : t_1, \ldots, x_{n-1} : t_{n-1} \vdash p_n : t_n \circ \varepsilon_n \quad [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon
\]

\[
\text{let } x_1 = p_1
\]

\[
\vdots
\]

\[
\text{in } 
\Gamma \vdash \ell (\sigma) = L
\]

\[
\Gamma \vdash \ell (\Gamma_1, \Gamma_2) = L(\Gamma_1, \Gamma_2)
\]

\[
\Gamma_1, L(\Gamma_2) \vdash \text{str}_{L_\ell, \Gamma_1, \Gamma_2} \Rightarrow L(\Gamma_1, \Gamma_2)
\]

Then, extending Theorem 8 to unlabel is not difficult.

**B Full Rules for Example Languages**

**B.1 DCC**

Here we present the full type system for our fragment of DCC from Section 2

**VAR**

\[
\Gamma \vdash e \vdash \tau \quad \Gamma \vdash x : \tau
\]

**UNIT**

\[
\Gamma \vdash () : \text{unit}
\]

**PAIR**

\[
\Gamma \vdash \ell_1 : \tau_1, \Gamma \vdash \ell_2 : \tau_2
\]

\[
\Gamma \vdash (\ell_1, \ell_2) : \tau_1 \times \tau_2
\]

**INR**

\[
\Gamma \vdash \ell : \tau_1 \quad \Gamma \vdash \text{inr}(\ell) : \tau_1 + \tau_2
\]

**MATCH**

\[
\Gamma \vdash \ell_1 : \tau_1 + \tau_2
\]

\[
\Gamma, x : \tau_1 \vdash \ell_1 : \tau_1 + \tau_2, \quad \Gamma, y : \tau_2 \vdash \ell_2 : \tau_2 \\
\Gamma \vdash \text{match } \ell_1 \text{ with } \text{inr}(x) \Rightarrow \ell_2 \text{ } | \text{inl}(y) \Rightarrow \ell_2 \text{ end : } \tau
\]

**LAM**

\[
\Gamma, x : \tau_1 \vdash e : \tau_2
\]

\[
\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \Rightarrow \tau_2
\]

**APP**

\[
\Gamma \vdash f : \tau_1 \Rightarrow \tau_2, \quad \Gamma \vdash e : \tau_1
\]

\[
\Gamma \vdash f e : \tau_2
\]

**LABEL**

\[
\Gamma \vdash e \vdash \tau
\]

\[
\Gamma \vdash \text{label}_{\ell} (e) : L(\tau)
\]

**UNLABEL**

\[
\Gamma \vdash e_1 : L(\tau_1), \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2, \quad \ell \in \tau_2
\]

\[
\Gamma \vdash \text{unlabel } e_1 \text{ as } x \text{ in } e_2 : \tau_2
\]
Protection Rules

\[
\begin{align*}
\ell \subseteq \ell' & \quad \ell \triangleq \ell' \tau \quad \ell \triangleq \tau \quad \ell \triangleq \tau_1 \quad \ell \triangleq \tau_2 \quad \ell \triangleq \tau_1 \times \tau_2 \quad \ell \triangleq \tau_1 \rightarrow \tau_2
\end{align*}
\]

Operational Semantics

\[
\begin{align*}
e & \rightarrow e' \\
E[e] & \rightarrow E[e'] \\
(\lambda x : \tau. e) \ v & \rightarrow e[x \mapsto v] \\
\text{proj}_i((v_1, v_2)) & \rightarrow v_i
\end{align*}
\]

match \(\text{inl}(v)\) with \(|\text{inl}(x) \Rightarrow e_1 | \text{inr}(y) \Rightarrow e_2\) end \(\rightarrow e_1[x \mapsto v]\)

match \(\text{inr}(v)\) with \(|\text{inl}(x) \Rightarrow e_1 | \text{inr}(y) \Rightarrow e_2\) end \(\rightarrow e_2[y \mapsto v]\)

unlabel \(\ell\) \((\ell \ e)\) as \(x \in e \rightarrow e[x \mapsto v]\)

