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_ω and prove that the translation preserves noninterference. To express noninterference in F_ω, 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:

$$\mathsf{e} : \mathsf{bool}_\mathsf{H} \to \mathsf{bool}_\mathsf{L}$$

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:

$$\mathsf{e}' : \mathsf{bool}_\mathsf{H} \times \mathsf{bool}_\mathsf{L} \to \mathsf{bool}_\mathsf{H} \times \mathsf{bool}_\mathsf{L}$$

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 [10]. 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 $\mathrm{DCC}_{\mathrm{pc}}$—for DCC with protection contexts—whose type system is more liberal than DCC's. Moreover, their translation targeted the simply typed λ-calculus, leaving open the explicit connection between noninterference and parametricity.

We provide a translation from DCC to F_ω, 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_ω, 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.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...
http://dx.doi.org/10.1145/2784731.2784733

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_ω (§4) and proof of its correctness (§6);
- the development of an *open* logical relation for F_ω (§5.2);
- the formalization of an observer-sensitive equivalence for F_ω terms of translation type using relational parametricity (§5.3);
- the back-translation from F_ω terms of translation type s^+ to DCC terms of type s (§7.1) 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 (§7.2).

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 λ-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 $\mathcal{L}$ with a set of labels $\mathcal{L}_\ell$[1] and an ordering on labels $\mathcal{L}_\sqsubseteq$. We write $\ell \sqsubseteq \ell'$ to mean ℓ' is at least as high as ℓ. The monadic type T_ℓ , and the return η_ℓ and bind operations are indexed by a label $\ell \in \mathcal{L}_\ell$.

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 ($\mathsf{e} \longmapsto \mathsf{e}'$) using evaluation contexts E to lift the primitive reductions to a call-by-name semantics for the language. We write $\longmapsto^*$ for the reflexive, transitive closure of $\longmapsto$. Note that in this call-by-name language, $\eta_\ell\,\mathsf{e}$ is a value form. The operation $\mathsf{bind}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2$ evaluates e_1 to a value $\eta_\ell\,\mathsf{e}$ and then substitutes e for x in e_2.

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

[1] Note that in $\mathcal{L}_\ell$, ℓ is part of the name for the set, not a meta-variable.

$$
\begin{array}{lll}
\textit{Types} & \mathsf{s} & ::= 1 \mid \mathsf{s}_1 \times \mathsf{s}_2 \mid \mathsf{s}_1 + \mathsf{s}_2 \mid \mathsf{s}_1 \to \mathsf{s}_2 \mid \mathsf{T}_\ell\,\mathsf{s} \\
\textit{Values} & \mathsf{v} & ::= \mathsf{x} \mid \langle\rangle \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \mid \mathsf{inj}_i\,\mathsf{e} \mid \lambda \mathsf{x} : \mathsf{s}.\,\mathsf{e} \mid \eta_\ell\,\mathsf{e} \\
\textit{Terms} & \mathsf{e} & ::= \mathsf{v} \mid \mathsf{prj}_i\,\mathsf{e} \mid \mathsf{case}\,\mathsf{e}\,\mathsf{of}\,\mathsf{inj}_1\,\mathsf{x}_1.\,\mathsf{e}_1 \parallel \mathsf{inj}_2\,\mathsf{x}_2.\,\mathsf{e}_2 \mid \\
 & & \quad \mathsf{e}_1\,\mathsf{e}_2 \mid \mathsf{bind}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2 \\
\textit{Eval. Ctxts} & \mathsf{E} & ::= [\cdot]_\mathsf{s} \mid \mathsf{prj}_i\,\mathsf{E} \mid \mathsf{case}\,\mathsf{E}\,\mathsf{of}\,\mathsf{inj}_1\,\mathsf{x}_1.\,\mathsf{e}_1 \parallel \mathsf{inj}_2\,\mathsf{x}_2.\,\mathsf{e}_2 \mid \\
 & & \quad \mathsf{E}\,\mathsf{e} \mid \mathsf{bind}\,\mathsf{x} = \mathsf{E}\,\mathsf{in}\,\mathsf{e}
\end{array}
$$

$\boxed{\mathsf{e} \longmapsto \mathsf{e}'}$

$$
\begin{array}{rcl}
(\lambda \mathsf{x} : \mathsf{s}.\,\mathsf{e}_1)\,\mathsf{e}_2 & \longmapsto & \mathsf{e}_1[\mathsf{e}_2/\mathsf{x}] \\
\mathsf{bind}\,\mathsf{x} = \eta_\ell\,\mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2 & \longmapsto & \mathsf{e}_2[\mathsf{e}_1/\mathsf{x}] \\
\multicolumn{3}{c}{\cdots}
\end{array}
$$

$\boxed{\Gamma \vdash \mathsf{e} : \mathsf{s}}$ *Term Environment* $\Gamma ::= \cdot \mid \Gamma, \mathsf{x} : \mathsf{s}$

$$
\cdots \quad \frac{\Gamma \vdash \mathsf{e} : \mathsf{s}}{\Gamma \vdash \eta_\ell\,\mathsf{e} : \mathsf{T}_\ell\,\mathsf{s}} \qquad \frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_\ell\,\mathsf{s}_1 \qquad \Gamma, \mathsf{x} : \mathsf{s}_1 \vdash \mathsf{e}_2 : \mathsf{s}_2 \qquad \ell \preceq \mathsf{s}_2}{\Gamma \vdash \mathsf{bind}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2 : \mathsf{s}_2}
$$

$\boxed{\ell \preceq \mathsf{s}}$

$$
\frac{}{\ell \preceq 1} \qquad \frac{\ell \preceq \mathsf{s}_1 \qquad \ell \preceq \mathsf{s}_2}{\ell \preceq \mathsf{s}_1 \times \mathsf{s}_2} \qquad \frac{\ell \preceq \mathsf{s}_2}{\ell \preceq \mathsf{s}_1 \to \mathsf{s}_2}
$$

$$
\frac{\ell \not\sqsubseteq \ell' \qquad \ell \preceq \mathsf{s}}{\ell \preceq \mathsf{T}_{\ell'}\,\mathsf{s}} \qquad \frac{\ell \sqsubseteq \ell'}{\ell \preceq \mathsf{T}_{\ell'}\,\mathsf{s}}
$$

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

with their types. Typing rules for all constructs except η_ℓ and bind are completely standard so we omit them here. The return operation $\eta_\ell\,\mathsf{e}$ protects the term e by wrapping it with the label ℓ. These protected terms can only be unwrapped through a bind operation $\mathsf{bind}\,\mathsf{x} = \mathsf{e}_1\,\mathsf{in}\,\mathsf{e}_2$. While e_2 may depend on the protected term inside e_1, the results produced by the entire bind expression (of type s_2) must be protected at the label ℓ or higher. This requirement is captured by the judgment $\ell \preceq \mathsf{s}_2$ (pronounced "s_2 is protected at ℓ").

Informally, a type s is protected at ℓ if expressions of this type do not leak information to levels lower than (or incomparable to) ℓ. Since there is only one value of type 1, this type cannot communicate any information, so it is protected at any ℓ. 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 $\ell \preceq \mathsf{T}_{\ell'}\,\mathsf{s}$. First, if s is protected at ℓ, then the type $\mathsf{T}_{\ell'}\,\mathsf{s}$ is protected at ℓ since unwrapping the protected value gives back an expression that is also protected at ℓ. Second, if ℓ' is at least as high as ℓ, then since the expression of type $\mathsf{T}_{\ell'}\,\mathsf{s}$ is protected already protected at the higher label ℓ', it is also protected at the lower label ℓ. Note that sum types are never protected; even the simplest sum type $1 + 1$ leaks information, and must be wrapped in a monadic type to be protected.

For use in examples, we encode bool as the sum $1 + 1$, true as $\mathsf{inj}_1\,\langle\rangle$, false as $\mathsf{inj}_2\,\langle\rangle$, and if using a case expression.

To see how DCC protects information, consider this example:

$$\lambda \mathsf{x} : \mathsf{T}_\mathsf{H}\,\mathsf{bool}.\,\mathsf{bind}\,\mathsf{y} = \mathsf{x}\,\mathsf{in}\,\mathsf{y}$$

This example is ill-typed. We cannot simply return y since it is of type bool, which is not protected at H. Instead, the typing rule for bind forces us to first protect the result. For example, we could instead return $\eta_\mathsf{H}\,\mathsf{y}$. This keeps the protected inputs to the function from being leaked.

