Noninterference for Free *

William J. Bowman
Northeastern University, USA
wjb@williamjbowman.com

Amal Ahmed
Northeastern University, USA
amal@ccs.neu.edu

Abstract

The dependency core calculus (DCC) is a framework for studying a variety of dependency analyses (e.g., secure information flow). The key property provided by DCC is noninterference, which guarantees that a low-level observer (attacker) cannot distinguish high-level (protected) computations. The proof of noninterference for DCC suggests a connection to parametricity in System F, which suggests that it should be possible to implement dependency analyses in languages with parametric polymorphism.

We present a translation from DCC into $F_\omega$ and prove that the translation preserves noninterference. To express noninterference in $F_\omega$, we define a notion of observer-sensitive equivalence that makes essential use of both first-order and higher-order polymorphism. Our translation provides insights into DCC’s type system and shows how DCC can be implemented in a polymorphic language without loss of the noninterference (security) guarantees available in DCC. Our contributions include proof techniques that should be valuable when proving other secure compilation or full abstraction results.

Categories and Subject Descriptors  D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics

General Terms  Languages, Security, Theory

Keywords  Noninterference, parametricity, dependency, security, information flow, polymorphism, logical relations, secure compilation, fully abstract compilation.

1. Introduction

The dependency core calculus (DCC) [2] was designed to capture the central notion of dependency that arises in settings like information-flow security, binding-time analysis, program slicing, and function-call tracking. As an example, consider information-flow security analyses which must prevent the publicly visible outputs of a program from revealing information about confidential inputs. Suppose we have a program $e$ with the following security type:

$$e : \textsf{bool}_H \rightarrow \textsf{bool}_L$$

In this type, the label H indicates high-security or private data that should not flow to public portions of the program. The label L indicates low-security or public data. A correct information-flow analysis must guarantee that the low-security output of the function does not depend on the high-security input, which means that $e$ must be a constant function. More generally, we may have programs with both private and public inputs and outputs, such as $e'$ below with the following type:

$$e' : \textsf{bool}_H \times \textsf{bool}_L \rightarrow \textsf{bool}_H \times \textsf{bool}_L$$

In this case, the low-security output may depend on the low-security input but not on the high-security input, while the high-security output may depend on either of the two inputs. In general, labels may be drawn from a lattice, where the lattice order determines illegal dependencies: computation lower in the lattice may not depend on data higher in the lattice. If there are no illegal dependencies, the program is said to satisfy noninterference.

Abadi et al. formalized and proved noninterference for DCC using a denotational semantics based on partial equivalence relations (PERs) indexed by lattice elements. Informally, the relation specifies an observer-sensitive equivalence, which says that data higher in the lattice looks indistinguishable to a lower observer. Abadi et al.’s proof technique suggests a connection to Reynolds’ proof of parametricity [16] using a PER semantics for the polymorphic lambda calculus (also known as System F). Such PER semantics are instances of the logical relations proof method, and many proofs of noninterference—drawing on Reynolds’ concept of parametricity—have made use of logical relations [19]. The connection suggests that it should be possible to use the parametric polymorphism in System F to express the dependency in DCC. Making this connection explicit would be of theoretical as well as practical value. As Tse and Zdancewic [19] point out, a noninterference-preserving translation from DCC to System F would provide a strategy for implementing secure information flow and other dependency analyses expressed in DCC in any language with parametric polymorphism.

Tse and Zdancewic [19] attempted to give a translation from DCC into System F and prove that noninterference in DCC follows as a consequence of parametricity in System F. Unfortunately, they had an error in the proof of a key lemma that says, in essence, that the translation preserves noninterference. Shikuma and Igarashi [17, 18] subsequently gave a counterexample to this lemma. They also gave a noninterference-preserving translation for a language equivalent to a weaker variant of DCC called DCCpc—for DCC with protection contexts—whose type system is more liberal than DCC’s. Moreover, their translation targeted the simply typed $\lambda$-calculus, leaving open the explicit connection between noninterference and parametricity.

We provide a translation from DCC to $F_\omega$, translating noninterference into parametricity. Our translation and proofs make essential use of both first-order and higher-order polymorphism and parametricity. We formalize a notion of observer-sensitive equivalence in $F_\omega$, which relates protected computations from the perspective of some observer, and show our translation preserves noninterference.

* In electronic versions of this paper, we use a blue sans-serif font to typeset our source language and a bold red serif font to typeset the target. The paper will be much easier to read if viewed printed in color.
The proof that our translation preserves noninterference is essentially a full abstraction result. A translation is fully abstract if it preserves and reflects contextual equivalence—i.e., if two source terms cannot be distinguished by source-level contexts if and only if their translations cannot be distinguished by target-level contexts. We show preservation of observer-sensitive equivalence, but for the highest observer—e.g., one that can observe both high- and low-security outputs—observer-sensitive equivalence corresponds to the natural contextual equivalence one would get after erasing security features from the language. As Abadi [1] and Kennedy [11] point out, fully abstract compilation is vital for secure compilation of high-level languages.

Contributions The main contributions of our work are:

• the development of a translation from the recursion-free fragment of DCC into $F_w$ ([4]) and proof of its correctness ([6]);

• the development of an open logical relation for $F_w$ ([5.2]);

• the formalization of an observer-sensitive equivalence for $F_w$ terms of translation type using relational parametricity ([5.3]);

• the back-translation from $F_w$ terms of translation type $s^+$ to DCC terms of type $s$ ([1, 1]) and a novel logical relation to prove that the back-translation is well-founded; and

• proofs that our translation preserves and reflects observer-sensitive equivalence ([1.2]).

Our translation is novel and sheds light on the type system for DCC and shows how DCC can be implemented in a polymorphic language without loss of security/noninterference guarantees. Proving a translation fully abstract is particularly difficult when the target language is more expressive than the source language, as is the case in this work. This paper provides new proof techniques that should be useful for establishing other full abstraction results.

We have elided most proofs and parts of some definitions from the paper. Detailed proofs and complete definitions may be found in the online technical appendix [7].

2. Dependency Core Calculus (DCC)

DCC is a call-by-name, simply-typed $\lambda$-calculus, based on the computational lambda calculus of Moggi [13]. DCC incorporates multiple monads, one for every level of a predetermined information lattice, used to restrict dependencies in the program. The lattice order captures permissible dependencies: computation interpreted in a monad higher in the lattice can depend on data interpreted in a monad lower in the lattice, but not vice versa. In effect, data higher in the lattice is held abstract with respect to computation lower in the lattice. DCC uses this lattice of monads and a nonstandard typing rule for their associated bind operation to describe the dependency of computations in a program.

We assume a lattice $L$ with a set of labels $L^1$ and an ordering on labels $\preceq$. We write $\ell \preceq \ell'$ to mean $\ell'$ is at least as high as $\ell$. The monadic type $T_\ell$, and the return $\eta_\ell$ and bind operations are indexed by a label $\ell \in L^1$.

Except for the monadic operations, the language is completely standard. Figure 1 defines the syntax and semantics of DCC. We define a small-step operational semantics $(e \rightarrow e')$ using evaluation contexts $E$ to lift the primitive reductions to a call-by-name semantics for the language. We write $e \rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$. Note that in this call-by-name language, $\eta_\ell e$ is a value form. The operation $\mathsf{bind} x = e_1 e_2$ evaluates $e_1$ to a value $\eta_\ell e$ and then substitutes $e$ for $x$ in $e_2$.

DCC typing judgments have the form $\Gamma \vdash e : s$ where the environment $\Gamma$ tracks the set of free term variables in scope, along

\[ e \rightarrow^* e' \]

\[ (\lambda x : s. e_1) e_2 \rightarrow e_1 e_2 / x \]

\[ \mathsf{bind} x = \eta_\ell e_1 in e_2 \rightarrow e_1 e_2 / x \]

Figure 1. DCC: Syntax + Dynamic & Static Semantics (excerpts)

with their types. Typing rules for all constructs except $\eta_\ell$ and bind are completely standard so we omit them here. The return operation $\eta_\ell e$ protects the term $e$ by wrapping it with the label $\ell$. These protected terms can only be unwrapped through a bind operation $\mathsf{bind} x = e_1 in e_2$. While $e_2$ may depend on the protected term inside $e_1$, the results produced by the entire bind $\mathsf{bind} (x : s)$ must be protected at the label $\ell$ or higher. This requirement is captured by the judgment $\ell \preceq s$ (pronounced “$s$ is protected at $\ell$”).