B.2 DCC with State and Exceptions

PC Type System

\[
\begin{align*}
\text{VAR} & \quad \Gamma(x) = \tau \\
& \quad \Gamma \vdash x : \tau \\
\text{UNIT} & \quad \Gamma \vdash () : \text{unit} \\
\text{PAIR} & \quad \Gamma \vdash pc \vdash e_1 : \tau_1 \\
& \quad \Gamma \vdash pc \vdash e_2 : \tau_2 \\
& \quad \Gamma \vdash pc \vdash (e_1, e_2) : \tau_1 \times \tau_2 \\
\text{INL} & \quad \Gamma \vdash pc \vdash e : \tau_1 \\
& \quad \Gamma \vdash pc \vdash \text{inl}(e) : \tau_1 + \tau_2 \\
\text{INR} & \quad \Gamma \vdash pc \vdash e : \tau_2 \\
& \quad \Gamma \vdash pc \vdash \text{inr}(e) : \tau_1 + \tau_2 \\
\text{MATCH} & \quad \Gamma \vdash pc \vdash \text{match } e \text{ with } | \text{inl}(x) \Rightarrow e_1 | \text{inr}(y) \Rightarrow e_2 \text{ end} : \tau \\
\text{LAM} & \quad \Gamma, x : \tau_1 \vdash pc \vdash e : \tau_2 \\
& \quad \Gamma \vdash pc \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \\
\text{APP} & \quad \Gamma \vdash pc \vdash f : \tau_1 \rightarrow \tau_2 \\
& \quad \Gamma \vdash pc \vdash e : \tau_1 \\
& \quad \Gamma \vdash pc \vdash e \ e : \tau_2 \\
\text{LABEL} & \quad \Gamma \vdash pc \vdash e : \tau \\
& \quad \Gamma \vdash pc \vdash \text{label}_\ell(e) : \ell_\ell(\tau) \\
\text{UNLABEL} & \quad \Gamma \vdash pc \vdash e_1 : \ell_\ell(\tau_1) \\
& \quad \Gamma, x : \tau_1 \vdash pc \sqcup \ell \vdash e_2 : \tau_2 \\
& \quad \ell \triangleq \tau_2 \\
& \quad \Gamma \vdash pc \vdash \text{unlabel } e_1 \text{ as } x \text{ in } e_2 : \tau_2 \\
\text{VARIANCE} & \quad \Gamma \vdash pc \vdash e : \tau \\
& \quad pc \subseteq pc' \\
& \quad \Gamma \vdash pc \vdash e : \tau \\
\text{READ} & \quad \Gamma \vdash pc \vdash \text{read} : \sigma \\
\text{WRITE} & \quad \Gamma \vdash pc \vdash e : \sigma \\
& \quad pc \subseteq \ell_\text{State} \\
& \quad \Gamma \vdash \text{write}(e) : \text{unit} \\
\text{THROW} & \quad \Gamma \vdash pc \vdash \text{throw} : \tau \\
\text{TRYCATCH} & \quad \Gamma \vdash pc \vdash e_1 : \tau \\
& \quad \Gamma \vdash pc \vdash e_2 : \tau \\
& \quad \ell_\text{Exn} \triangleq \tau \\
& \quad \Gamma \vdash pc \vdash \text{try } \{e_1\} \text{ catch } \{e_2\} : \tau
\end{align*}
\]
PC Protection Rules

\[
\begin{align*}
\ell & \sqsubseteq \ell' \\
\ell & \triangleleft \tau \\
\ell & \triangleleft \tau_1 \\
\ell & \triangleleft \tau_2 \\
\ell & \sqsubseteq \text{pc}
\end{align*}
\]