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:

$$(\mathsf{T}_\ell\,\mathsf{s})^+ = \alpha_\ell \to \mathsf{s}^+$$

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

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 $\mathsf{T}_\mathsf{H}\,\mathsf{bool}$. 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 true and false at the type $\mathsf{T}_\mathsf{H}\,\mathsf{bool}$. DCC formalizes this situation using an *observer-indexed logical relation* that says these two values are *equivalent* at L (written $\eta_\mathsf{H}\,\mathsf{true} \approx_\mathsf{L} \eta_\mathsf{H}\,\mathsf{false} : \mathsf{T}_\mathsf{H}\,\mathsf{bool}$) but not at H. Hence, the relation $\approx_\mathsf{L}$ relates every possible pair of boolean expressions at the type $\mathsf{T}_\mathsf{H}\,\mathsf{bool}$, while the relation $\approx_\mathsf{H}$ at the same type only relates boolean expressions that evaluate to the same value. We will refer to this relation as *observer-sensitive equivalence*: we write $\mathsf{e}_1 \approx_\zeta \mathsf{e}_2 : \mathsf{s}$ to mean that terms e_1 and e_2 of type s are equivalent from the perspective of an observer at level ζ in the lattice.[2]

Intuitively, we must show that if $\mathsf{e}_1 \approx_\zeta \mathsf{e}_2 : \mathsf{s}$ and e_1 and e_2 translate to target terms $\mathbf{m_1}$ and $\mathbf{m_2}$, respectively, then $\mathbf{m_1} \approx_\zeta \mathbf{m_2} : \mathsf{s}^+$. The key is to formalize the target-level $\approx_\zeta$ relation in terms of the standard logical relation for System F (or F_ω), which is not indexed by an observer. To see how Tse and Zdancewic do this, consider the type $\mathsf{T}_\mathsf{H}\,\mathsf{bool}$ again, which they translate to the target type $\boldsymbol{\alpha}_\mathsf{H} \to \mathbf{bool}$. This translation uses abstract types $\boldsymbol{\alpha}_\ell$ to encode each element ℓ in the source-level lattice. This simulates DCC's observer-sensitive equivalence by requiring, in essence, that an L-observer not have access to any terms of type $\boldsymbol{\alpha}_\mathsf{H}$, which means that any function of type $\boldsymbol{\alpha}_\mathsf{H} \to \mathbf{bool}$ can never be applied. Hence, the two functions $\boldsymbol{\lambda}\mathbf{x}{:}\boldsymbol{\alpha}_\mathsf{H}.\mathbf{true}$ and $\boldsymbol{\lambda}\mathbf{x}{:}\boldsymbol{\alpha}_\mathsf{H}.\mathbf{false}$ are indistinguishable. Meanwhile, an H-observer is given access to terms of type $\boldsymbol{\alpha}_\mathsf{H}$ and can use these as keys to gain access to the computation hidden inside functions of type $\boldsymbol{\alpha}_\mathsf{H} \to \mathbf{bool}$.

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

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 [18]. DCC terms of the protected function type $\mathsf{s}_\mathsf{f} = \mathsf{T}_\ell\,((\mathsf{T}_\ell\,\mathsf{bool}) \to \mathsf{bool})$ must be equivalent to $\eta_\ell\,(\lambda \mathsf{x} : \mathsf{T}_\ell\,\mathsf{bool}.\,\mathsf{v})$, where v is either true or false. In particular, Shikuma and Igarashi point out that the following term is ill-typed due to the $\ell \preceq \mathsf{s}_2$ restriction in the typing rule for bind (since $\ell \not\preceq \mathsf{bool}$).

$$\mathsf{e}_\mathsf{f} = \eta_\ell\,(\lambda \mathsf{x} : \mathsf{T}_\ell\,\mathsf{bool}.\,\mathsf{bind}\,\mathsf{y} = \mathsf{x}\,\mathsf{in}\,\mathsf{y}) \quad \texttt{// ill typed!}$$

Thus, the following two terms e_1 and e_2 are equivalent at type $\mathsf{s} = \mathsf{s}_\mathsf{f} \to \mathsf{T}_\ell\,\mathsf{bool}$ for an observer at level ℓ, since the only functions we can pass in for f are the constant functions above—*i.e.*, we cannot pass in non-constant functions such as e_f (since they are not well-typed).

$$\mathsf{e}_1 = \lambda \mathsf{f} : \mathsf{s}_\mathsf{f}.\,\mathsf{bind}\,\mathsf{f}' = \mathsf{f}\,\mathsf{in}\,\eta_\ell\,(\mathsf{f}'\,(\eta_\ell\,\mathsf{true}))$$

$$\mathsf{e}_2 = \lambda \mathsf{f} : \mathsf{s}_\mathsf{f}.\,\mathsf{bind}\,\mathsf{f}' = \mathsf{f}\,\mathsf{in}\,\eta_\ell\,(\mathsf{f}'\,(\eta_\ell\,\mathsf{false}))$$

While e_1 and e_2 are equivalent at type s at level ℓ in DCC, their translations are *not* equivalent at the following type s^+ in System F.

$$\mathsf{s}^+ = (\boldsymbol{\alpha}_\ell \to ((\boldsymbol{\alpha}_\ell \to \mathbf{bool}) \to \mathbf{bool})) \to (\boldsymbol{\alpha}_\ell \to \mathbf{bool})$$

The term $\mathbf{m_f}$ defined below, which corresponds to e_f, can distinguish the translations of e_1 and e_2:

$$\mathbf{m_f} = \boldsymbol{\lambda}\mathbf{k}{:}\boldsymbol{\alpha}_\ell.\,\boldsymbol{\lambda}\mathbf{y}{:}\boldsymbol{\alpha}_\ell \to \mathbf{bool}.\,\mathbf{y}\,\mathbf{k}$$

Hence, we have two terms of type s that are equivalent (at ℓ) in DCC, but their translations are not equivalent at the type s^+ (at ℓ) 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 bind prevents us from concluding that $\mathsf{e}_\mathsf{f} : \mathsf{s}_\mathsf{f}$, we should not be able to conclude that the corresponding (behaviorally equivalent) target term $\mathbf{m_f}$ is well-typed at s_f^+. In more detail, consider the restrictions on the continuation inside e_f that uses $\mathsf{y} : \mathsf{T}_\ell\,\mathsf{bool}$—*i.e.*, that the bind expression inside e_f must have a result type that is protected at ℓ. In contrast, there is no such restriction on the body of $\mathbf{m_f}$ that uses $\mathbf{y} : (\mathsf{T}_\ell\,\mathsf{bool})^+$. 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 bind. We need to find an alternative translation for types of the form $\mathsf{T}_\ell\,\mathsf{s}$ that captures these restrictions so that for every $\mathsf{e} : \mathsf{T}_\ell\,\mathsf{s}$, all target-level uses of the translation of e *must* satisfy the same constraints as the ones imposed by the 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 $\approx_\zeta$ at the target level (§5.3) is semantically pleasing and insightful given the semantics of DCC. We define $\approx_\zeta$ at the target-level by instantiating a novel *open* logical relation for F_ω, which interprets types as relations on open terms rather than closed terms as is standard. In §5.2 we explain why we need this. Our open logical relation is inspired by Zhao *et al.*'s open logical relation for a linear System F [22].

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, $\mathbf{m} : \mathsf{s}^+$, there exists a semantically equivalent source term $\mathsf{e} : \mathsf{s}$. We formalize this via a "back-translation" relation between target terms of type s^+ and source terms of type s (§7.1). The need for back-translation arises when proving that if two source functions f_1 and f_2 are equivalent then their translations $\mathbf{f_1}$ and $\mathbf{f_2}$ are equivalent. To show the latter, we must assume that we're given two equivalent target-level arguments $\mathbf{m_1}$ and $\mathbf{m_2}$ and show that $\mathbf{f_1}\,\mathbf{m_1}$ is equivalent to $\mathbf{f_2}\,\mathbf{m_2}$. 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 $\mathbf{m_1}$ and $\mathbf{m_2}$ to source terms e_1 and e_2—*which is exactly the failure the counterexample exploits*—then we could conclude that $\mathsf{f}_1\,\mathsf{e}_1$ is equivalent to $\mathsf{f}_2\,\mathsf{e}_2$ which implies that $\mathbf{f_1}\,\mathbf{m_1}$ is equivalent to $\mathbf{f_2}\,\mathbf{m_2}$ 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 (*e.g.*, F_ω can encode natural numbers, while DCC cannot). Back-translation in this setting is more complicated. We give a more detailed comparison in §8.

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 bind typing rule resembles the typing rule for unpacking a term of existential type:

$$\frac{\Delta; \Gamma \vdash e_1 : \exists \alpha.\,\tau \qquad \Delta, \alpha; \Gamma, x : \tau \vdash e_2 : \tau_2 \qquad \alpha \notin \mathrm{ftv}(\tau_2)}{\Delta; \Gamma \vdash \mathrm{unpack}\ \alpha, x\ =\ e_1\ \mathrm{in}\ e_2 : \tau_2}$$

Note that, just as the bind rule requires the side condition $\ell \preceq \mathsf{s}_2$, the unpack rule requires the side condition $\alpha \notin \mathrm{ftv}(\tau_2)$. This idea inspires our translation, because a simple encoding captures this side condition through parametricity.

[2] Following convention, we represent the level of the observer using the meta-variable ζ rather than ℓ.

Recall that the encoding of existential types using universal types captures all of the restrictions imposed by the unpack typing rule:

$$\exists\alpha.\,\tau \stackrel{\text{def}}{=} \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

The above encoding says that an existential package is a data value that, given a result type τ_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 α and a value of type τ, and uses them to compute a result of type τ_2. In particular, note that since α is not in scope when we instantiate β, the above encoding perfectly captures the $\alpha \notin \text{ftv}(\tau_2)$ requirement from the unpack typing rule since we cannot instantiate β with any τ_2 with a free α.

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

$$(\mathsf{T}_\ell\ \mathsf{s})^+ \stackrel{\text{roughly}}{=} \forall\beta.\,([\![\ell \preceq \beta]\!] \times (\mathsf{s}^+ \to \beta)) \to \beta$$

This encoding says that a protected computation is a data value that, given a result type $\mathbf{t}_2$, a proof that $\mathbf{t}_2$ is protected at ℓ—which we informally write as $[\![\ell \preceq \mathbf{t}_2]\!]$ 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 $\mathbf{t}_2$. In particular, note that we must provide a proof that the result type of the continuation is protected at ℓ, only then will this data value call the continuation to compute a result.