Informally, a type $s$ is protected at $\ell$ if expressions of this type do not leak information to levels lower than (or incomparable to) $\ell$. Since there is only one value of type $1$, this type cannot communicate any information, so it is protected at any $\ell$. Pairs are protected if both components of the pair are protected. Functions are protected if they produce protected results; the input to a function does not matter. Finally, there are two cases for $\ell \leq s$.

We examine why the translation given by Tse and Zdancewic fails and shows how DCC can be implemented in a polymorphic language.

3. Background and Main Ideas

We examine why the translation given by Tse and Zdancewic fails to preserve noninterference and describe the key ideas behind our translation. Below, we write $s^+$ to denote the translation of the DCC type $s$.

Preserving noninterference Tse and Zdancewic translate the monadic type as follows:

$$(T_\ell s)^+ = \alpha \rightarrow s^+$$

The translation of all other types is defined by structural recursion.

Note that in $L^1$, $\ell$ is part of the name for the set, not a meta-variable.
As mentioned in \[1\] the key lemma we must prove about our translation is that it preserves DCC’s noninterference guarantee. Let us take a closer look at how noninterference is expressed in DCC and how this type translation captures noninterference at the target level.

Consider the type \( T_\text{bool} \). In DCC, this type represents a boolean value that is visible only to \( H \) computations; \( L \) computations must treat such values as opaque and hence cannot distinguish between \text{true} and \text{false} at the type \( T_\text{bool} \). DCC formalizes this situation using an \textit{observer-indexed logical relation} that says these two values are \textit{equivalent} at \( L \) (written \( \eta_L \text{true} \approx_L \eta_L \text{false} : T_\text{bool} \)) but not at \( H \). Hence, the relation \( \approx_L \) relates every possible pair of boolean expressions at the type \( T_\text{bool} \), while the relation \( \approx_H \) at the same type only relates boolean expressions that evaluate to the same value. We will refer to this relation as \textit{observer-sensitive equivalence}:

\[

\text{we write } e_1 \approx_L e_2 : s \text{ to mean that terms } e_1 \text{ and } e_2 \text{ of type } s \text{ are equivalent from the perspective of an observer at level } \zeta \text{ in the lattice}\]

Intuitively, we must show that if \( e_1 \approx_L e_2 : s \) and \( e_1 \) and \( e_2 \) translate to target terms \( m_1 \) and \( m_2 \), respectively, then \( m_1 \approx_L m_2 : s^+ \). The key is to formalize the target-level \( \approx_L \) relation in terms of the standard logical relation for System F (or \( F_\omega \)), which is not indexed by an observer. To see how Tse and Zdancewic do this, consider the type \( T_\text{bool} \) again; they translate to the target type \( \alpha_H \rightarrow \text{bool} \). This translation uses abstract types \( \alpha_H \) to encode each element \( \ell \) in the source-level lattice. This simulates DCC’s observer-sensitive equivalence by requiring, in essence, that an \textit{L}-observer not have access to any terms of type \( \alpha_H \), which means that any function of type \( \alpha_H \rightarrow \text{bool} \) can never be applied. Hence, the two functions \( \lambda x. \alpha_H \text{true} \) and \( \lambda x. \alpha_H \text{false} \) are indistinguishable. Meanwhile, an \textit{H}-observer is given access to terms of type \( \alpha_H \) and can use these as keys to gain access to the computation hidden inside functions of type \( \alpha_H \rightarrow \text{bool} \).

This type translation is simple and promising, but it does not preserve observer-sensitive equivalence.

\textbf{Counterexample to Tse-Zdancewic’s key lemma} To see why the above translation fails to preserve observer-sensitive equivalence, consider the counterexample given by Shikuma and Igarashi \[13\]. DCC terms of the protected function type \( s_1 = T_\ell \left((T_\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool}\right) \) must be equivalent to \( \eta_L (\lambda x. T_\text{bool} \text{true}) \), where \( \nu \) is either \text{true} or \text{false}. In particular, Shikuma and Igarashi point out that the following term is ill-typed due to the \( \ell \leq \zeta \) restriction in the typing rule for \textit{bind} (since \( \ell \neq \zeta \)).

\[
e_1 = \eta_L (\lambda x. T_\text{bool}. \text{bind} y = x \text{ in } y) \quad // \text{ill typed!}
\]

Thus, the following two terms \( e_1 \) and \( e_2 \) are equivalent at type \( s = s_1 \rightarrow T_\text{bool} \) for an observer at level \( \ell \), since the only functions we can pass in for \( \ell \) are the constant functions above—i.e., we cannot pass in non-constant functions such as \( e_1 \) (since they are not well-typed).

\[
e_1 = \lambda f : s_1. \text{bind } f' = f \text{ in } \eta_L (f' (\eta_L \text{true}))
\]

\[
e_2 = \lambda f : s_1. \text{bind } f' = f \text{ in } \eta_L (f' (\eta_L \text{false}))
\]

While \( e_1 \) and \( e_2 \) are equivalent at type \( s \) at level \( \ell \) in DCC, their translations are \textit{not} equivalent at the following type \( s^+ \) in System F:

\[
s^+ = (\alpha_\ell \rightarrow ((\alpha_\ell \rightarrow \text{bool} \rightarrow \text{bool})) \rightarrow (\alpha_\ell \rightarrow \text{bool})
\]

The term \( m_2 \) defined below, which corresponds to \( e_2 \), can distinguish the translations of \( e_1 \) and \( e_2 \):

\[
m_2 = \lambda k : \alpha_\ell. \lambda y : \alpha_\ell \rightarrow \text{bool}. y \ k
\]

Hence, we have two terms of type \( s \) that are equivalent (at \( \ell \)) in DCC, but their translations are not equivalent at the type \( s^+ \) (at \( \ell \)) in System F, which means that the translation fails to preserve (observer-sensitive) equivalence.

How can we fix this problem? At a minimum, since the typing rule for \textit{bind} prevents us from concluding that \( e_1 : s_1 \), we should not be able to conclude that the corresponding (behaviorally equivalent) target term \( m_2 \) is well-typed at \( s^+ \). In more detail, consider the restrictions on the continuation inside \( e_2 \) that uses \( y : T_\ell \text{bool} \)—i.e., that the \textit{bind} expression inside \( e_2 \) must have a result type that is protected at \( \ell \). In contrast, there is no such restriction on the body of \( m_2 \) that uses \( y : (T_\ell \text{bool})^+ \). The problem is that this simple encoding of the monadic type as a function fails to capture all of the restrictions imposed by the typing rule for \textit{bind}. We need to find an alternative translation for types of the form \( T_\ell \text{bool} \) that captures these restrictions so that for every \( e : T_\ell \text{bool} \), all target-level uses of the translation of \( e \) must satisfy the same constraints as the ones imposed by the \textit{bind} typing rule on source-level uses of \( e \).

Our translation is more involved than this simple type translation and requires higher-order polymorphism, but the way in which we leverage higher-order relational parametricity when defining \( \approx_L \) at the target level (\S 5.3) is semantically pleasing and insightful given the semantics of DCC. We define \( \approx_L \) at the target-level by instantiating a novel \textit{open} logical relation for \( F_\omega \), which interprets types as relations on open terms rather than closed terms as is standard. In \S 5.2 we explain why we need this. Our open logical relation is inspired by Zhao \textit{et al.’s} open logical relation for a linear System F \[22\].

\textbf{Need for back-translation} As with all proofs of full abstraction, a key step of our proof requires showing that for any target term of translation type, \( m : s^+ \), there exists a semantically equivalent source term \( e : s \). We formalize this via a “back-translation” relation between target terms of type \( s^+ \) and source terms of type \( s \) (\S7.1). The need for back-translation arises when proving that if two source functions \( f_1 \) and \( f_2 \) are equivalent then their translations \( t_1 \) and \( t_2 \) are equivalent. To show the latter, we must assume that we’re given two equivalent target-level arguments \( m_1 \) and \( m_2 \) and show that \( t_1 m_1 \) is equivalent to \( t_2 m_2 \). The only way to proceed is by making use of the equivalence of the source functions \( f_1 \) and \( f_2 \), but they can only be applied to source-level inputs. If we could back-translate \( m_1 \) and \( m_2 \) to source terms \( e_1 \) and \( e_2 \)—which is exactly the failure the counterexample exploits—then we could conclude that \( f_1 e_1 \) is equivalent to \( f_2 e_2 \), which implies that \( f_1 m_1 \) is equivalent to \( f_2 m_2 \), since each of those source terms is semantically equivalent to the corresponding target terms.