Type-and-Effect Typing Rules

\[
\begin{align*}
\text{VAR} & \quad \Gamma(x) = \tau \\
& \quad \Gamma \vdash x : \tau \circ \varnothing \\
\text{PAIR} & \quad \Gamma \vdash e_1 : \tau_1 \circ \varepsilon_1 \\
& \quad \Gamma \vdash e_2 : \tau_2 \circ \varepsilon_2 \\
& \quad \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \circ \varepsilon_1 \cup \varepsilon_2 \\
\text{PROJ} & \quad \Gamma \vdash \text{proj}_1(e) : \tau_1 \circ \varepsilon \\
& \quad \Gamma \vdash \text{proj}_2(e) : \tau_2 \circ \varepsilon \\
\text{INL} & \quad \Gamma \vdash \text{inl}(e) : \tau_1 \circ \varepsilon \\
\text{INR} & \quad \Gamma \vdash \text{inr}(e) : \tau_2 \circ \varepsilon \\
\text{MATCH} & \quad \Gamma \vdash \text{match} \ e \ \text{with} \ | \ \text{in} \ (x) \ \Rightarrow \ e_1 \ | \ \text{in} \ (y) \ \Rightarrow \ e_2 \\
& \quad \text{end} : \tau \circ \varepsilon_1 \cup \varepsilon_2 \\
\text{LAM} & \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \circ \varepsilon \\
& \quad \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \circ \varnothing \\
\text{APP} & \quad \Gamma \vdash f : \tau_1 \rightarrow \tau_2 \circ \varepsilon_2 \\
& \quad \Gamma \vdash e : \tau_1 \circ \varepsilon_3 \\
& \quad \Gamma \vdash f \ e : \tau_2 \circ \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \\
\text{UNLABEL} & \quad \Gamma \vdash \text{unlabel} \ e_1 \ \text{as} \ x \ \text{in} \ e_2 : \tau_2 \circ \varepsilon_1 \cup \varepsilon_2 \\
\text{VARIANCE} & \quad \Gamma \vdash e : \tau \circ \varepsilon' \\
& \quad e' \sqsubseteq \varepsilon \\
\text{READ} & \quad \Gamma \vdash \text{read} : \sigma \circ \mathcal{R} \\
\text{WRITE} & \quad \Gamma \vdash \text{write} : \tau \circ \varepsilon \cup \mathcal{W} \\
\text{THROW} & \quad \Gamma \vdash \text{throw} : \tau \circ \mathcal{E} \\
\text{TRYCATCH} & \quad \Gamma \vdash \text{try} \ \{ e_1 \} \ \text{catch} \ \{ e_2 \} : \tau \circ \varepsilon_1 \cup \varepsilon_2 \\
\end{align*}
\]

Type-and-Effect Protection Rules

\[
\begin{align*}
\ell & \sqsubseteq \ell' \\
\ell & \triangleleft \tau \\
\ell & \triangleleft \tau_1 \\
\ell & \triangleleft \tau_2 \\
\ell & \sqsubseteq \text{pc}
\end{align*}
\]
Operational Semantics

\[
\begin{align*}
\langle e, s \rangle &\rightsquigarrow \langle e', s' \rangle & \langle (\lambda x : \tau. e) v, s \rangle &\rightsquigarrow \langle e[x \mapsto v], s \rangle & \langle \text{proj}_1((v_1, v_2)), s \rangle &\rightsquigarrow \langle v_1, s \rangle \\
(E[e], s) &\rightsquigarrow \langle E[e'], s' \rangle & \langle \text{match } \text{inl}(v) \text{ with } | \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2 \text{ end}, s \rangle &\rightsquigarrow \langle e_1[x \mapsto v], s \rangle & \langle \text{match } \text{inr}(v) \text{ with } | \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2 \text{ end}, s \rangle &\rightsquigarrow \langle e_2[y \mapsto v], s \rangle \\
\langle \text{unlabel } \text{label}_\ell(v) \text{ as } x \text{ in } e, s \rangle &\rightsquigarrow \langle e[x \mapsto v], s \rangle & \langle \text{read}, s \rangle &\rightsquigarrow \langle s, s \rangle & \langle \text{write}(v), s \rangle &\rightsquigarrow \langle (), v \rangle \\
\langle T[\text{throw}], s \rangle &\rightsquigarrow \langle \text{throw}, s \rangle & \langle \text{try } \{v\} \text{ catch } \{e\}, s \rangle &\rightsquigarrow \langle v, s \rangle & \langle \text{try } \{\text{throw}\} \text{ catch } \{e\}, s \rangle &\rightsquigarrow \langle e, s \rangle \\
\end{align*}
\]