The remaining question then is how to encode the type $[\![\ell \preceq \mathbf{t}_2]\!]$. Note that we should only be able to construct a term with the "protection type" $[\![\ell \preceq \mathbf{t}_2]\!]$ if $\exists \mathsf{s}_2.\mathbf{t}_2 = \mathsf{s}_2^+$ and $\ell \preceq \mathsf{s}_2$. First, like Tse-Zdancewic, we shall use type variables $\boldsymbol{\alpha}_\ell$ to encode each $\ell \in \mathcal{L}_\ell$. Then, our desired "protection type" can be built using an abstract type constructor $\boldsymbol{\alpha}_\preceq :: * \to * \to *$ applied to the types $\boldsymbol{\alpha}_\ell$ and $\mathbf{t}_2$. That is, we shall represent $[\![\ell \preceq \mathbf{t}_2]\!]$ with the protection type $(\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \mathbf{t}_2)$. Of course, in addition to introducing the higher-kinded abstract type $\boldsymbol{\alpha}_\preceq$ and a set of type variables $\boldsymbol{\alpha}_\ell :: *$ for each $\ell \in \mathcal{L}_\ell$, we must also provide an interface for constructing terms of protection type—*e.g.*, given terms of type $(\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \mathbf{t}_1)$ and $(\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \mathbf{t}_2)$, we should be able to construct a term of type $(\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ (\mathbf{t}_1 \times \mathbf{t}_2))$. 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:

$$(\mathsf{T}_\ell\ \mathsf{s})^+ = \forall\beta::*.\,((\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \beta) \times (\mathsf{s}^+ \to \beta)) \to \beta$$

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

4. Translating DCC to F_ω

In this section, we begin by presenting the target language F_ω and then give a type-directed translation from DCC to F_ω.

Target language: F_ω The target language F_ω is the call-by-name, higher-order polymorphic lambda calculus with unit, pairs, and sums. Figure 2 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 [15]).

Typing judgments in F_ω have the form $\Delta; \Gamma \vdash \mathbf{m} : \mathbf{t}$, which says that an F_ω term $\mathbf{m}$ has type $\mathbf{t}$ under type environment Δ and term environment Γ. The kinding judgment $\Delta \vdash \mathbf{t} :: \kappa$ says that a type $\mathbf{t}$ has kind κ under type environment Δ, where Δ maps abstract types $\boldsymbol{\alpha}$ to kinds κ. Since we have type-level functions, *i.e.*, type constructors, we define type equivalence $\mathbf{t}_1 \equiv \mathbf{t}_2$ to account for beta-reduction at the type-level.

Translation We start by defining a translation from DCC types to F_ω 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 $\mathsf{T}_\ell\ \mathsf{s}$ is translated to the type

$$\begin{array}{lll}
\textit{Kinds} & \kappa & ::= * \mid \kappa \to \kappa \\
\textit{Types} & \mathbf{t} & ::= \mathbf{1} \mid \mathbf{t}_1 \times \mathbf{t}_2 \mid \mathbf{t}_1 \to \mathbf{t}_2 \mid \boldsymbol{\alpha} \mid \forall\boldsymbol{\alpha}::\kappa.\,\mathbf{t} \mid \mathbf{t}_1 + \mathbf{t}_2 \mid \\
 & & \lambda\boldsymbol{\alpha}::\kappa.\mathbf{t} \mid \mathbf{t}_1\,\mathbf{t}_2 \\
\textit{Values} & \mathbf{u} & ::= \mathbf{x} \mid \langle\rangle \mid \langle \mathbf{m}_1, \mathbf{m}_2 \rangle \mid \lambda\mathbf{x}{:}\mathbf{t}.\mathbf{m} \mid \Lambda\boldsymbol{\alpha}::\kappa.\mathbf{m} \mid \mathbf{inj}_\mathbf{i}\ \mathbf{m} \\
\textit{Terms} & \mathbf{m} & ::= \mathbf{u} \mid \mathbf{prj}_\mathbf{i}\ \mathbf{m} \mid \mathbf{m}_1\ \mathbf{m}_2 \mid \mathbf{m}\,[\mathbf{t}] \mid \\
 & & \mathbf{case}\ \mathbf{m}\ \mathbf{of}\ \mathbf{inj}_1\ \mathbf{x}_1.\,\mathbf{m}_1 \parallel \mathbf{inj}_2\ \mathbf{x}_2.\ \mathbf{m}_2
\end{array}$$

$\boxed{\Delta \vdash \mathbf{t} :: \kappa}$

$$\begin{array}{lll}
\textit{Type Env.} & \Delta & ::= \cdot \mid \Delta, \boldsymbol{\alpha} :: \kappa \\
\textit{Term Env.} & \Gamma & ::= \cdot \mid \Gamma, \mathbf{x} : \mathbf{t}
\end{array}$$

$$\cdots \quad \frac{\boldsymbol{\alpha} :: \kappa \in \Delta}{\Delta \vdash \boldsymbol{\alpha} :: \kappa} \qquad \frac{\Delta, \boldsymbol{\alpha} :: \kappa \vdash \mathbf{t} :: *}{\Delta \vdash \forall\boldsymbol{\alpha}::\kappa.\,\mathbf{t} :: *}$$

$$\frac{\Delta, \boldsymbol{\alpha} :: \kappa_1 \vdash \mathbf{t} :: \kappa_2}{\Delta \vdash \lambda\boldsymbol{\alpha}::\kappa.\mathbf{t} :: \kappa_1 \to \kappa_2}$$

$$\frac{\Delta \vdash \mathbf{t}_1 :: \kappa_1 \to \kappa_2 \qquad \Delta \vdash \mathbf{t}_2 :: \kappa_1}{\Delta \vdash \mathbf{t}_1\,\mathbf{t}_2 :: \kappa_2}$$

$\boxed{\Delta; \Gamma \vdash \mathbf{m} : \mathbf{t}}$

$$\cdots \quad \frac{\Delta; \Gamma, \mathbf{x} : \mathbf{t}_1 \vdash \mathbf{m} : \mathbf{t}_2 \qquad \Delta \vdash \mathbf{t}_1 :: *}{\Delta; \Gamma \vdash \lambda\mathbf{x}{:}\mathbf{t}_1.\mathbf{m} : \mathbf{t}_1 \to \mathbf{t}_2}$$

$$\frac{\Delta, \boldsymbol{\alpha} :: \kappa; \Gamma \vdash \mathbf{m} : \mathbf{t}}{\Delta; \Gamma \vdash \Lambda\boldsymbol{\alpha}::\kappa.\mathbf{m} : \forall\boldsymbol{\alpha}::\kappa.\,\mathbf{t}}$$

$$\frac{\Delta; \Gamma \vdash \mathbf{m} : \forall\boldsymbol{\alpha}::\kappa.\,\mathbf{t}_1 \qquad \Delta \vdash \mathbf{t}_2 :: \kappa}{\Delta; \Gamma \vdash \mathbf{m}\,[\mathbf{t}_2] : \mathbf{t}_1[\mathbf{t}_2/\boldsymbol{\alpha}]}$$

$$\frac{\Delta; \Gamma \vdash \mathbf{m} : \mathbf{t}_1 \qquad \mathbf{t}_1 \equiv \mathbf{t}_2 \qquad \Delta \vdash \mathbf{t}_2 :: *}{\Delta; \Gamma \vdash \mathbf{m} : \mathbf{t}_2}$$

$\boxed{\mathbf{t} \equiv \mathbf{t}'}$

$$\cdots \quad \frac{\mathbf{t}_1 \equiv \mathbf{t}_2}{\lambda\boldsymbol{\alpha}::\kappa.\mathbf{t}_1 \equiv \lambda\boldsymbol{\alpha}::\kappa.\mathbf{t}_2} \qquad \frac{\mathbf{t}_1 \equiv \mathbf{t}_1' \qquad \mathbf{t}_2 \equiv \mathbf{t}_2'}{\mathbf{t}_1\,\mathbf{t}_2 \equiv \mathbf{t}_1'\,\mathbf{t}_2'}$$

$$\frac{}{(\lambda\boldsymbol{\alpha}::\kappa.\mathbf{t}_1)\ \mathbf{t}_2 \equiv \mathbf{t}_1[\mathbf{t}_2/\boldsymbol{\alpha}]}$$

Figure 2. F_ω: Syntax and Static Semantics (excerpts)

$$\begin{array}{rcl}
\mathcal{L}_\ell^+ & = & \{\boldsymbol{\alpha}_\ell :: * \mid \ell \in \mathcal{L}_\ell\} \cup \{\boldsymbol{\alpha}_\preceq :: * \to * \to *\} \\
\mathcal{L}_\sqsubseteq^+ & = & \{\mathbf{c}_{\ell'\ell} : \boldsymbol{\alpha}_{\ell'} \to \boldsymbol{\alpha}_\ell \mid \ell \sqsubseteq \ell' \in \mathcal{L}_\sqsubseteq\}
\end{array}$$

$\boxed{\mathsf{s}^+}$ where $\mathcal{L}_\ell^+ \vdash \mathsf{s}^+ :: *$

$$\begin{array}{rclrcl}
\mathbf{1}^+ & = & \mathbf{1} & (\mathsf{s}_1 + \mathsf{s}_2)^+ & = & \mathsf{s}_1^+ + \mathsf{s}_2^+ \\
(\mathsf{s}_1 \times \mathsf{s}_2)^+ & = & \mathsf{s}_1^+ \times \mathsf{s}_2^+ & (\mathsf{s}_1 \to \mathsf{s}_2)^+ & = & \mathsf{s}_1^+ \to \mathsf{s}_2^+
\end{array}$$

$$(\mathsf{T}_\ell\ \mathsf{s})^+ = \forall\beta::*.\,((\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \beta) \times (\mathsf{s}^+ \to \beta)) \to \beta$$

Figure 3. DCC to F_ω: Lattice (top) and Type (bottom) Translations

of a polymorphic function that expects a continuation and a proof that the result type of the continuation is protected at label ℓ. This requires encoding labels and the DCC $\ell \preceq \mathsf{s}$ judgment in F_ω.