Our back-translation technique handles more complex languages compared to the “inverse translation” given by Shikuma and Igarashi \[17\][18]. Note that their source and target languages are both simply-typed and, thus, in closer correspondence. This simplifies the back-translation. By contrast, our target language is more expressive than the source (\textit{e.g.}, \( F_\omega \) can encode natural numbers, while DCC cannot). Back-translation in this setting is more complicated. We give a more detailed comparison in \[8\].

\textbf{Our translation: key ideas} The idea for our type translation can be explained by analogy with existential types and their well-known encoding using universal types. As a starting point, notice that the \textit{bind} typing rule resembles the typing rule for unpacking a term of existential type:

\[

\Delta ; \Gamma \vdash e_1 : \exists \alpha. \tau \quad \Delta, \alpha ; \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \alpha \notin \text{ftv}(\tau_2) \quad \Delta, \Gamma \vdash \text{unpack } \alpha, x = e_1 : e_2 : \tau_2
\]

Note that, just as the \textit{bind} rule requires the side condition \( \ell \leq \zeta \), the unpack rule requires the side condition \( \alpha \notin \text{ftv}(\tau_2) \). This idea inspires our translation, because a simple encoding captures this side condition through parametricity.

\footnote{Following convention, we represent the level of the observer using the meta-variable \( \zeta \) rather than \( \ell \).}
Recall that the encoding of existential types using universal types captures all of the restrictions imposed by the unpack typing rule:

\[ \exists \alpha. \tau \equiv \forall \beta. (\alpha \land \tau \to \beta) \to \beta \]

The above encoding says that an existential package is a data value that, given a result type \( \tau_2 \) and a continuation, calls the continuation to yield a final result. The continuation corresponds to the body of an unpack: it takes a type \( \alpha \) and a value of type \( \tau \), and uses them to compute a result of type \( \tau_2 \). In particular, note that since \( \alpha \) is not in scope when we instantiate \( \beta \), the above encoding perfectly captures the \( \alpha \in \text{fv}(\tau_2) \) requirement from the unpack typing rule since we cannot instantiate \( \beta \) with any \( \tau_2 \) with a free \( \alpha \).

We can analogously capture the constraints in the bind typing rule by encoding monadic types roughly as follows:

\[ (T \ell \rho)^+ \text{ roughly } \forall \beta. ([\ell \leq \beta] \times (s^+ \to \beta)) \to \beta \]

This encoding says that a protected computation is a data value that, given a result type \( \tau_2 \), a proof that \( \tau_2 \) is protected at \( \ell \)—which we informally write as \([\ell \leq \tau_2]\) for now and formalize below—and a continuation, calls the continuation to yield a final result. The continuation here corresponds to the body of a bind: it takes a computation of type \( s^+ \) and uses it to compute a result of type \( \tau_2 \). In particular, note that we must provide a proof that the result type of the continuation is protected at \( \ell \), only then will this data value call the continuation to compute a result.

The remaining question then is how to encode the type \([\ell \leq \tau_2]\). Note that we should only be able to construct a term of this kind, by the protection type \([\ell \leq \tau_2]\) if \( \exists \alpha. \tau_2 = s^+ \) and \( \ell \leq s_2 \). First, like Tse-Zdancewic, we shall use type variables \( \alpha_t \) to encode each \( \ell \in L_\ell \). Then, our desired “protection type” can be built using an abstract type constructor \( \alpha :: \to \to \to \) applied to the types \( \alpha_t \) and \( \tau_2 \). That is, we shall represent \([\ell \leq \tau_2]\) with the protection type \( \alpha :: \to \to \to \to \to \to \). Of course, in order to introducing the higher-ranked abstract type \( \alpha :: \to \to \to \to \to \to \) and a set of type variables \( \alpha_t : \to \to \to \to \to \to \) for each \( \ell \in L_\ell \), we must also provide an interface for constructing terms of protection type—e.g., given terms of type \( \alpha :: \to \to \to \to \to \to \) and \( \alpha :: \to \to \to \to \to \to \), we should be able to construct a term of type \( \alpha :: \to \to \to \to \to \to \). The types of these proof constructors mirror the protection rules in DCC.

Thus, our translation uses higher-order polymorphism to encode DCC’s protection judgment at the target-level. To summarize, we translate monadic types as follows:

\[ (T \ell s)^+ = \forall \beta :: \to \to \to , ([\alpha :: \to \to \to \to ] \times (s^+ \to \beta)) \to \beta \]

We describe the details of the translation in the next section.

4. Translating DCC to F_\omega

In this section, we begin by presenting the target language \( F_\omega \) and then give a type-directed translation from DCC to \( F_\omega \).

Target language: \( F_\omega \). The target language \( F_\omega \) is the call-by-name, higher-order polymorphic lambda calculus with unit, pairs, and sums. Figure 3 presents the syntax and excerpts of the static semantics. We omit much of the formal presentation as the language is completely standard (e.g., see Pierce [13]).

Typing judgments in \( F_\omega \) have the form \( \Delta ; \Gamma \vdash m : t \), which says that an \( F_\omega \) term \( m \) has type \( t \) under type environment \( \Delta \) and term environment \( \Gamma \). The kinding judgment \( \Delta \vdash t :: \kappa \) says that a type \( t \) has kind \( \kappa \) under type environment \( \Delta \), where \( \Delta \) maps abstract types \( \alpha \) to kinds \( \kappa \). Since we have type-level functions, i.e., type constructors, we define type equivalence \( t_1 \equiv t_2 \) to account for beta-reduction at the type-level.

Translation. We start by defining a translation from DCC types to \( F_\omega \) types, shown in Figure 3. We write \( s^+ \) to mean the translation of the DCC type \( s \). Most types are translated by structural recursion. As discussed above, the monadic type \( T \ell \rho \) is translated to the type \( \kappa \to \kappa \to \kappa \to \kappa \).
to this fully applied type \((\alpha \times \alpha \times s^+)\) as a protection type. Since the type constructor is abstract, the only terms that can inhabit a protection type are terms built using the provided proof constructors.

Figure 4 shows \(\preceq\) which contains the constructors for terms of protection types (i.e., the proof constructors). In essence, these constructors encode the inference rules of the \(\preceq\) judgment. Each constructor is named suggesting the rule by which \(\preceq\) is inferred. For instance, \(p_1\) encodes the rule for \(\preceq 1\), that any label is protected at the unit type.

During term translation, we need to construct terms that inhabit a protection type (i.e., protection proofs). We provide a function \(pf[\ell \preceq s]\) for constructing proof terms by induction on a given derivation of \(\ell \preceq s\). Note that \(pf[\ell \preceq s]\) yields the following lemma.

**Lemma 4.1**

If \(\ell \preceq s\) then \(3m\). \(\mathcal{L}_1^\ell \cap \mathcal{L}_2^s \subseteq \preceq + \vdash m : (\alpha \times \alpha \times s^+)\)

That is, if a source type is protected at some label, then a protection proof exists, namely \(pf[\ell \preceq s]\), for the translated type and label under \(\mathcal{L}_1^\ell \cap \mathcal{L}_2^s \subseteq \preceq\) (which we refer to as the protection ADT).

The translation judgment \(\Gamma \vdash e : s \rightarrow m\) takes an open source term \(e\) of type \(s\) and produces the target term \(m\). The term \(m\) has type \(s^+\) under the type environment \(\mathcal{L}_1^\ell \cap \mathcal{L}_2^s \subseteq \preceq\) and \(\preceq\). We write \(\Gamma \vdash\) to mean the point-wise translation of \(s : \in \Gamma \times s' : s^+\).

Since DCC and \(F_\omega\) share the same basic constructs, we translate most terms by structural recursion. The translation of an \(\eta\) value expects a continuation and a proof construction, so the translation of \(\eta\) must produce such a proof and continuation. Several translation rules are presented in Figure 5.