B.3 DCC with Fixpoints and Pointed Types

\[
\begin{align*}
\text{VAR} & \quad \Gamma(x) = \tau & \Gamma \vdash x : \tau \\
\text{PAIR} & \quad \Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 & \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \\
\text{INL} & \quad \Gamma \vdash e : \tau_1 & \Gamma \vdash \text{inl}(e) : \tau_1 + \tau_2 \\
\text{INR} & \quad \Gamma \vdash e : \tau_2 & \Gamma \vdash \text{inr}(e) : \tau_1 + \tau_2 \\
\text{MATCH} & \quad \Gamma \vdash \text{match } e \text{ with } | \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2 \text{ end} : \tau \\
\text{LAM} & \quad \Gamma, x : \tau_1 \vdash e : \tau_2 & \Gamma \vdash \lambda x : \tau_1. e : \tau_2 \\
\text{APP} & \quad \Gamma \vdash f : \tau_1 \rightarrow \tau_2 & \Gamma \vdash e : \tau_1 & \Gamma \vdash f \cdot e : \tau_2 \\
\text{LABEL} & \quad \Gamma \vdash e : \tau & \Gamma \vdash \text{label}_\ell(e) : L_\ell(\tau) \\
\text{UNLABEL} & \quad \Gamma \vdash e_1 : L_\ell(\tau_1) & \Gamma, x : \tau_1 \vdash e_2 : \tau_2 & \ell \triangleleft \tau_2 & \Gamma \vdash \text{unlabel } e_1 \text{ as } x \text{ in } e_2 : \tau_2 \\
\text{FIX} & \quad \Gamma, f : \tau \vdash e : \tau & \Gamma \vdash \text{fix } f : \tau. e : \tau \\
\text{LIFT} & \quad \Gamma \vdash e : \tau & \Gamma \vdash \text{lift}(e) : \tau_\bot \\
\text{SEQ} & \quad \Gamma \vdash e_1 : \tau_\bot & \Gamma, x : \tau_1 \vdash e_2 : \tau_2 & \Gamma \vdash \text{seq } x = e_1 \text{ in } e_2 : \tau_2 \\
\end{align*}
\]

Protection Rules

\[
\begin{align*}
\ell \subseteq \ell' & \quad \ell \triangleleft \tau & \ell \triangleleft \tau_1 & \ell \triangleleft \tau_2 & \ell \triangleleft \tau_1 \rightarrow \tau_2 & \ell \triangleleft \ell_{\text{PNT}} & \ell \triangleleft \tau \\
\ell \triangleleft L_\ell'(\tau) & \quad \ell \triangleleft \tau_1 \times \tau_2 & \ell \triangleleft \tau_2 & \ell \triangleleft \tau_\bot \\
\end{align*}
\]
Operational Semantics

\[
\begin{align*}
e \rightarrow e' & \quad E[e] \rightarrow E[e'] \\
(\lambda x : \tau . e) \ v & \rightarrow e[x \mapsto v] & \quad \text{proj}_i((v_1, v_2)) & \rightarrow v_i
\end{align*}
\]

match \text{inl}(v) with \mid \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2 \text{ end} \rightarrow e_1[x \mapsto v]

match \text{inr}(v) with \mid \text{inl}(x) \Rightarrow e_1 \mid \text{inr}(y) \Rightarrow e_2 \text{ end} \rightarrow e_2[y \mapsto v]

\text{unlabel}_\ell(v) as x in e \rightarrow e[x \mapsto v]

fix \ f : \tau . e \rightarrow e[f \mapsto \text{fix} f : \tau . e] \quad \text{seq} x = \text{lift}(v) in e \rightarrow e[x \mapsto v]