Following Tse and Zdancewic [19], we encode the labels of the DCC lattice by generating a fresh abstract type $\boldsymbol{\alpha}_\ell$ for each ℓ in $\mathcal{L}_\ell$, defined as $\mathcal{L}_\ell^+$ in Figure 3. To encode the ordering on labels, $\mathcal{L}_\sqsubseteq$, we generate coercion functions $\mathbf{c}_{\ell'\ell}$ if $\ell \sqsubseteq \ell'$, defined as $\mathcal{L}_\sqsubseteq^+$ in Figure 3; informally, these allow us to convert a higher label ℓ' to a lower label ℓ.

To support encoding of the protection judgment $\ell \preceq \mathsf{s}$, our translation $\mathcal{L}_\ell^+$ also introduces an abstract type constructor $\boldsymbol{\alpha}_\preceq$. When we use the type constructor $\boldsymbol{\alpha}_\preceq$ in the translation, it takes a type representing a label (*i.e.*, an $\boldsymbol{\alpha}_\ell$) and some type s^+, and returns a type representing a proof that s^+ is protected at ℓ.[3] We will refer

[3] Syntactically, it appears that the proof constructors for $\boldsymbol{\alpha}_\preceq$ could be applied to types other than $\boldsymbol{\alpha}_\ell$ and s^+, but we prevent this via parametricity using the relational interpretation given in §5.

$\boxed{\preceq^+}$ proof constructors

$\mathbf{p_1} : \forall\beta_\ell{::}*.\,(\alpha_\preceq\ \beta_\ell\ \mathbf{1})$,

$\mathbf{p_\times} : \forall\beta_\ell{::}*.\,\forall\alpha_1{::}*.\,\forall\alpha_2{::}*.\ ((\alpha_\preceq\ \beta_\ell\ \alpha_1) \times (\alpha_\preceq\ \beta_\ell\ \alpha_2)) \to (\alpha_\preceq\ \beta_\ell\ (\alpha_1 \times \alpha_2))$,

$\mathbf{p_\to} : \forall\beta_\ell{::}*.\,\forall\alpha_1{::}*.\,\forall\alpha_2{::}*.\ (\alpha_\preceq\ \beta_\ell\ \alpha_2) \to (\alpha_\preceq\ \beta_\ell\ (\alpha_1 \to \alpha_2))$,

$\mathbf{p_{T_1}} : \forall\beta_\ell{::}*.\,\forall\beta_{\ell'}{::}*.\,\forall\alpha{::}*.\,(\alpha_\preceq\ \beta_\ell\ \alpha) \to (\alpha_\preceq\ \beta_\ell\ (\forall\beta{::}*.\,((\alpha_\preceq\ \beta_{\ell'}\ \beta) \times (\alpha \to \beta)) \to \beta))$

$\mathbf{p_{T_2}} : \forall\beta_\ell{::}*.\,\forall\beta_{\ell'}{::}*.\,\forall\alpha{::}*.\,(\beta_{\ell'} \to \beta_\ell) \to (\alpha_\preceq\ \beta_\ell\ (\forall\beta{::}*.\,((\alpha_\preceq\ \beta_{\ell'}\ \beta) \times (\alpha \to \beta)) \to \beta))$,

$\boxed{\mathbf{pf}[\![\ell \preceq \mathsf{s}]\!] : (\alpha_\preceq\ \alpha_\ell\ \mathsf{s}^+)}$ proof-term construction

$\mathbf{pf}[\![\ell \preceq 1]\!] \stackrel{\text{def}}{=} \mathbf{p_1}\ [\alpha_\ell]$

$\mathbf{pf}[\![\ell \preceq \mathsf{s}_1 \times \mathsf{s}_2]\!] \stackrel{\text{def}}{=} \mathbf{p_\times}\ [\alpha_\ell]\ [\mathsf{s}_1^+]\ [\mathsf{s}_2^+]\ \langle \mathbf{pf}[\![\ell \preceq \mathsf{s}_1]\!], \mathbf{pf}[\![\ell \preceq \mathsf{s}_2]\!] \rangle$

$\mathbf{pf}[\![\ell \preceq \mathsf{s}_1 \to \mathsf{s}_2]\!] \stackrel{\text{def}}{=} \mathbf{p_\to}\ [\alpha_\ell]\ [\mathsf{s}_1^+]\ [\mathsf{s}_2^+]\ \mathbf{pf}[\![\ell \preceq \mathsf{s}_2]\!]$

$\mathbf{pf}[\![\ell \preceq \mathsf{T}_{\ell'}\ \mathsf{s}]\!] \stackrel{\text{def}}{=} \mathbf{p_{T_1}}\ [\alpha_\ell]\ [\alpha_{\ell'}]\ [\mathsf{s}^+]\ \mathbf{pf}[\![\ell \preceq \mathsf{s}]\!]$ if $\ell \preceq \mathsf{s}$ and $\ell \not\sqsubseteq \ell'$

$\mathbf{p_{T_2}}\ [\alpha_\ell]\ [\alpha_{\ell'}]\ [\mathsf{s}^+]\ \mathbf{c}_{\ell'\ell}$ if $\ell \sqsubseteq \ell'$

Figure 4. F_ω: Protection Proofs

to this fully applied type $(\alpha_\preceq\ \alpha_\ell\ \mathsf{s}^+)$ 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 $\ell \preceq \mathsf{s}$ judgment. Each constructor is named suggesting the rule from the $\ell \preceq \mathsf{s}$ judgment which the constructor encodes. For instance, $\mathbf{p_1}$ encodes the rule for $\ell \preceq 1$, 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 $\mathbf{pf}[\![\ell \preceq \mathsf{s}]\!]$ for constructing protection proofs by induction on a given derivation of $\ell \preceq \mathsf{s}$. Note that $\mathbf{pf}[\![\ell \preceq \mathsf{s}]\!]$ yields the following lemma.

Lemma 4.1
If $\ell \preceq \mathsf{s}$ *then* $\exists \mathbf{m}.\ \mathcal{L}_\ell^+; \mathcal{L}_\sqsubseteq^+, \preceq^+ \vdash \mathbf{m} : (\alpha_\preceq\ \alpha_\ell\ \mathsf{s}^+)$

That is, if a source type is protected at some label ℓ, then a protection proof exists, namely $\mathbf{pf}[\![\ell \preceq \mathsf{s}]\!]$, for the translated type and label under $\mathcal{L}_\ell^+; \mathcal{L}_\sqsubseteq^+, \preceq^+$ (which we refer to as the *protection ADT*).

The translation judgment $\Gamma \vdash \mathsf{e} : \mathsf{s} \rightsquigarrow \mathbf{m}$ takes an open source term e of type s and produces the target term $\mathbf{m}$. The term $\mathbf{m}$ has type s^+ under the type environment $\mathcal{L}_\ell^+$ and the term environments $\mathcal{L}_\sqsubseteq^+$, $\preceq^+$, and Γ^+. We write Γ^+ to mean the point-wise translation of $\mathsf{x} : \mathsf{s} \in \Gamma$ to $\mathbf{x} : \mathsf{s}^+$.

Since DCC and F_ω share the same basic constructs, we translate most terms by structural recursion. The translation of an η_ℓ value expects a continuation and a protection proof, so the translation of bind must produce such a proof and continuation. Several translation rules are presented in Figure 5.

$\boxed{\Gamma \vdash \mathsf{e} : \mathsf{s} \rightsquigarrow \mathbf{m}}$ where $\mathcal{L}_\ell^+; \mathcal{L}_\sqsubseteq^+, \preceq^+, \Gamma^+ \vdash \mathbf{m} : \mathsf{s}^+$

$$\frac{}{\Gamma \vdash \langle\rangle : 1 \rightsquigarrow \langle\rangle} \qquad \frac{(\mathsf{x} : \mathsf{s}) \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{s} \rightsquigarrow \mathbf{x}} \qquad \frac{\Gamma, \mathsf{x} : \mathsf{s} \vdash \mathsf{e} : \mathsf{s}_2 \rightsquigarrow \mathbf{m}}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{s}_1.\, \mathsf{e} : \mathsf{s}_1 \to \mathsf{s}_2 \rightsquigarrow \lambda \mathbf{x}{:}\mathsf{s}_1^+.\, \mathbf{m}}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{s}_1 \to \mathsf{s}_2 \rightsquigarrow \mathbf{m_1} \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{s}_1 \rightsquigarrow \mathbf{m_2}}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : \mathsf{s}_2 \rightsquigarrow \mathbf{m_1}\ \mathbf{m_2}} \quad \ldots$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{s} \rightsquigarrow \mathbf{m} \qquad \text{where } \mathbf{t} = ((\alpha_\preceq\ \alpha_\ell\ \beta) \times (\mathsf{s}^+ \to \beta))}{\Gamma \vdash \eta_\ell\ \mathsf{e} : \mathsf{T}_\ell\ \mathsf{s} \rightsquigarrow \Lambda\beta{::}*.\lambda \mathbf{x}{:}\mathbf{t}.((\mathbf{pr}j_2\ \mathbf{x})\ \mathbf{m})}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_\ell\ \mathsf{s}_1 \rightsquigarrow \mathbf{m_1} \qquad \Gamma, \mathsf{x} : \mathsf{s}_1 \vdash \mathsf{e}_2 : \mathsf{s}_2 \rightsquigarrow \mathbf{m_2} \qquad \ell \preceq \mathsf{s}_2}{\Gamma \vdash \mathsf{bind}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 : \mathsf{s}_2 \rightsquigarrow \mathbf{m_1}\ [\mathsf{s}_2^+]\ \langle \mathbf{pf}[\![\ell \preceq \mathsf{s}_2]\!], (\lambda \mathbf{x}{:}\mathsf{s}_1^+.\, \mathbf{m_2}) \rangle}$$