**Figure 5.** DCC to \(F_\omega\): Term Translation (excerpts)

\[
\begin{align*}
\Gamma \vdash e : s \rightarrow m & \text{ where } \mathcal{L}_1^\ell \cap \mathcal{L}_2^s \subseteq \preceq, \Gamma \vdash m : s^+ \\
(x : \in \Gamma) & \vdash (x : s) \in \Gamma \\
\Gamma \vdash ⟨\cdot⟩ : (≤ 1) & \vdash (x : s) \in \Gamma \\
\Gamma \vdash λx : s. e : s \rightarrow m & \vdash λx : s. e : s \rightarrow m \\
\Gamma \vdash e_1 : s_1 \rightarrow s_2 \rightarrow m_1 & \vdash e_1 : s_1 \rightarrow s_2 \rightarrow m_1 \\
\Gamma \vdash e_2 : s_2 \rightarrow m_2 & \vdash e_2 : s_2 \rightarrow m_2 \\
\end{align*}
\]

The logical relation is defined by structural recursion on types. The value relation \(\mathcal{V}[s][\cdot]\) relates closed values at type \(s\). We use the relation \(\text{Atom}[s]\) to ensure that the logical relation relates only well-typed terms.

**Lemma 4.2** (Translation preserves well-typedness)

If \(\Gamma \vdash e : s\) then \(\Gamma \vdash e : s \approx m\) and \(\mathcal{L}_1^\ell \cap \mathcal{L}_2^s \subseteq \preceq, \Gamma \vdash m : s^+\).

5. Observer-Sensitive Equivalence

In this section, we define a notion of observer-sensitive equivalence for DCC that is formalized using a logical relation. This logical relation is essentially the same as the one defined by Tse and Zdancewic [19]. We also define a novel open logical relation for \(F_\omega\) and then formalize an observer-sensitive equivalence for \(F_\omega\) using higher-order parametricity.

5.1 Logical Relation for DCC

The logical relation for DCC, written \(\Gamma \vdash e_1 \approx e_2 : s\), says that \(e_1\) and \(e_2\) appear equivalent from the perspective of an observer at level \(\zeta\). The relation is defined in Figure 6 and enforces that an observer whose label in the lattice is \(\zeta\) cannot distinguish data protected at a lattice level higher than (or incomparable to) \(\zeta\).

The logical relation is defined by structural recursion on types. The value relation \(\mathcal{V}[s][\cdot]\) relates closed values at type \(s\). We use the relation \(\text{Atom}[s]\) to ensure that the logical relation relates only well-typed terms.

The value \((\cdot)\) appears equivalent to itself at type \(1\) to an observer at any level. Summing \(\eta_1 e_1\) and \(\eta_2 e_2\) appear equivalent at type \(s + s'\) to an observer at level \(\zeta\) if \(e_1\) and \(e_2\) appear equivalent at type \(s\) to the observer, and similarly for the second injections at type \(s'\). Pairs are related if their components are related at their respective types. Functions are related if, given inputs related at the argument type, they produce results related at the result type. The values \(\eta_1 e_1\) and \(\eta_2 e_2\) are related at type \(T\) if the observer \(\zeta\) is lower than \(\ell\), or if \(e_1\) and \(e_2\) appear equivalent to the observer at type \(s\). This captures the idea that an observer at a level lower than \(\ell\) is unable to distinguish \(\eta_1 e_1\) from \(\eta_2 e_2\).

The relation \(\mathcal{V}[s][\cdot]\) relates closed terms if they reduce to related values. We extend the logical relation to open terms \(e_1\) and \(e_2\) by picking substitutions \(\gamma_1\) and \(\gamma_2\) that map variables to terms related at the corresponding types in \(\Gamma\), and requiring that the closed terms \(\gamma_1(e_1)\) and \(\gamma_2(e_2)\) be related.

The fundamental property of this logical relation is noninterference, formally stated in Theorem 5.2. The key property enforced by the logical relation is that any two terms whose type is protected at level \(\ell\) are related if the observer \(\zeta\) is lower than or incomparable to \(\ell\). This property, stated in Lemma 5.1, is similar to Abadi et al.'s Proposition 3.2 [2].
Lemma 5.1
If \( \ell \leq s \) and \( \ell \not\models \zeta \) then \( \forall (e_1, e_2) \in \text{Atom}[s], (e_1, e_2) \in \mathcal{E}[s] \).

Theorem 5.2 (Noninterference)
If \( \Gamma \vdash e : s \) then \( \forall \Gamma \vdash e \equiv s : s \).

We prove Theorem 5.2 directly by induction on the typing judgment, which requires Lemma 5.1 in the bind case. In §7.2 we will also show that noninterference follows as a consequence of our translation and parametricity in the target language.

5.2 An Open Logical Relation for \( F_\omega \)

We formalize equivalence in \( F_\omega \) via an open logical relation whose structure resembles that of the logical relation for \( R_\omega \) given by Vytiniotis and Weirich [20]. The main difference is that we interpret types as relations on open terms with open types—instead of closed terms of closed type, as is standard practice—following ideas from Zhao et al.’s [24] open logical relation for a linear System F. Specifically, our value relation \( \mathcal{V} \) and term relation \( \mathcal{E} \) for \( F_\omega \) may relate open terms with open types, unlike the \( \mathcal{V} \) and \( \mathcal{E} \) relations for \( \text{DCC} \) which relate closed terms of closed type.

We need an open logical relation due to our type translation. Recall from §3 that when proving that equivalence of functions is preserved, we have two arguments \( m_1 \) and \( m_2 \) related by the \( \text{F}_\omega \) term relation \( \mathcal{E} \). We must show that we can back-translate these to two related source terms \( e_1 \) and \( e_2 \). In order to back-translate \( m_1 \) and \( m_2 \), we must know they have a translation type, since only terms of translation type can be back-translation (see §7.4). However, our translation produces types that contain free variables \( \alpha_1 \) and \( \alpha_1 \) with terms of the free variables from \( \leq s \). If our \( \text{F}_\omega \) relation closed these free variables, as is standard, then terms that belong to the relation \( \mathcal{E} \) will not have translation type, and we will not be able to back-translate them.

Figure 7 presents the open logical relation for \( F_\omega \). As usual, the top-level logical relation \( \Delta ; \Gamma \vdash t_1 \equiv t_2 : \tau \) requires that we close all the free variables of \( \Delta \) and \( \Gamma \) in \( t_1 \) and \( t_2 \) choosing a relational type interpretation \( \rho \) and related term substitutions \( \gamma_1 \) and \( \gamma_2 \). However, in contrast to a standard logical relation, such as our logical relation for \( \text{DCC} \), these substitutions contain terms that may be open with respect to a fresh type environment \( \Delta \) and term environment \( G \). By fresh, we mean the domains of the environments \( \Delta \) and \( G \) are disjoint from \( \Delta \) and \( G \) written \( \text{dom}(\Delta) \otimes \text{dom}(G) \) and \( \text{dom}(G) \otimes \text{dom}(\Gamma) \). In effect, this allows us to extend the logical relation with new constants and leverage these when we provide relational interpretations for the original type variables. In §8 we show how we take advantage of this extra expressive power to extend the logical relation with constants for our protection ADT and relational interpretations for the type variables in \( \mathcal{E}_\omega \).

We define a value relation \( \mathcal{V} \) and a term relation \( \mathcal{E} \) which are inhabited by open terms that are well-typed under \( \text{DCC} \). The relations are indexed by a type \( t \) such that \( \Delta ; t : \kappa \). Hence, the relations need to be parameterized by a relational interpretation \( \rho \) which maps the free type variables of \( \alpha_i \) in \( \Delta \) to triples \( (t_1, t_2, \rho) \) (abbreviated \( \pi \)). The types \( t_1 \) and \( t_2 \) must be well-formed under \( \text{DCC} \) rather than \( \Delta \), and are the types we substitute for \( \alpha_i \) in pairs of related terms. We write \( \pi_1 \) and \( \pi_2 \) to denote the projections of \( t_1 \) and \( t_2 \) from \( \pi_1 \) and \( \pi_2 \) to denote the projection of \( \rho \) from \( \pi_1 \). We extend this notation to \( \rho \) if \( \rho = [\alpha_1 \vdash \kappa_1 \vdash \pi_1] \cdots [\alpha_n \vdash \kappa_n \vdash \pi_n] \) then \( \rho_1 = [\alpha_1 \vdash \pi_1] \cdots [\alpha_n \vdash \pi_n] \) and we analogously use \( \rho_2 \) and \( \rho_3 \).