B.4 DCC with Fixpoints: PC and Type-and-Effect Systems

PC Type System

\[
\begin{align*}
\text{VAR} & \quad \Gamma(x) = \tau & \Gamma \circ \text{pc} \vdash x : \tau \\
\text{UNIT} & \quad \Gamma \circ \text{pc} \vdash () : \text{unit} \\
\text{PAIR} & \quad \Gamma \circ \text{pc} \vdash e_1 : \tau_1 \quad \Gamma \circ \text{pc} \vdash e_2 : \tau_2 \quad \Gamma \circ \text{pc} \vdash (e_1, e_2) : \tau_1 \times \tau_2 \\
\text{INL} & \quad \Gamma \circ \text{pc} \vdash e : \tau_1 \quad \Gamma \circ \text{pc} \vdash \text{inl}(e) : \tau_1 + \tau_2 \\
\text{INR} & \quad \Gamma \circ \text{pc} \vdash e : \tau_2 \quad \Gamma \circ \text{pc} \vdash \text{inr}(e) : \tau_1 + \tau_2 \\
\text{MATCH} & \quad \Gamma, x : \tau_1 \circ \text{pc} \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \circ \text{pc} \vdash e_2 : \tau \quad \Gamma \circ \text{pc} \vdash \text{match} e with | \text{inl}(x) \Rightarrow e_1 | \text{inr}(y) \Rightarrow e_2 \text{ end} : \tau \\
\text{LAM} & \quad \Gamma, x : \tau_1 \circ \text{pc} \vdash e : \tau_2 \quad \Gamma \circ \text{pc} \vdash \lambda x : \tau_1 . e : \tau_1 \circ \text{pc}_1 \rightarrow \tau_2 \\
\text{APP} & \quad \Gamma \circ \text{pc}_1 \vdash f : \tau_1 \circ \text{pc}_2 \rightarrow \tau_2 \quad \Gamma \circ \text{pc}_1 \vdash e : \tau_1 \quad \Gamma \circ \text{pc}_1 \vdash f \ e : \tau_2 \\
\text{LABEL} & \quad \Gamma \circ \text{pc} \vdash \text{label}_\ell(e) : L_\ell(\tau) \\
\text{UNLABEL} & \quad \Gamma \circ \text{pc} \vdash e_1 : L_\ell(\tau_1) \quad \Gamma, x : \tau_1 \circ \text{pc} \sqcup \ell \vdash e_2 : \tau_2 \quad \ell \ll \tau_2 \quad \Gamma \circ \text{pc} \vdash \text{unlabel}_\ell e_1 as x in e_2 : \tau_2 \\
\text{VARIANCE} & \quad \Gamma \circ \text{pc}' \vdash e : \tau \quad \text{pc} \sqsubseteq \text{pc}' \quad \Gamma \circ \text{pc} \vdash e : \tau \\
\text{FIX} & \quad \Gamma, f : \tau \circ \text{pc} \vdash e : \tau \quad \text{pc} \sqsubseteq \text{pc}_\text{PNT} \quad \Gamma \circ \text{pc} \vdash \text{fix} f : \tau . e : \tau
\end{align*}
\]

PC Protection Rules

\[
\begin{align*}
\ell \sqsubseteq \ell' & \quad \ell \ll \tau \quad \ell \ll \tau_1 \quad \ell \ll \tau_2 \quad \ell \ll \tau_1 \times \tau_2 \quad \ell \ll \tau_1 \circ \text{pc} \rightarrow \tau_2
\end{align*}
\]

33
C.1 Type Translations

Here we define the type translations between the pc system and the type-and-effect systems from Sections 3 and 4.
The translation for the language with state and exceptions where \( E = 2^{\{R, W, E\}} \) is as follows.

\[
\begin{align*}
\left[\text{unit}\right] &= \text{unit} \\
\left[\tau_1 + \tau_2\right] &= \left[\tau_1\right] + \left[\tau_2\right] \\
\left[\tau_1 \times \tau_2\right] &= \left[\tau_1\right] \times \left[\tau_2\right] \\
\left[\tau_1 \rightarrow_{pc} \tau_2\right] &= \begin{cases} 
\left[\tau_1\right] \overset{\text{RWE}}{\rightarrow} \left[\tau_2\right] & \text{if } \text{pc} \sqsubseteq \ell_{\text{Exn}} \\
\left[\tau_1\right] \overset{\text{RW}}{\rightarrow} \left[\tau_2\right] & \text{if } \text{pc} \not\sqsubseteq \ell_{\text{Exn}} \text{ but } \text{pc} \sqsubseteq \ell_{\text{State}} \\
\left[\tau_1\right] \overset{R_\tau}{\rightarrow} \left[\tau_2\right] & \text{otherwise}
\end{cases}
\end{align*}
\]

The translation for the potentially nonterminating language where \( E = \{\emptyset, \text{PNT}\} \) is as follows.

\[
\begin{align*}
\left[\text{unit}\right] &= \text{unit} \\
\left[\tau_1 + \tau_2\right] &= \left[\tau_1\right] + \left[\tau_2\right] \\
\left[\tau_1 \times \tau_2\right] &= \left[\tau_1\right] \times \left[\tau_2\right] \\
\left[\tau_1 \rightarrow_{pc} \tau_2\right] &= \begin{cases} 
\left[\tau_1\right] \overset{\text{PNT}}{\rightarrow} \left[\tau_2\right] & \text{if } \text{pc} \sqsubseteq \ell_{\text{PNT}} \\
\left[\tau_1\right] \overset{\emptyset}{\rightarrow} \left[\tau_2\right] & \text{otherwise}
\end{cases}
\end{align*}
\]

\[L_{\ell}(\tau) = L_{\ell}(\left[\tau\right])\]

### C.2 Effectful DCC

In Sections 3 and 4 our translation made use of indexed monads. Here we define the full translation between effectful and pure languages for our examples. For simplicity, we use the syntactic sugar \( \text{let } x = e_1 \text{ in } e_2 \) and \( \text{let } (x, y) = e_1 \text{ in } e_2 \) with their standard meanings.