Figure 5. DCC to F_ω: Term Translation (excerpts)

$$\begin{aligned}
Atom\,[\mathsf{s}] &= \{\, (\mathsf{e}_1, \mathsf{e}_2) \mid \vdash \mathsf{e}_1 : \mathsf{s} \ \wedge\ \vdash \mathsf{e}_2 : \mathsf{s} \,\} \\
\mathcal{V}[\![1]\!]_\zeta &= \{\, (\langle\rangle, \langle\rangle) \in Atom\,[1] \,\} \\
\mathcal{V}[\![\mathsf{s} \times \mathsf{s}']\!]_\zeta &= \{(\langle \mathsf{e}_1, \mathsf{e}_1' \rangle, \langle \mathsf{e}_2, \mathsf{e}_2' \rangle) \in Atom\,[\mathsf{s} \times \mathsf{s}'] \mid \\
&\qquad (\mathsf{e}_1, \mathsf{e}_2) \in \mathcal{E}[\![\mathsf{s}]\!]_\zeta \ \wedge\ (\mathsf{e}_1', \mathsf{e}_2') \in \mathcal{E}[\![\mathsf{s}']\!]_\zeta\} \\
\mathcal{V}[\![\mathsf{s} + \mathsf{s}']\!]_\zeta &= \{(\mathsf{inj}_1\ \mathsf{e}_1, \mathsf{inj}_1\ \mathsf{e}_2) \in Atom\,[\mathsf{s} + \mathsf{s}'] \mid (\mathsf{e}_1, \mathsf{e}_2) \in \mathcal{E}[\![\mathsf{s}]\!]_\zeta\} \\
&\quad \cup \{(\mathsf{inj}_2\ \mathsf{e}_1, \mathsf{inj}_2\ \mathsf{e}_2) \in Atom\,[\mathsf{s} + \mathsf{s}'] \mid (\mathsf{e}_1, \mathsf{e}_2) \in \mathcal{E}[\![\mathsf{s}']\!]_\zeta\} \\
\mathcal{V}[\![\mathsf{s}' \to \mathsf{s}]\!]_\zeta &= \{(\lambda \mathsf{x} : \mathsf{s}'.\, \mathsf{e}_1, \lambda \mathsf{x} : \mathsf{s}'.\, \mathsf{e}_2) \in Atom\,[\mathsf{s}' \to \mathsf{s}] \mid \\
&\qquad \forall (\mathsf{e}_1', \mathsf{e}_2') \in \mathcal{E}[\![\mathsf{s}']\!]_\zeta.(\mathsf{e}_1[\mathsf{e}_1'/\mathsf{x}], \mathsf{e}_2[\mathsf{e}_2'/\mathsf{x}]) \in \mathcal{E}[\![\mathsf{s}]\!]_\zeta\} \\
\mathcal{V}[\![\mathsf{T}_\ell\ \mathsf{s}]\!]_\zeta &= \{(\eta_\ell\ \mathsf{e}_1, \eta_\ell\ \mathsf{e}_2) \in Atom\,[\mathsf{T}_\ell\ \mathsf{s}] \mid \\
&\qquad \ell \sqsubseteq \zeta \implies (\mathsf{e}_1, \mathsf{e}_2) \in \mathcal{E}[\![\mathsf{s}]\!]_\zeta\} \\
\mathcal{E}[\![\mathsf{s}]\!]_\zeta &= \{(\mathsf{e}_1, \mathsf{e}_2) \in Atom\,[\mathsf{s}] \mid \exists \mathsf{v}_1, \mathsf{v}_2. \\
&\qquad \mathsf{e}_1 \longmapsto^* \mathsf{v}_1 \ \wedge\ \mathsf{e}_2 \longmapsto^* \mathsf{v}_2 \ \wedge\ (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\mathsf{s}]\!]_\zeta\} \\
\mathcal{G}[\![\cdot]\!]_\zeta &= (\emptyset, \emptyset) \\
\mathcal{G}[\![\Gamma, \mathsf{x} : \mathsf{s}]\!]_\zeta &= \{(\gamma_1\,[\mathsf{x} \mapsto \mathsf{e}_1], \gamma_2\,[\mathsf{x} \mapsto \mathsf{e}_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\zeta \ \wedge \\
&\qquad ((\mathsf{e}_1, \mathsf{e}_2) \in \mathcal{E}[\![\mathsf{s}]\!]_\zeta)\}
\end{aligned}$$

$$\Gamma \vdash \mathsf{e}_1 \approx_\zeta \mathsf{e}_2 : \mathsf{s} \stackrel{\text{def}}{=} \Gamma \vdash \mathsf{e}_1 : \mathsf{s} \ \wedge\ \Gamma \vdash \mathsf{e}_2 : \mathsf{s} \ \wedge\ \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\zeta.\ (\gamma_1(\mathsf{e}_1), \gamma_2(\mathsf{e}_2)) \in \mathcal{E}[\![\mathsf{s}]\!]_\zeta$$

Figure 6. DCC: Logical Relation

Lemma 4.2 (Translation preserves well-typedness)
If $\Gamma \vdash \mathsf{e} : \mathsf{s}$ *then* $\Gamma \vdash \mathsf{e} : \mathsf{s} \rightsquigarrow \mathbf{m}$ *and* $\mathcal{L}_\ell^+; \mathcal{L}_\sqsubseteq^+, \preceq^+, \Gamma^+ \vdash \mathbf{m} : \mathsf{s}^+$.

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_ω and then formalize an observer-sensitive equivalence for F_ω using higher-order parametricity.

5.1 Logical Relation for DCC

The logical relation for DCC, written $\Gamma \vdash \mathsf{e}_1 \approx_\zeta \mathsf{e}_2 : \mathsf{s}$, says that e_1 and e_2 appear equivalent from the perspective of an observer at level ζ. The relation is defined in Figure 6 and enforces that an observer whose label in the lattice is ζ cannot distinguish data protected at a lattice level higher than (or incomparable to) ζ.

The logical relation is defined by structural recursion on types. The value relation $\mathcal{V}[\![\mathsf{s}]\!]_\zeta$ relates closed values at type s. We use the relation $Atom\,[\mathsf{s}]$ to ensure that the logical relation relates only well-typed terms.

The value $\langle\rangle$ appears equivalent to itself at type 1 to an observer at any level. Sums $\mathsf{inj}_1\ \mathsf{e}_1$ and $\mathsf{inj}_1\ \mathsf{e}_2$ appear equivalent at type $\mathsf{s} + \mathsf{s}'$ to an observer at level ζ 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 $\eta_\ell\ \mathsf{e}_1$ and $\eta_\ell\ \mathsf{e}_2$ are related at type $\mathsf{T}_\ell\ \mathsf{s}$ if the observer ζ is lower than ℓ, 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 ℓ is unable to distinguish $\eta_\ell\ \mathsf{e}_1$ from $\eta_\ell\ \mathsf{e}_2$.

The relation $\mathcal{E}[\![\mathsf{s}]\!]_\zeta$ 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 γ_1 and γ_2 that map variables to terms related at the corresponding types in Γ, and requiring that the closed terms $\gamma_1(\mathsf{e}_1)$ and $\gamma_2(\mathsf{e}_2)$ 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 ℓ are related if the observer ζ is lower than or incomparable to ℓ. This property, stated in Lemma 5.1 is similar to Abadi *et al.*'s Proposition 3.2 [2].

$$
\begin{array}{lcl}
\mathit{Atom}\,[\mathbf{t}_1,\mathbf{t}_2]^{\mathbf{D};\mathbf{G}} & = & \{(\mathbf{m}_1,\mathbf{m}_2) \mid \mathbf{D} \vdash \mathbf{t}_1 \wedge \mathbf{D} \vdash \mathbf{t}_2 \wedge \\
 & & \quad \mathbf{D};\mathbf{G} \vdash \mathbf{m}_1 : \mathbf{t}_1 \wedge \mathbf{D};\mathbf{G} \vdash \mathbf{m}_2 : \mathbf{t}_2\} \\
\mathit{Atom}\,[\mathbf{t}]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \mathit{Atom}\,[\rho_1(\mathbf{t}),\rho_2(\mathbf{t})]^{\mathbf{D};\mathbf{G}} \\
\mathit{Rel}^{\mathbf{D};\mathbf{G}}_{*} & = & \{(\mathbf{t}_1,\mathbf{t}_2,\mathbf{R}) \mid \mathbf{R} \subseteq \mathit{Atom}\,[\mathbf{t}_1,\mathbf{t}_2]^{\mathbf{D};\mathbf{G}}\} \\
\mathit{Rel}^{\mathbf{D};\mathbf{G}}_{\kappa \to \kappa'} & = & \{(\mathbf{t}_1,\mathbf{t}_2,\mathbf{R}) \mid (\forall \boldsymbol{\pi} \in \mathit{Rel}^{\mathbf{D};\mathbf{G}}_{\kappa}. \\
 & & \quad (\mathbf{t}_1\,\boldsymbol{\pi}_1,\ \mathbf{t}_2\,\boldsymbol{\pi}_2,\ (\mathbf{R}\,\boldsymbol{\pi})) \in \mathit{Rel}^{\mathbf{D};\mathbf{G}}_{\kappa'} \wedge \\
 & & \quad (\forall \boldsymbol{\pi}' \in \mathit{Rel}^{\mathbf{D};\mathbf{G}}_{\kappa}.\ \boldsymbol{\pi} \equiv^{\mathbf{D};\mathbf{G}}_{\kappa} \boldsymbol{\pi}' \implies \\
 & & \quad\quad \mathbf{R}\,\boldsymbol{\pi} \equiv^{\mathbf{D};\mathbf{G}}_{\kappa'} \mathbf{R}\,\boldsymbol{\pi}')\}
\end{array}
$$