Finally, we write \( \rho(e) \) to denote applying the substitution \( \rho \) to all the type variables in \( e \). We use the notation for application of other substitutions to terms and types.

The relation \( \mathcal{E}[t]_{\rho}^{\mathcal{DCC}} \) runs terms until they are irreducible, and then requires that the irreducible terms be related in \( \mathcal{V}[t]_{\rho}^{\mathcal{DCC}} \). Note that unless \( t \equiv \alpha \), the terms must reduce to values of the appropriate canonical form for type \( t \). If \( t \equiv \alpha \), then \( \rho_1 \) is a relation on terms with free (term and type) variables and \( \mathcal{V}[t]_{\rho_1}^{\mathcal{DCC}} \) relates terms that are not values. This follows the formalism by Zhao et al. [24].

As usual, when relating the values \( \Lambda : \alpha \otimes \kappa_1 \) and \( \Lambda : \alpha \otimes \kappa_2 \) in \( \mathcal{V}[\alpha : \kappa_1, t]_{\rho}^{\mathcal{DCC}} \), we consider arbitrary types \( t_1 \) and \( t_2 \) and a relation object \( R \). At kind \( s \), relation objects are just sets of terms. At kind \( \pi \), relation objects are relation-level functions. Intuitively, relation-level functions take relations as inputs and produce relations as outputs. Formally, relation-level functions written \( \Lambda \text{R} \text{R} \) take triples \( \pi \) and produce relation objects. We write \( \text{R}_\pi \) as the application of the relation-level function \( \text{R} \) to \( \pi \).

The relation objects for all types, including the higher-kinded types, are defined inductively on the judgment \( \Delta ; t : \kappa \) by \( \mathcal{T}[t]_{\rho}^{\mathcal{DCC}} \). When \( t \) has kind \( s \), this is just the relation \( \mathcal{V}[t]_{\rho}^{\mathcal{DCC}} \).
∀\ell \quad \text{let } \rho = \{ \ell, \exists \gamma \text{ s.t. } \gamma \in \mathcal{L}_S^+ \text{ and } \gamma \preceq \ell \}. \\
\left( \rho_1(\gamma_1(\gamma_1(y_1(m_1)))), \rho_2(\gamma_2(\gamma_2(y_2(m_2)))) \right) \in \mathcal{E}[s+]_\rho
\[ \eta^k_\mathbf{s} \mathrel{\overset{\text{def}}{=}} \lambda \mathbf{y}. s^+ \cdot \lambda \mathbf{x}. ((\alpha \lessdot \alpha_i \beta) \times (s^+ \rightarrow \beta)). \]

\[ \text{Atom}^+ [s]_\mathbf{s} \mathrel{=} \{ (e, m) \mid e \vdash s : \Sigma \Sigma_D \vdash \delta (s^+) \wedge (\varepsilon (e) [s], m [s] / x) \in E^+ [s]_\mathbf{s} \} \]

\[ \forall e' : m : e / \mathbf{s} \vdash \delta (e) [\mathbf{s}] \wedge \forall e' : m' : (e', m') \in E^+ [s]_\mathbf{s} \]

\[ \forall e' : m : e / \mathbf{s} \vdash \delta (e) [\mathbf{s}] \wedge \forall e' : m' : (e', m') \in E^+ [s]_\mathbf{s} \]

\[ \forall e' : m : e / \mathbf{s} \vdash \delta (e) [\mathbf{s}] \wedge \forall e' : m' : (e', m') \in E^+ [s]_\mathbf{s} \]

\[ \Gamma \vdash e : m : s \mathrel{\overset{\text{def}}{=}} \Gamma \vdash e : s \wedge \Sigma_D \Sigma_G \Gamma \vdash m : \delta (s^+) \wedge \forall e, y : m : e / \mathbf{s} \wedge \delta (\varepsilon (e) [s]) \in E^+ [s]_\mathbf{s} \]

\[ \Gamma \vdash e : m : s \mathrel{\overset{\text{def}}{=}} \Gamma \vdash e : s \wedge \Sigma_D \Sigma_G \Gamma \vdash m : \delta (s^+) \wedge \forall e, y : m : e / \mathbf{s} \wedge \delta (\varepsilon (e) [s]) \in E^+ [s]_\mathbf{s} \]

To prove the translation is correct, we’ll need the following two lemmas. Lemma 6.1 is a free theorem, called the Parametricity Condition by Wadler [21]. Intuitively it states that passing two composed continuations to a function is the same as passing one, and then applying the other to the result.

**Lemma 6.1 (Free theorem: parametricity condition)**

If \( D \), \( \alpha : \ast \vdash \rho \alpha (t) : \ast \), \( D \vdash \rho \alpha (t) : \ast \), \( D \vdash \rho \beta (t) : \ast \), \( D \vdash \rho \beta (t) : \ast \), \( D \vdash m_1 : t \rightarrow t_1 \), \( D \vdash m_2 : t_2 \rightarrow t_3 \), \( (m_1, m_2) \in E^+ [t]_D G \]

\[ \text{then } (m_1 (m [t] \circ m_2, m_2 (m_1 m_2 \circ m_3))) \in E^+ [t]_D G \]

**Lemma 6.2 (Cross-language relation respects \( \mathcal{F}_e \) relation)**

Let \( \rho = [\ell]_\mathcal{F}_e \). If \( (u, u_2) \in E^+ [s]_\mathbf{s} \) then \( (u, u_2) \in E^+ [s]_\mathbf{s} \)

We prove Theorem 6.3, which says that if a source term is well-typed then it must be related to its translation in the cross-language logical relation. The standard notion of adequacy—given a closed boolean expression \( e \), its translation \( m \) runs to the same boolean value as \( e \)—follows from this theorem. The proof of Theorem 6.3 is by induction on \( \Gamma \vdash e : s \rightarrow m \). The case for \( \text{bind} \) is the most interesting. We sketch this case by noting a series of equivalences. We decompose the translation of the \( \text{bind} \) continuation \( (\lambda x : s^+ m_2) \) in the proof below) into the \( \eta^k_\mathbf{s} \) continuation and the continuation \( m_2 \) defined below. Then we use the parametricity condition and the definition of \( E^+ [t]_D \) to find the protected target value \( m' \) inside \( m_1 \).

**Theorem 6.3 (Translation preserves semantics)**

If \( \Gamma \vdash e : s \) and \( \Gamma \vdash e : s \rightarrow m \) then \( \Gamma \vdash e : m : s \)

**Proof Sketch:**

Show \( \text{bind} x = e_1 \in e_2 : s_2 \) is related to \( m_1 (s_2^+) \approx (\lambda x : s_2^+ \cdot m_2) \).

Let \( x' \approx (p \cdot \ell \leq s_2) \).

\[ m_1 (s_2^+) \approx (\lambda x : s_2^+ \cdot m_2) \rightarrow (\lambda x : s_2^+ \cdot m_2) \]

Suffices to show:

\[ \text{bind} x = e_1 \in e_2 : s_2 \]

Follows by IH on \( e_1 \) and \( m_1 \)

\[ m_2 (m_1 [s] / x) \]

by evaluation

\[ m_2 (m_1 [s] / x) \approx m_2 (m_1 [s] / x) \]

by IH on \( e_1 \) and \( m_1 \)

**7. Translation Preserves Noninterference**

In this section, we present our central result, that our translation from DCC to \( \mathcal{F}_e \) preserves noninterference. The proof requires a back-translation. Our back-translation is inspired by Ahmed and Blum’s [5], but contains several novel features as discussed below. We first present the back-translation and then prove that our translation preserves observer-sensitive equivalence.