As the translation for Section 3 is based on an indexed monad [Orchard et al., 2014; Wadler and Thiemann, 1998], we define a type-directed translation making use of \( \eta_\varepsilon, \text{bind}_\varepsilon, \) and \( \text{coerce}_\varepsilon \) operations for our monadic forms. For notational clarity, we use a different but equivalent type for \( \text{bind}_\varepsilon \) than we did in Section 3. Specifically, we use \( \text{bind}_\varepsilon : P(\tau_1) \rightarrow (\tau_1 \rightarrow P(\tau_2)) \rightarrow P(\tau_2) \).

We start by providing a general translation for the fragment of DCC that appears in both Sections 3 and 4, and then discuss the specifics of each translation including their monads and the operations that appear in only one.

All of the translations are type-directed. We use \( [e]_\varepsilon \) as shorthand to denote the translation of a derivation of \( \Gamma \vdash e : \tau \circ \varepsilon \).

### C.3 Common Language Fragment

The type-directed translation is as follows.

\[
\begin{align*}
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \circ \emptyset} &= x \\
\frac{\Gamma \vdash () : \text{unit} \circ \emptyset}{\Gamma \vdash \text{inl } e : \tau_1 \circ \varepsilon} &= \text{bind}_\varepsilon [e]_\varepsilon (\lambda x : \tau_1. \eta_\varepsilon (\text{inl } x))
\end{align*}
\]
C.4 State and Exceptions

For this we first define the η and bind rules for each component of our indexed monad. Note that we omit the trivial monad $P_\emptyset$ because it is the identity.
\[ \eta_E v = \text{label}_{\text{Exn}}(\text{inr } v) \]
\[ \text{bind}_E v f = \text{unlabel } v \text{ as } x \]
\[ \text{in match } x \text{ with} \]
\[ | \text{inl}(_\_\_ ) \Rightarrow \text{label}_{\text{Exn}}(\text{inl }()) \]
\[ | \text{inr}(y) \Rightarrow f y \]
\[ \text{end} \]

Read

\[ \eta_R v = \lambda s : \sigma. v \]
\[ \text{bind}_R v f = \lambda s : \sigma. f (v s) \]

Read and Write  Note that this also applies to W effects, as we use the same monad.

\[ \eta_{RW} v = \lambda s : \sigma. (v, s) \]
\[ \text{bind}_{RW} v f = \lambda s : \sigma. \text{let } (x, s') = v s \text{ in } f x s' \]

Read and Exception

\[ \text{bind}_{RE} v f = \lambda s : \sigma. \text{unlabel } (v s) \text{ as } x \]
\[ \text{in match } x \text{ with} \]
\[ | \text{inl}(_\_\_ ) \Rightarrow \text{label}_{\text{Exn}}(\text{inl }()) \]
\[ | \text{inr}(y) \Rightarrow f y s \]
\[ \text{end} \]

Read, Write, and Exception  Note that this also applies to \{W, E\}, as we use the same monad.

\[ \text{bind}_{RWE} v f = \lambda s : \sigma. \text{let } (x, s') = v s \text{ in } x \text{ as } y \]
\[ \text{in unlabel } x \text{ as } y \]
\[ \eta_{RWE} v = \lambda s : \sigma. (\text{label}_{\text{Exn}}(\text{inr } v), s) \]
\[ \text{in match } y \text{ with} \]
\[ | \text{inl}(_\_\_ ) \Rightarrow (\text{label}_{\text{Exn}}(\text{inl }()), s') \]
\[ | \text{inr}(z) \Rightarrow f z s' \]
\[ \text{end} \]