$$
\begin{array}{lcl}
\boldsymbol{\pi} \equiv^{\mathbf{D};\mathbf{G}}_{\kappa} \boldsymbol{\pi} & \stackrel{\text{def}}{=} & \boldsymbol{\pi}_1 \equiv \boldsymbol{\pi}'_1 \wedge \boldsymbol{\pi}_2 \equiv \boldsymbol{\pi}'_2 \wedge \boldsymbol{\pi}_{\mathbf{R}} \equiv^{\mathbf{D};\mathbf{G}}_{\kappa} \boldsymbol{\pi}'_{\mathbf{R}} \\
\mathbf{R} \equiv^{\mathbf{D};\mathbf{G}}_{*} \mathbf{R}' & \stackrel{\text{def}}{=} & (\mathbf{m}_1,\mathbf{m}_2) \in \mathbf{R} \iff (\mathbf{m}_1,\mathbf{m}_2) \in \mathbf{R}' \\
\mathbf{R} \equiv^{\mathbf{D};\mathbf{G}}_{\kappa_1 \to \kappa_2} \mathbf{R}' & \stackrel{\text{def}}{=} & \forall \boldsymbol{\pi} \in \mathit{Rel}^{\mathbf{D};\mathbf{G}}_{\kappa_1}.\ \mathbf{R}\,\boldsymbol{\pi} \equiv^{\mathbf{D};\mathbf{G}}_{\kappa_2} \mathbf{R}'\,\boldsymbol{\pi}
\end{array}
$$

$$
\begin{array}{lcl}
\boldsymbol{\mathcal{V}}[\![\boldsymbol{\alpha}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \{(\mathbf{m}_1,\mathbf{m}_2) \in \rho_{\mathbf{R}}(\boldsymbol{\alpha})\} \\
 & \vdots & \\
\boldsymbol{\mathcal{V}}[\![\mathbf{t}' \to \mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \{(\lambda \mathbf{x}{:}\mathbf{t}'_1.\mathbf{m}_1, \lambda \mathbf{x}{:}\mathbf{t}'_2.\mathbf{m}_2) \in \mathit{Atom}\,[\mathbf{t}' \to \mathbf{t}]^{\mathbf{D};\mathbf{G}}_{\rho} \mid \\
 & & \quad \forall (\mathbf{m}'_1,\mathbf{m}'_2) \in \boldsymbol{\mathcal{E}}[\![\mathbf{t}']\!]^{\mathbf{D};\mathbf{G}}_{\rho}. \\
 & & \quad\quad (\mathbf{m}_1[\mathbf{m}'_1/\mathbf{x}], \mathbf{m}_2[\mathbf{m}'_2/\mathbf{x}]) \in \boldsymbol{\mathcal{E}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}\} \\
\boldsymbol{\mathcal{V}}[\![\forall \boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\,\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \{(\boldsymbol{\Lambda}\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{m}_1, \boldsymbol{\Lambda}\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{m}_2) \in \mathit{Atom}\,[\forall \boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\,\mathbf{t}]^{\mathbf{D};\mathbf{G}}_{\rho} \mid \\
 & & \quad \forall \boldsymbol{\pi} \in \mathit{Rel}^{\mathbf{D};\mathbf{G}}_{\kappa}. \\
 & & \quad\quad (\mathbf{m}_1[\boldsymbol{\pi}_1/\boldsymbol{\alpha}], \mathbf{m}_2[\boldsymbol{\pi}_2/\boldsymbol{\alpha}]) \in \boldsymbol{\mathcal{E}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho[\boldsymbol{\alpha}::\boldsymbol{\kappa} \mapsto \boldsymbol{\pi}]}\} \\
\boldsymbol{\mathcal{V}}[\![\mathbf{t}\ \mathbf{t}']\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \boldsymbol{\mathcal{T}}[\![\mathbf{t}\ \mathbf{t}' :: *]\!]^{\mathbf{D};\mathbf{G}}_{\rho} \\
\boldsymbol{\mathcal{E}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \{(\mathbf{m}_1,\mathbf{m}_2) \in \mathit{Atom}\,[\mathbf{t}]^{\mathbf{D};\mathbf{G}}_{\rho} \mid \\
 & & \quad \exists \mathbf{m}'_1,\mathbf{m}'_2.\ \mathbf{m}_1 \longmapsto^* \mathbf{m}'_1 \wedge \mathbf{m}_2 \longmapsto^* \mathbf{m}'_2 \wedge \\
 & & \quad \mathrm{irred}(\mathbf{m}'_1) \wedge \mathrm{irred}(\mathbf{m}'_2) \wedge (\mathbf{m}'_1,\mathbf{m}'_2) \in \boldsymbol{\mathcal{V}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}\} \\
\boldsymbol{\mathcal{D}}[\![\cdot]\!]^{\mathbf{D};\mathbf{G}} & = & \emptyset \\
\boldsymbol{\mathcal{D}}[\![\boldsymbol{\Delta},\boldsymbol{\alpha}::\boldsymbol{\kappa}]\!]^{\mathbf{D};\mathbf{G}} & = & \{\rho\,[\boldsymbol{\alpha} :: \boldsymbol{\kappa} \mapsto \boldsymbol{\pi}] \mid \rho \in \boldsymbol{\mathcal{D}}[\![\boldsymbol{\Delta}]\!]^{\mathbf{D};\mathbf{G}} \wedge \boldsymbol{\pi} \in \mathit{Rel}^{\mathbf{D};\mathbf{G}}_{\kappa}\} \\
\boldsymbol{\mathcal{G}}[\![\cdot]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & (\emptyset,\emptyset) \\
\boldsymbol{\mathcal{G}}[\![\boldsymbol{\Gamma},\mathbf{x} : \mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \{(\boldsymbol{\gamma}_1\,[\mathbf{x} \mapsto \mathbf{m}_1], \boldsymbol{\gamma}_2\,[\mathbf{x} \mapsto \mathbf{m}_2]) \mid \\
 & & \quad (\boldsymbol{\gamma}_1,\boldsymbol{\gamma}_2) \in \boldsymbol{\mathcal{G}}[\![\boldsymbol{\Gamma}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} \wedge (\mathbf{m}_1,\mathbf{m}_2) \in \boldsymbol{\mathcal{E}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}\}
\end{array}
$$

$$
\begin{array}{lcl}
\boldsymbol{\mathcal{T}}[\![\mathbf{t} :: *]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \boldsymbol{\mathcal{V}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} \\
\quad \text{if } \mathbf{t} \in \{\mathbf{1}, \boldsymbol{\alpha}, \mathbf{t}_1 + \mathbf{t}_2, \mathbf{t}_1 \times \mathbf{t}_2, \mathbf{t}_1 \to \mathbf{t}_2, \forall \boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\,\mathbf{t}\} & & \\
\boldsymbol{\mathcal{T}}[\![\boldsymbol{\alpha} :: \boldsymbol{\kappa}_1 \to \boldsymbol{\kappa}_2]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \rho_{\mathbf{R}}(\boldsymbol{\alpha}) \\
\boldsymbol{\mathcal{T}}[\![\boldsymbol{\lambda}\boldsymbol{\alpha}{::}\boldsymbol{\kappa}_1.\mathbf{t} :: \boldsymbol{\kappa}_1 \to \boldsymbol{\kappa}_2]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & \lambda_R \boldsymbol{\pi}.\boldsymbol{\mathcal{T}}[\![\mathbf{t} :: \boldsymbol{\kappa}_2]\!]^{\mathbf{D};\mathbf{G}}_{\rho[\boldsymbol{\alpha}::\boldsymbol{\kappa}_1 \mapsto \boldsymbol{\pi}]} \\
\boldsymbol{\mathcal{T}}[\![\mathbf{t}\ \mathbf{t}' :: \boldsymbol{\kappa}_2]\!]^{\mathbf{D};\mathbf{G}}_{\rho} & = & (\boldsymbol{\mathcal{T}}[\![\mathbf{t} :: \boldsymbol{\kappa}_1 \to \boldsymbol{\kappa}_2]\!]^{\mathbf{D};\mathbf{G}}_{\rho}\ \mathbf{t}', \rho_2\ \mathbf{t}', \boldsymbol{\mathcal{T}}[\![\mathbf{t}' :: \boldsymbol{\kappa}_1]\!]^{\mathbf{D};\mathbf{G}}_{\rho})) \\
 & & \quad (\rho_1\ \mathbf{t}', \rho_2\ \mathbf{t}', \boldsymbol{\mathcal{T}}[\![\mathbf{t}' :: \boldsymbol{\kappa}_1]\!]^{\mathbf{D};\mathbf{G}}_{\rho}))
\end{array}
$$

$$
\boldsymbol{\Delta};\boldsymbol{\Gamma} \vdash \mathbf{m}_1 \approx \mathbf{m}_2 : \mathbf{t} \stackrel{\text{def}}{=} \boldsymbol{\Delta};\boldsymbol{\Gamma} \vdash \mathbf{m}_1 : \mathbf{t} \wedge \boldsymbol{\Delta};\boldsymbol{\Gamma} \vdash \mathbf{m}_2 : \mathbf{t} \wedge \forall \mathbf{D},\mathbf{G},\rho,\boldsymbol{\gamma}_1,\boldsymbol{\gamma}_2.\ \mathrm{dom}(\mathbf{D}) \# \mathrm{dom}(\boldsymbol{\Delta}) \wedge \mathrm{dom}(\mathbf{G}) \# \mathrm{dom}(\boldsymbol{\Gamma}) \wedge \rho \in \boldsymbol{\mathcal{D}}[\![\boldsymbol{\Delta}]\!]^{\mathbf{D};\mathbf{G}} \wedge (\boldsymbol{\gamma}_1,\boldsymbol{\gamma}_2) \in \boldsymbol{\mathcal{G}}[\![\boldsymbol{\Gamma}]\!]^{\mathbf{D};\mathbf{G}}_{\rho} \implies (\rho_1(\boldsymbol{\gamma}_1(\mathbf{m}_1)), \rho_2(\boldsymbol{\gamma}_2(\mathbf{m}_2))) \in \boldsymbol{\mathcal{E}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}
$$