Below we write \( s^+ \) to denote \( s^+ \) with all instances of \( \alpha \) and \( \alpha_i \) replaced with \( \alpha_i \) and \( \alpha_i \), respectively. We analogously put the hat symbol `\` on other notation that we have defined already, such as \( t^+ \rightarrow t^+ \), to indicate substitution of type and term variables in \( L^+_1 := L^+_2 \cap \Sigma^+ \) with those in \( D_t ; G_t ; G_\Sigma \).
\[
\Sigma; G_k; \Gamma \vdash m : s^+ \ni e
\]
where \(\Sigma \cap \Sigma_G; G_k; \Gamma \vdash m : s^+\) and \(\Gamma \vdash e : s\)

\[
\Sigma; G_k; \Gamma \vdash (\_) : 1^+ ()
\]

\[
\Sigma; G_k; \Gamma \vdash \text{inj}_1 m : s_1^+ \ni s_2 \ni \text{inj}_e
\]

\[
\Sigma; G_k; \Gamma \vdash x : s^+ \ni \Gamma
\]

\[
\Sigma; G_k; \Gamma \vdash \lambda_x m : s_1^+ \ni \lambda x : s_1 \ni e
\]

\[
\Sigma; G_k; \Gamma \vdash \eta \eta \rightarrow m
\]

\[
\Sigma; G_k; \Gamma \vdash \text{prj}_2 m : s^+ \rightarrow s^+ \ni e
\]
\[ F \text{ to DCC: Back-Translation Contexts} \]

\[ F_\omega \text{ ctx} \quad F ::= \text{prj}_1 \mid \text{prj}_2 \mid \text{prj}_1 \text{ prj}_2 \mid \text{prj}_2 \text{ prj}_1 \]

\[ \Gamma \text{ DCC ctx} \quad F ::= \text{prj}_1 \mid \text{prj}_2 \mid \text{prj}_1 \text{ prj}_2 \mid \text{prj}_2 \text{ prj}_1 \]

\[ E^F = F_0 [F_1 \ldots F_n] \]

\[ \Sigma \mathcal{D}; \Sigma G; G \kappa_1 \Gamma \rightarrow F_1 : t_1 \rightarrow s \]

\[ \forall i \in [1, n+1]. \exists \kappa_i . t_i = s \]

\[ \Sigma \mathcal{D}; \Sigma G; G \kappa_1 \Gamma \rightarrow F_1 : t_1 \rightarrow s \]

\[ \Sigma \mathcal{D}; \Sigma G; G \kappa_1 \Gamma \rightarrow F_1 : t_1 \rightarrow s \]

2. Subterms are not of translation type

Next, consider the scenario where the subterm \( m \) is of non-translation type \( s^+ \times s^+ \). Let us consider the structure of \( m \). If \( m \) is a value \( u \) that is not a variable, then \( u = (m_1, m_2) \). Hence, we can reduce \( \text{prj}_1 \) \( (m_1, m_2) \rightarrow m_1 \) and then back-translate \( m_1 : s_1^+ \). We will come back to the possibility of \( u \) being a variable shortly.

But what if \( m \) is not a value? Intuitively, there must be some redex \( F[u] \) in \( m \). If we can find that redex and reduce it, then we can eliminate a term of non-translation type and continue back-translating. For example, if \( m \) is the redex \( F[u] \), then we can reduce \( \text{prj}_1 \) \( F[u] \rightarrow \text{prj}_1 \) \( m' \) and continue back-translating \( \text{prj}_1 \) \( m' : s_1^+ \). Note that this means our back-translation depends on strong-normalization. We discuss this further in [8].

More generally, when a subterm is of non-translation type, we evaluate away terms of non-translation type by repeatedly reducing the inner-most redex until all subterms are of translation type. To find the inner-most redex of a term, we decompose the term into \( F_0[F_1[F_2 \ldots F_n[u]]] \) (case 2a). When a term cannot be decomposed like this and is of non-translation type, there is additional structure imposed by our type translation that we use to rewrite the term (case 2b).

2a. There exists a non-translation redex

Suppose we decompose a term into \( E^F[u] = F_0[F_1[F_2 \ldots F_n[u]]] \), where the result of \( F_0 \) is a translation type and the hole and result types of all other \( F_i \) are non-translation types. We refer to the redex \( F_0[u] \) as a non-translation redex. We use \( E^F \) to denote such a sequence of \( F_i \) contexts, formally defined in Figure 11. In this case, the rule FD-HOLE applies. We perform one step of evaluation, eliminating the redex \( F_0[u] \), then continue back-translating the resulting term. We might worry that here \( u \) can be a variable, and thus \( E^F[u] \) is stuck. However, recall by assumption that the hole of \( F_0 \) must be of non-translation type. Let us consider which variables can appear in the hole of \( F_n \).

During back-translation, only certain variables are free. Free variables can be either the coercion functions \( \text{c}_{\text{prj}} \), the proof constructors \( \text{prj}_1 \), the variables \( x : s^+ \) from functions of translation type, or the variables \( k : s^+ \rightarrow (T_y s^+) \) introduced by the back-translation. Clearly a variable of translation type cannot appear in the hole, so \( u \) cannot be one of the variables from the final two cases.

Suppose \( u \) is a coercion function \( \text{c}\_{\text{prj}} : \alpha_{\text{prj}} \rightarrow \alpha_{\text{prj}}. \) Then \( F_n[u] \) has type \( \alpha_{\text{prj}} \). However, a term of type \( \alpha_{\text{prj}} \) cannot appear in any evaluation context. This follows by considering the type of each evaluation context in \( F_n \). Since there must be at least an outer \( F_0 \) whose result is of translation type, \( u \) cannot be a coercion function.

Finally, suppose \( u \) is a proof constructor, for instance, \( \text{prj}_1 \). Then \( F_n[u] = \text{prj}_1 \) \( [t] \) has type \( \alpha_{\text{prj}} \cdot \alpha_{\text{prj}}. \). Again, terms of this type cannot appear in any evaluation context due to the types of evaluation context in \( F_n \). Similar reasoning applies to all proof constructors since after wrapping them in some number of \( F \) contexts the final result will be a term of protection type which cannot appear in any evaluation context. So \( u \) cannot be a proof constructor. Hence, we conclude that it is impossible for a value \( u \) in \( E^F[u] \) to be a variable.

2b. There does not exist a non-translation redex

Note that the FD-HOLE \( \Gamma \) rule requires that the hole of each \( F_i \) be of non-translation type. Suppose that before we find a non-translation redex, we reach some boundary where the hole of a context has translation type. That is, we decompose a term into \( E^F[F_i[m]] = F_0[F_1[F_2 \ldots F_n[m]]] \), where \( m \) is the first term (going outside in) that has translation type, but the result of \( F_i \) is of non-translation type. In this case \( E^F[F_i] \) is not a valid \( E^F \) and there is no non-translation redex. This could happen, for instance, if \( m \) is a variable of translation type, which we can only rule out when \( E^F[F_i] \) is a valid \( E^F \). Let us analyze what \( F_i \) could be.

If \( F_i = \text{prj}_1 \) \( [\cdot] \), then \( m \) must have type \( s^+ \times s^+ \). But the result of \( F_i \) is a translation type, so \( F_i \) cannot be a proof constructor.

If \( F_i = [\cdot] \), then \( m' \) must have type \( s^+ \rightarrow s^+ \). But the result of \( F_i \) is a translation type, so \( F_i \) cannot be an application.

Finally, if \( F_i \) is \( \text{case m of inj_1 x_1 m_1 \ inj_2 x_2 m_2} \), then \( m \) must have type \( s^+ \rightarrow s^+ \), yet the result of \( F_i \) can be of non-translation type! That is, we are trying to back-translate the following expression:

\[ \ldots \vdash E^F[\text{case m of inj_1 x_1 m_1 \ inj_2 x_2 m_2}] : s_1^+ \]

We could back-translate \( E^F[\text{case m of inj_1 x_1 m_1 \ inj_2 x_2 m_2}] \) if we could rewrite this expression into one in which all the subterms are of translation type. Recall that we know the result of \( E^F \) must result in a translation type, and by assumption \( m \) is of translation type. Therefore, we can rewrite the term! Specifically, we rewrite the term into:

\[ \ldots \vdash E^F[\text{case m of inj_1 x_1 \ inj_2 x_2 \ inj_3 x_3} : s_1^+] \]

This scenario is captured by the rule FD-HOLE \( \Gamma \). This rule is analogous to the commuting conversion used by Shikuma and Igarashi [17]. All of the subterms of this rewritten term are of translation type, so we can continue back-translating the structurally smaller terms.