We now define the various coercions used in our translation. First we note that, for all effects \( \varepsilon \),
\( \text{coerce}_{\emptyset \rightarrow \varepsilon} = \eta_\varepsilon \) and \( \text{coerce}_{\varepsilon \rightarrow \varepsilon} \) is the identity. The rest are defined as follows:

\[ \text{coerce}_{E \rightarrow RW} v = \lambda s : \sigma. (v s, s) \]
\[ \text{coerce}_{R \rightarrow RW} v = \lambda s : \sigma. \text{unlabel } (v s) \]
\[ \text{coerce}_{E \rightarrow RE} v = \lambda s : \sigma. \eta_E (v s) \]
\[ \text{coerce}_{R \rightarrow RE} v = \lambda s : \sigma. (\eta_E (v s), s) \]
\[ \text{coerce}_{E \rightarrow RWE} v = \lambda s : \sigma. v \]
\[ \text{coerce}_{R \rightarrow RWE} v = \lambda s : \sigma. (v, s) \]
\[ \text{coerce}_{RW \rightarrow RWE} v = \lambda s : \sigma. \text{let } (x, s') = v s \text{ in } (\eta_E x, s') \]
\[ \text{coerce}_{RE \rightarrow RWE} v = \lambda s : \sigma. (v s, s) \]
Finally, we provide the translations for the four stateful operations defined in Section 3. We note that try-catch operates differently depending on the effects of the expression in the try block, so we provide several different translations.

\[
\frac{\Gamma \vdash \text{read} : \sigma \cdot R}{\lambda s : \sigma \cdot s}
\]

\[
\frac{\Gamma \vdash e : \sigma \cdot \varepsilon}{\text{bind}_{\varepsilon \cup \mathcal{W}} (\text{coerce}_{\varepsilon \mapsto \varepsilon \cup \mathcal{W}} [e]_{\varepsilon}) (\lambda x : \sigma \cdot \text{coerce}_{\varepsilon \cup \mathcal{W}} (((), x)))}
\]

\[
\frac{\Gamma \vdash \text{throw} : \tau \cdot E}{\text{label}_{\text{Exn}} (\text{inl} (()))}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \cdot E \quad \Gamma \vdash e_2 : \tau \cdot \varepsilon}{\begin{array}{l}
\text{let } (x, s') = [e_1]_{\mathcal{RWE}} s \\
\text{in unlabel } x \text{ as } y \\
\text{in match } y \text{ with } \\
| \text{inl}(\_\text{)} \Rightarrow \text{coerce}_{\varepsilon \mapsto \varepsilon \cup \mathcal{R}} [e_2]_{\varepsilon} s \\
| \text{inr}(v) \Rightarrow \eta_{\varepsilon \cup \mathcal{R}} v s'
\end{array}}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \cdot \mathcal{RWE} \quad \Gamma \vdash e_2 : \tau \cdot \varepsilon}{\begin{array}{l}
\text{let } (x, s') = [e_1]_{\mathcal{RWE}} s \\
\text{in unlabel } x \text{ as } y \\
\text{in match } y \text{ with } \\
| \text{inl}(\_\text{)} \Rightarrow \text{coerce}_{\varepsilon \mapsto \varepsilon \cup \mathcal{RW}} [e_2]_{\varepsilon} s' \\
| \text{inr}(v) \Rightarrow \eta_{\varepsilon \cup \mathcal{RW}} v s'
\end{array}}
\]

C.5 Potential Nontermination

The translation for Section 4 is considerably simpler because it uses only a single nontermination monad. Moreover, the monadic operations are provided directly in the language as follows:

\[\eta v = \text{lift}(v)\quad \text{bind } v f = (\text{seq } x = v \text{ in } f x)\]

The only stateful operation not present in the base language (see Appendix C.3) is fix, which has the following translation:

\[
\frac{\Gamma, f : \tau \vdash e : \tau \cdot \varepsilon}{\text{fix } f : \tau \cdot \varepsilon \cdot \text{coerce}_{\varepsilon \mapsto \varepsilon \cup \mathcal{PNT}} [e]_{\varepsilon}}
\]

Note that, since \(\varepsilon\) is either \(\emptyset\) or \(\mathcal{PNT}\), \(\text{coerce}_{\varepsilon \mapsto \varepsilon \cup \mathcal{PNT}}\) is either \(\eta_{\mathcal{PNT}}\) or the identity, respectively.