Figure 7. F_ω: Logical Relation (excerpts)

Lemma 5.1
If $\ell \preceq \mathsf{s}$ *and* $\ell \not\preceq \zeta$ *then* $\forall (\mathsf{e}_1,\mathsf{e}_2) \in \mathit{Atom}\,[\mathsf{s}].(\mathsf{e}_1,\mathsf{e}_2) \in \mathcal{E}[\![\mathsf{s}]\!]_\zeta$

Theorem 5.2 (Noninterference)
If $\Gamma \vdash \mathsf{e} : \mathsf{s}$ *then* $\forall \zeta.\Gamma \vdash \mathsf{e} \approx_\zeta \mathsf{e} : \mathsf{s}$

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_ω

We formalize equivalence in F_ω via an *open* logical relation whose structure resembles that of the logical relation for R_ω 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 [22] open logical relation for a linear System F. Specifically, our value relation $\boldsymbol{\mathcal{V}}$ and term relation $\boldsymbol{\mathcal{E}}$ for F_ω may relate open terms with open types, unlike the $\mathcal{V}$ and $\mathcal{E}$ relations for 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 $\mathbf{m}_1$ and $\mathbf{m}_2$ related by the F_ω term relation $\boldsymbol{\mathcal{E}}$. We must show that we can back-translate these to two related source terms e_1 and e_2. In order to back-translate $\mathbf{m}_1$ and $\mathbf{m}_2$, we must know they have a translation type, since only terms of translation type can be back-translated (see §7.1). However, our translation produces types that contain free variables $\boldsymbol{\alpha}_{\preceq}$ and $\boldsymbol{\alpha}_\ell$, and terms with the free variables from $\preceq^+$. If our F_ω relation closed these free variables, as is standard, then terms that belong to the relation $\boldsymbol{\mathcal{E}}$ will not have translation type, and we will not be able to back-translate them.

Figure 7 presents the open logical relation for F_ω. As usual, the top-level logical relation $\boldsymbol{\Delta};\boldsymbol{\Gamma} \vdash \mathbf{m}_1 \approx \mathbf{m}_2 : \mathbf{t}$ requires that we close all the free variables of $\boldsymbol{\Delta}$ and $\boldsymbol{\Gamma}$ in $\mathbf{m}_1$ and $\mathbf{m}_2$ choosing a relational type interpretation ρ and related term substitutions $\boldsymbol{\gamma}_1$ and $\boldsymbol{\gamma}_2$. However, in contrast to a standard logical relation, such as our logical relation for DCC, these substitutions contain terms that may be open with respect to a fresh type environment $\mathbf{D}$ and term environment $\mathbf{G}$. By fresh, we mean the domains of the environments $\mathbf{D}$ and $\mathbf{G}$ are disjoint from $\boldsymbol{\Delta}$ and $\boldsymbol{\Gamma}$, written $\mathrm{dom}(\mathbf{D})\#\mathrm{dom}(\boldsymbol{\Delta})$ and $\mathrm{dom}(\mathbf{G})\#\mathrm{dom}(\boldsymbol{\Gamma})$. 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 §5.3, 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 $\mathcal{L}^+_\ell$.

We define a value relation $\boldsymbol{\mathcal{V}}$ and a term relation $\boldsymbol{\mathcal{E}}$ which are inhabited by *open* terms that are well-typed under $\mathbf{D};\mathbf{G}$. The relations are indexed by a type $\mathbf{t}$ such that $\boldsymbol{\Delta} \vdash \mathbf{t} :: \boldsymbol{\kappa}$. Hence, the relations need to be parameterized by a relational interpretation ρ which maps the free type variables $\boldsymbol{\alpha}$ in $\boldsymbol{\Delta}$ to triples $(\mathbf{t}_1,\mathbf{t}_2,\mathbf{R})$ (abbreviated $\boldsymbol{\pi}$). The types $\mathbf{t}_1$ and $\mathbf{t}_2$ must be well-formed under $\mathbf{D}$ rather than $\boldsymbol{\Delta}$, and are the types we substitute for $\boldsymbol{\alpha}$ in pairs of related terms. We write $\boldsymbol{\pi}_1$ and $\boldsymbol{\pi}_2$ to denote the projections of $\mathbf{t}_1$ and $\mathbf{t}_2$ from $\boldsymbol{\pi}$, and $\boldsymbol{\pi}_{\mathbf{R}}$ to denote the projection of $\mathbf{R}$ from $\boldsymbol{\pi}$. We extend this notation to ρ; if $\rho = [\boldsymbol{\alpha}_1 :: \boldsymbol{\kappa}_1 \mapsto \boldsymbol{\pi}^1] \ldots [\boldsymbol{\alpha}_n :: \boldsymbol{\kappa}_n \mapsto \boldsymbol{\pi}^n]$, then $\rho_1 = [\boldsymbol{\alpha}_1 \mapsto \boldsymbol{\pi}^1_1] \ldots [\boldsymbol{\alpha}_n \mapsto \boldsymbol{\pi}^n_1]$, and we analogously use ρ_2 and $\rho_{\mathbf{R}}$. Finally, we write $\rho_1(\mathbf{m})$ to denote applying the substitution ρ_1 to all the type variables in $\mathbf{m}$. We use similar notation for application of other substitutions to terms and types.

The relation $\boldsymbol{\mathcal{E}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}$ runs terms until they are irreducible, and then requires that the irreducible terms be related in $\boldsymbol{\mathcal{V}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}$. Note that unless $\mathbf{t}$ is $\boldsymbol{\alpha}$, the terms must reduce to values of the appropriate canonical form for type $\mathbf{t}$. If $\mathbf{t}$ is $\boldsymbol{\alpha}$, $\rho_{\mathbf{R}}(\boldsymbol{\alpha})$ can be a relation on terms with free (term and type) variables and $\boldsymbol{\mathcal{V}}[\![\boldsymbol{\alpha}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}$ relates terms that are not values. This follows the formalism by Zhao *et al.* [22].

As usual, when relating the values $\boldsymbol{\Lambda}\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{m}_1$ and $\boldsymbol{\Lambda}\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{m}_2$ in $\boldsymbol{\mathcal{V}}[\![\forall \boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\,\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}$, we consider arbitrary types $\mathbf{t}_1$ and $\mathbf{t}_2$ and a relation object $\mathbf{R}$. At kind $*$, relation objects are just sets of terms. At kind $\boldsymbol{\kappa}_1 \to \boldsymbol{\kappa}_2$, relation objects are relation-level functions. Intuitively, relation-level functions take relations as inputs and produce relations as outputs. Formally, relation-level functions written $\lambda_R \boldsymbol{\pi}.\mathbf{R}$ take triples $\boldsymbol{\pi}$ and produce relation objects. We write $(\mathbf{R}\,\boldsymbol{\pi})$ as the application of the relation-level function $\mathbf{R}$ to $\boldsymbol{\pi}$.

The relation objects for all types, including the higher-kinded types, are defined inductively on the judgment $\boldsymbol{\Delta} \vdash \mathbf{t} :: \boldsymbol{\kappa}$ by $\boldsymbol{\mathcal{T}}[\![\mathbf{t} :: \boldsymbol{\kappa}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}$. When $\mathbf{t}$ has kind $*$, this is just the relation $\boldsymbol{\mathcal{V}}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_{\rho}$.

$$
\begin{array}{l}
\mathcal{L}^+_\ell; \mathcal{L}^+_\sqsubseteq, \preceq^+, \Gamma^+ \vdash \mathbf{m_1} \approx_\zeta \mathbf{m_2} : \mathsf{s}^+ \stackrel{\text{def}}{=} \\
\quad \mathcal{L}^+_\ell; \mathcal{L}^+_\sqsubseteq, \preceq^+, \Gamma^+ \vdash \mathbf{m_1} : \mathsf{s}^+ \wedge \mathcal{L}^+_\ell; \mathcal{L}^+_\sqsubseteq, \preceq^+, \Gamma^+ \vdash \mathbf{m_1} : \mathsf{s}^+ \wedge \\
\quad \text{let } \rho = [\![\mathcal{L}^+_\ell]\!]^{\Sigma}_\zeta, \gamma_\sqsubseteq = [\![\mathcal{L}^+_\sqsubseteq]\!], \gamma_\preceq = [\![\preceq^+]\!] \text{in} \\
\quad \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma^+]\!]^{\Sigma}_\rho . \\
\quad\quad (\rho_1(\gamma_\sqsubseteq(\gamma_\preceq(\gamma_1(\mathbf{m_1})))), \rho_2(\gamma_\sqsubseteq(\gamma_\preceq(\gamma_2(\mathbf{m_2}))))) \in \mathcal{E}[\![\mathsf{s}^+]\!]^{\Sigma}_\rho
\end{array}
$$