Back-translation is well-founded

While Ahmed and Blume claimed that their back-translation is well-founded by a nested induction metric, careful inspection of their back-translation reveals that their nested induction metric is not valid for their back-translation. We believe that their back-translation is well-founded, but that a more advanced technique is necessary to prove it.

To prove that our back-translation is well-founded, we use a novel logical relations argument. We formalize an open unary logical relation and prove that any well-typed \( F_n \) term under \( \Sigma \) belongs to this logical relation. When proving strong normalization of the simply-typed lambda calculus via logical relations, one wants terms to belong to the relation if the terms evaluate to a value. We, however, do not wish to “run” terms; we want terms \( m \) to belong to our logical relation if there exists a term \( e \) such that \( m \) back-translates to \( e \). Back-translation performs partial evaluation, but the “normalization” done during back-translation differs from evaluation using the \( F_n \) dynamic semantics, for instance, because we perform reduction under \( \lambda \).

We briefly present the logical relation at a high-level. Full definitions and proofs are available in our online technical appendix [7].
\[ \text{Atom}^t \Sigma^E = \{ ((G_k, \Gamma^t), m) \mid \Sigma_D, \Sigma_G, G_k, \Gamma^t \vdash m : \delta(t) \} \]

\[ \text{Atom}^{\text{ctx}}^t \Sigma^E = \{ ((G_k(G_k, \Gamma^t), E^t)) \mid \Sigma_D, \Sigma_G, G_k, \Gamma^t \vdash E^t : s^t \} \]

\[ \Theta^t \Sigma^E = \{(W, m) \mid (W, m) \in \text{Atom}^t \Sigma^E \} \wedge \exists \Sigma, \Psi, W_k : W_k \vdash m : s^t \}

\[ \text{Rel}^t \Sigma^E = \{ (s^t, R) \mid R \subseteq \Theta^t \Sigma^E \wedge \forall m, W, (W, m) \in R \wedge W' \supseteq W \implies (W', m) \in R \}
\]

\[ \text{Rel}^t \Sigma^E = \{ (t_1, t_2) \mid \forall (m_1, m_2) \in \text{Rel}^t \Sigma^E \wedge \forall u \in W \} \implies (R_1(t_1, t_2), R_2(t_1, t_2)) \}

\[ \text{Figure 15. } F_w \text{ to DCC: Well-Formed Relations} \]

The logical relation considers terms of type translation and non-
translation type separately. The essence of the relation is that a term of translation type belongs to the relation \( E^t[s^t]^E \) if it is back-translatable, while a term of non-translation type belongs to the relation \( \Theta^t \Sigma^E \) if plugging the term into a valid \( E^t \) context results in a term that is back-translatable. We use a \( \top \) closed-style relation to formalize this. In particular, we say that \( E^t \) belongs to the continuation relation \( \mathcal{K}(t, s^t)^E \) if, given any \( u \in \mathcal{V}(t)^E \), \( E^t[u] \) is back-translatable. Implicit in the definition of this logical relation is an obligation to show any expression we back-translate will not get stuck, formalizing our informal argument from earlier about rewriting \( E^t[p \text{ case of ind}_1, x_1 m_1 | \text{ind}_2, x_2 m_2] \). There is no \( \mathcal{V}(t)^E \) relation for translation types. The relation \( \mathcal{V}(t)^E \) for non-translation types is completely standard and ensures that subterms belong to the \( \Theta^t \Sigma^E \) relation, so we give only excerpts.

Our logical relation is based on a possible-worlds model. Typically, a possible-worlds model is necessary when membership in the relation depends on some state. For instance, a term may only be well-typed under certain heaps, and when evaluating that term the heap changes. Worlds are used to keep track of these possible heaps as they change. Our language is not “stateful” in the usual sense, but as we back-translate a term we may add new free variables to the environment—e.g., \( \cdot \), \( s^t \rightarrow T \) \( s^t \). Since target terms can only be back-translated under certain term environments—i.e., the environments \( G_k \) and \( \Gamma^t \)—we use worlds \( W \) to keep track of these environments as they are extended during the back-translation.

We define \( \text{Rel}^t \Sigma^E \) in Figure 15 which contains well-formed relations. For translation types at kind \( * \), a well-formed relation guarantees that the elements of the relation are back-translatable. For non-translation types at kind \( * \), a well-formed relation guarantees that for all contexts \( E^t \), if filling \( E^t \) with a value of type \( s^t \) results in a back-translatable term, so does filling \( E^t \) with elements of \( R \). Relations on types of higher kinds are well-formed if, given equivalent relations, they produce equivalent relations. We omit the definition of equivalence on relations as the definition is standard and analogous to our \( F_w \) definition of relation equivalence.

Finally, since this logical relation is for \( F_w \) terms, we require an interpretation of the binding judgment as is the case in our \( D \) relation in 5.2. These relations, defined in Figure 17 are standard.

The definition of the top-level-back-translated logical relation, \( D \), \( \Delta, G_k, G_k, \Gamma \vdash m : t \), first closes all free variables in \( \Delta \) and \( G \). We omit the definitions of \( D^t[\Delta]^E \) and \( G[\Gamma]^t \), which are standard. We also permit closing some of the variables in \( \Gamma^t \). This is to account for partial evaluation. For example, a function whose parameter is of translation type may be back-translated by leaving the variable free, if the result is of translation type, or by reducing the function via FD-\text{-}HOL;\( E^t \), if the result is of non-translation type. All the above environments are required to obtain a strong enough induction hypothesis. Below we show that any well-formed \( F_w \) term, as long as it is open with respect to \( \Sigma \) and the other environments required by the back-translation, belongs to the logical relation.

**Lemma 7.1 (Type interpretation is well-formed)**

*If \( \Delta \vdash t : \kappa \) and \( s \in D^{\Delta}[\kappa]^E \) then*

1. \( (s(t), T[t :: \kappa]^E) \in \text{Rel}^t \Sigma^E \)

2. \( s' \in D^{\Delta}[\kappa]^E \) such that \( s \equiv D^{\Delta}[\kappa]^E \), then \( T[t :: \kappa]^E \equiv D^{\Delta}[\kappa]^E \)
Lemma 7.2 (Fundamental property of logical relation) 
If \( D_1, \Delta : G_k T, G_k \vdash \Gamma \vdash m : t \) then 
\( D_1, \Delta : G_k T, G_k \vdash \Gamma \vdash \rightarrow m : \rightarrow t \).

Finally, as a corollary of the fundamental property above, we show that the back-translation exists.

Corollary 7.3 (Back-translation exists (1))
If \( \Sigma D_1 \vdash m : s^+ \vdash t \) then \( \exists \alpha, \Sigma : G_k T \vdash \Gamma \vdash \rightarrow m : \rightarrow s^+ \).

Corollary 7.4 (Back-translation exists (2))
If \( \Sigma D_1 \vdash m : s^+ \vdash t \) then \( \exists \alpha, \Sigma : \rightarrow m : \rightarrow s^+ \).

The first corollary form is needed to show the back-translation exists in its general form. The second corollary is the form needed in the proof of equivalence preservation. Note that the contexts are empty except for the protection ADT.

Properties of back-translation
To show our back-translation preserves semantics we prove Corollary 7.5. Note that in order to have a strong enough induction hypothesis we first prove Lemma 7.6 which quantifies over arbitrary \( G_k T \) environments. We substitute variables in \( G_k T \) with appropriate \( \alpha \).

Lemma 7.5
Let \( \gamma_k = \{ k \mapsto \alpha_k : k : s^+ \mapsto (T_\ell) s^+ \in G_k \} \)
\( \delta = \{ \alpha_\ell \mapsto \alpha_\ell, \ell \in L_1 \} \cup \{ \alpha_\ell \mapsto \alpha_\ell, \ell \in L_2 \} \).

If \( \Sigma D_1 \vdash \Sigma G_k T \vdash \Gamma \vdash m : s^+ \vdash t \) then
\( \exists \alpha, \Sigma : G_k T \vdash \Gamma \vdash \rightarrow m : \rightarrow s^+ \) and \( \Gamma \vdash \rightarrow \equiv \gamma_k (m) : s \vdash \delta \).

Corollary 7.6 (Back-translation preserves semantics)
If \( \Sigma D_1 \vdash m : s^+ \vdash t \) then \( \exists \alpha, \Sigma : \rightarrow m : \rightarrow s^+ \) and \( \Sigma \vdash \equiv \gamma_k (m) : s \vdash \delta \).

7.2 Preservation of Observer-Sensitive Equivalence
With the back-translation defined, we prove that the translation preserves equivalence. To prove equivalence preservation, we must simultaneously prove equivalence reflection. That is, the statement of our theorem is in two parts. Part 1 (preservation) states that given related source terms \( e_1 \) and \( e_2 \) that translate to target terms \( m_1 \) and \( m_2 \), the target terms must be related. Part 2 (reflection) states the converse: if the translations of two source terms are related, the source terms must be related.

Theorem 7.7 (\( \approx \) preservation and reflection)
Let \( \Gamma \vdash e_1 : s \approx m_1 \) and \( \Gamma \vdash e_2 : s \approx m_2 \).
1. If \( \Gamma \vdash e_1 \approx e_2 : s \) then \( \Gamma \vdash e_1 \approx e_2 : s \).
2. If \( \Gamma \vdash e_1 \approx e_2 : s \) then \( \Gamma \vdash e_1 \approx e_2 : s \).

Indirect proof of noninterference
Finally, as a sanity check that our notion of noninterference in \( F_\pi \) makes sense, we can prove that noninterference in DCC, Theorem 5.2, follows from parametricity:

Theorem 5.2 (Noninterference)
If \( \Gamma \vdash e : s \) then \( \forall \gamma, \Gamma \vdash e \approx e : s \).

Indirect Proof of Noninterference:
By correctness of the translation:
\( \Gamma \vdash e : s \approx m, \Gamma \vdash e \approx m : s, \) and \( \Gamma \vdash e \approx m : s \).

By parametricity, \( \Gamma \vdash e \approx m : s \).

Since \( \Gamma \vdash e \approx m : s \) quantifies over all \( \rho, \gamma, \gamma' \), the particular implementations we use in the observer-sensitive definition work, so \( \Gamma \vdash e \approx m : s \).

By reflection, \( \Gamma \vdash e \approx e : s \).

8. Related Work, Future Work, and Conclusion
In [3, 18] we examined the counterexample Shikuma and Igarashi [17, 18] gave to show that Tse and Zdancewic’s, from a variant of DCC to a simply-typed target language. Specifically, their source language was the sealing calculus—a simply-typed \( \lambda \)-calculus with operations for sealing at level \( k \) and unsealing—which they proved equivalent to a variant of DCC introduced by Tse and Zdancewic [19] called DCC\(_{pi} \) for “DCC with protection contexts.” DCC\(_{pi} \) has a lattice of monads like DCC and associated \( \eta_k \) and \( \eta_k \) operations but a different type system. DCC\(_{pi} \) typing judgments have the form \( \Gamma \vdash e : s \) where \( \pi : \pi, \eta \vdash \ell \in L_\ell \) is the protection context. The typing rules for \( \pi, \eta \) and \( \eta, \pi \) are:

\[ \Gamma \vdash e : s \]
\[ \Gamma \vdash \eta : e_1 : s_1 \]
\[ \Gamma \vdash x : s_1 \]
\[ \Gamma \vdash \eta \vdash s_2 \]
\[ \pi : \pi, \eta \vdash \ell \in L_\ell \]
\[ \pi, \eta \vdash e_1 \in e_2 : s_2 \]

Note that the premise \( \pi : \pi, \eta \vdash \ell \in L_\ell \) means that either \( \ell \leq s_2 \) as usual or \( \ell \leq \ell' \) for some \( \ell' \in \pi \). Thus, the following term—which is ill typed in DCC—is well typed in DCC\(_{pi} \):

\[ e = \eta \eta (\lambda y : T_\ell \triangleright \text{bool} \mapsto \text{unit} \mapsto y) : T_\ell \]

This is the same ill-typed term we discussed in [3]. Thus, Shikuma and Igarashi have weakened their source language to admit terms disallowed by DCC. While DCC\(_{pi} \) is of independent interest and arguably has a more pragmatic type system because it admits terms that intuitively should be well-behaved in DCC, adding this rule simplifies the proof of full-abstraction.

In addition to weakening their source language, Shikuma and Igarashi also strengthen their target language: it admits fewer terms than the source—e.g., we can encode arithmetic operations in \( F_\pi \) but not in DCC. This is important because, in general, relying on a close correspondence between source and target is not practical since we want to be able to implement dependency calculi in rich general-purpose languages—or compile them to intermediate representations—that may be more expressive than the source language. Shikuma and Igarashi’s inverse translation relies on full beta-reduction and “commuting conversions” that they add to their source and target operational semantics. This reduces terms to a normal form that satisfies a subformula property; if the term has type \( s \) then all of its subterms have a type that appears within \( s \). Also, their inverse translation cannot be extended to languages with recursion since it relies on “normalization” of terms. Our rewriting rule for stuck terms is similar to their commuting conversions but our back-translation does not demand any changes to the standard call-by-name operational semantics of the source or target, and it is designed to be easily extended to a setting with recursion, following the proposal by Ahmed and Blume [5].

Fully Abstract Translation
As noted earlier, our key result (Theorem 7.7) is reminiscent of full abstraction. Proving full abstraction is particularly difficult when the target language is more expressive than the source language, as is the case with \( F_\pi \) and DCC. Our back-translation is based on Ahmed and Blume’s [5] but is more

\[ ^{3} \] Much prior work resorts to bringing the source and target languages into closer correspondence in order to prove full abstraction; see Ahmed and Blume [5] for a discussion.
challenging because our target language is not in CPS form as theirs is. Other work on proving translations fully abstract takes advantage of the source and target language being syntactically identical, though proving that the transformation preserves equivalence is still a nontrivial result. For instance, Ahmed and Blume [4] prove that typed closure conversion for System F with recursive types is fully abstract in this way. Fournet et al. [9] prove that a translation from a λ-calculus with references and exceptions to an encoding of JavaScript in the source language is fully abstract.

Recurrent Like Tse and Zdancewic, we have focused on the terminating fragment of DCC, leaving recursion as future work. One can extend DCC with recursion by adding pointed types and a fix operator (as in the original version of DCC [2]). We foresee three issues that will need to be addressed. First, the back-translation would need to be extended to work in the presence of recursion, which we are fairly confident can be done following the proposal by Ahmed and Blume [5]. Second, the parametricity condition (Lemma 6.1), central to our proof of semantics preservation, does not hold as stated in the presence of effects such as recursion. To prove a similar lemma in the presence of recursion, our type translation will have to make use of linear types to ensure that the continuation is used exactly once as in the work on linearly-used continuations [6]. Third, in the presence of recursion we would need to use logical relations that are step indexed [3]. Our proof of semantics preservation relies on transitivity across the cross-language and target logical relations, but it is not known how to prove transitivity for cross-language step-indexed logical relations. This can be handled by defining a multi-language semantics [12,14] for DCC and Fω, which would then allow us work with the definition of contextual equivalence for the multi-language whenever transitivity is required. This strategy also has the advantage of scaling to correctness of multi-pass compilers [14], which is not the case for cross-language logical relations.

Conclusion It is folklore that noninterference can be encoded via parametricity but we are unaware of any work that successfully shows how to do that. By expressing source-level noninterference using target-level parametricity, we can implement security-typed features in a more standard (polymorphic) typed language. Furthermore, ensuring that compilation preserves noninterference is important if code compiled from security-typed languages is to be linked with target components compiled from other source languages, or those written directly in the target language. We give a translation from DCC to Fω that leverages first-order and higher-order parametricity to encode the key property required to ensure that the translation preserves source-level noninterference.

Several elements of our translation and proof techniques should be applicable to security-preserving and fully abstract compilation. We provide a more general back-translation technique as compared to prior work; we expect this to be useful for proving translations fully abstract. We show how to encode DCC’s security lattice and protection judgment using our protection ADT at the target level; a similar strategy could be used to encode other specialized security or safety properties captured by the source type system. Finally, we demonstrate the use of an open logical relation at the target-level to prove parametricity while also accommodating back-translation of target terms that need to be linked with such ADTs.

Acknowledgments We gratefully acknowledge the valuable feedback provided by anonymous reviewers and J. Ian Johnson on earlier versions of this paper. This research was supported in part by the National Science Foundation (grant CCF-1422133).

References