Figure 8. F_ω: Observer-Sensitive Logical Relation

$$
\begin{array}{l}
\Sigma \;\; = \mathbf{D}_\ell; \mathbf{G}_\ell, \mathbf{G}_\preceq \qquad \text{Note } \Sigma_\mathbf{D} = \mathbf{D}_\ell \text{ and } \Sigma_\mathbf{G} = \mathbf{G}_\ell, \mathbf{G}_\preceq \\
\mathbf{D}_\ell = \{\hat{\alpha}_\ell :: * \mid \ell \in \mathcal{L}_\ell\} \cup \{\hat{\alpha}_\preceq :: * \to * \to *\} \\
\mathbf{G}_\ell = \{\hat{\mathsf{c}}_{\ell\ell'} : \hat{\alpha}_\ell \to \hat{\alpha}_{\ell'} \mid \ell' \sqsubseteq \ell \in \mathcal{L}_\sqsubseteq\} \\
\mathbf{G}_\preceq = \{\hat{\mathbf{p}}_1 : \forall \beta_\ell :: *. (\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \mathbf{1}), \\
\quad \hat{\mathbf{p}}_\times : \forall \beta_\ell :: *. \forall \alpha_1 :: *. \forall \alpha_2 :: *. \\
\quad\quad ((\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \alpha_1) \times (\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \alpha_2)) \to (\hat{\alpha}_\preceq \;\; \beta_\ell \;\; (\alpha_1 \times \alpha_2)), \\
\quad \hat{\mathbf{p}}_\to : \forall \beta_\ell :: *. \forall \alpha_1 :: *. \forall \alpha_2 :: *. (\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \alpha_2) \to \\
\quad\quad (\hat{\alpha}_\preceq \;\; \beta_\ell \;\; (\alpha_1 \to \alpha_2)), \\
\quad \hat{\mathbf{p}}_{\mathbf{T}_1} : \forall \beta_\ell :: *. \forall \beta_{\ell'} :: *. (\hat{\alpha}_\preceq \;\; \beta_{\ell'} \;\; \mathbf{t}) \to \\
\quad\quad (\hat{\alpha}_\preceq \;\; \hat{\alpha}_\ell \;\; (\forall \beta :: *. ((\alpha_\preceq \;\; \beta_{\ell'} \;\; \beta) \times (\mathbf{t} \to \beta)) \to \beta)) \\
\quad \hat{\mathbf{p}}_{\mathbf{T}_2} : \forall \beta_\ell :: *. \forall \beta_{\ell'} :: *. \forall \alpha :: *. (\beta_{\ell'} \to \beta_\ell) \to \\
\quad\quad (\hat{\alpha}_\preceq \;\; \beta_\ell \;\; (\forall \beta :: *. ((\alpha_\preceq \;\; \beta_{\ell'} \;\; \beta) \times (\alpha \to \beta)) \to \beta))\}
\end{array}
$$

Figure 9. F_ω: Open Protection ADT

For type-level functions $\lambda\alpha::\kappa.\mathbf{t}$, we construct a relation-level function that takes a triple π and produces a relation object where ρ is extended to map α to π. For type application $\mathbf{t}\ \mathbf{t}' :: \kappa_2$, we apply the inductively defined relation-level function for $\mathbf{t} :: \kappa_1 \to \kappa_2$ to the closed types $\rho_1(\mathbf{t}')$, $\rho_2(\mathbf{t}')$, and the inductively defined relation object for $\mathbf{t}' :: \kappa_1$.

The set $Rel^{\mathbf{D};\mathbf{G}}_\kappa$ defines well-formed relations. Formally, $Rel^{\mathbf{D};\mathbf{G}}_\kappa$ contains triples π where $\pi_\mathbf{R}$ is a relation object defined on types π_1 and π_2. For kind $*$, a relation is well-formed if it relates well-typed terms. For higher kinds, a relation object $\mathbf{R}$ is well-formed if, for equivalent inputs π and π', it produces equivalent outputs $\mathbf{R}\ \pi$ and $\mathbf{R}\ \pi'$.

We prove the fundamental property of this logical relation, stated in Theorem 5.3 (Parametricity). The proof essentially follows that of Vytiniotis and Weirich [20].

Theorem 5.3 (Parametricity)
If $\Delta; \Gamma \vdash \mathbf{m} : \mathbf{t}$ *then* $\Delta; \Gamma \vdash \mathbf{m} \approx \mathbf{m} : \mathbf{t}$

5.3 Observer-Sensitive Relation for F_ω

To prove that *observer-sensitive equivalence* is preserved, we need an observer-sensitive relation for F_ω. Recall that the DCC logical relation is indexed by an observer ζ, but so far the F_ω relation is simply $\Delta; \Gamma \vdash \mathbf{m_1} \approx \mathbf{m_2} : \mathbf{t}$. In Figure 8, we define the relation $\mathcal{L}^+_\ell; \mathcal{L}^+_\sqsubseteq, \preceq^+, \Gamma^+ \vdash \mathbf{m_1} \approx_\zeta \mathbf{m_2} : \mathsf{s}^+$ referenced in §3. We explain the interpretations $[\![\mathcal{L}^+_\ell]\!]^{\Sigma}_\zeta$, $[\![\mathcal{L}^+_\sqsubseteq]\!]$ and $[\![\preceq^+]\!]$ in detail shortly. For now, note that we pick the relational interpretation ρ based on the observer ζ. This is where we use parametricity to encode the notion of an observer and the properties necessary to preserve noninterference.

Recall that we must be able to back-translate terms given only that they are in the $\mathcal{E}[\![\mathbf{t}]\!]^{\mathbf{D};\mathbf{G}}_\rho$ relation. We need to pick a particular $\mathbf{D}; \mathbf{G}$ that allows us to implement the protection ADT and still identify translation types. In Figure 9 we provide an *open* protection ADT $\mathbf{D}_\ell; \mathbf{G}_\ell, \mathbf{G}_\preceq$, abbreviated Σ. This open ADT is simply an alpha-renaming of our protection ADT, adding a hat symbol ($\hat{}$) to each name. This satisfies the freshness condition for $\mathbf{D}; \mathbf{G}$ and we can still identify translation types.

In Figure 10, we provide implementations of $\preceq^+$ and $\mathcal{L}^+_\sqsubseteq$, written $[\![\preceq^+]\!]$ and $[\![\mathcal{L}^+_\sqsubseteq]\!]$ in terms of the open ADT Σ. Recall that the $\mathcal{V}$ relation for F_ω only relates stuck terms at abstract types. We ensure the new open constructors can only appear fully applied by

$$
\begin{array}{l}
[\![\mathcal{L}^+_\sqsubseteq]\!] = \{\, \mathbf{c}_{\ell\ell'} \mapsto \lambda \mathbf{x}{:}\hat{\alpha}_\ell.\hat{\mathbf{c}}_{\ell\ell'}\ \mathbf{x} \mid \ell' \sqsubseteq \ell \in \mathcal{L}_\sqsubseteq \,\} \\
[\![\preceq^+]\!] = \{\mathbf{p}_1 \mapsto \Lambda\beta_\ell::*.\hat{\mathbf{p}}_1\ [\beta_\ell], \\
\qquad \mathbf{p}_\times \mapsto \Lambda\beta_\ell::*.\Lambda\alpha_1::*.\Lambda\alpha_2::*. \\
\qquad\quad \lambda \mathbf{x}{:}((\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \alpha_1) \times (\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \alpha_2)). \\
\qquad\quad\quad \hat{\mathbf{p}}_\times\ [\beta_\ell]\ [\alpha_1]\ [\alpha_2]\ \mathbf{x}, \\
\qquad \cdots \}
\end{array}
$$

Figure 10. F_ω: Impl. of Coercions & Proof Constructors (excerpts)

$$
\begin{array}{l}
[\![\mathcal{L}^+_\ell]\!]^{\Sigma}_\zeta = \{\alpha_\ell :: * \mapsto (\hat{\alpha}_\ell, \hat{\alpha}_\ell, \mathit{Atom}\,[\hat{\alpha}_\ell, \hat{\alpha}_\ell]^{\Sigma}) \mid \ell \in \mathcal{L}_\ell\} \\
\qquad \cup \\
\qquad \{\alpha_\preceq :: * \to * \to * \mapsto \\
\qquad\quad (\lambda\beta_\ell::*.\lambda\beta::*.(\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \beta),\ \lambda\beta_\ell::*.\lambda\beta::*.(\hat{\alpha}_\preceq \;\; \beta_\ell \;\; \beta), \\
\qquad\quad \lambda_R(\mathbf{t}_1, \mathbf{t}_2, \mathbf{R}_\ell).\lambda_R(\mathbf{t}'_1, \mathbf{t}'_2, \mathbf{R}_\beta). \\
\qquad\quad\quad \{(\mathbf{m_1}, \mathbf{m_2}) \in \mathit{Atom}\,[(\hat{\alpha}_\preceq \;\; \mathbf{t}_1 \;\; \mathbf{t}'_1), (\hat{\alpha}_\preceq \;\; \mathbf{t}_2 \;\; \mathbf{t}'_2)]^{\Sigma} \mid \\
\qquad\quad\quad\quad \mathbf{t}_1 = \hat{\alpha}_\ell \wedge \mathbf{t}_2 = \hat{\alpha}_\ell \wedge \\
\qquad\quad\quad\quad \exists \mathsf{s}^+_1.\mathbf{t}'_1 = \mathsf{s}^+_1 \wedge \ell \preceq \mathsf{s}_1 \wedge \exists \mathsf{s}^+_2.\mathbf{t}'_2 = \mathsf{s}^+_2 \wedge \ell \preceq \mathsf{s}_2 \wedge \\
\qquad\quad\quad\quad (\ell \not\sqsubseteq \zeta \implies \mathbf{R}_\beta = \mathit{Atom}\,[\mathbf{t}'_1, \mathbf{t}'_2]^{\Sigma})\}\}
\end{array}
$$

Figure 11. F_ω: Relational Interpretation of Labels and $\alpha_\preceq$

implementing the original constructors as eta-expansions of the new constructors.

In Figure 11, we build the notion of an observer into the interpretation of $\alpha_\preceq$. In particular, we build in the key property given by Lemma 5.1. The relation on $\alpha_\preceq$ requires that if the observer is too low, then every well-typed term of the protected type must be related. Since $\alpha_\preceq$ is a higher-kinded type, we encode this property using a relation-level function. When the observer is too low, we require that all well-typed terms are in $\mathbf{R}_\beta$—the relation given for terms of the protected type. The relation also enforces that a protection proof, *i.e.*, a term of type $(\alpha_\preceq \;\; \alpha_\ell \;\; \mathsf{s}^+)$, actually implies $\ell \preceq \mathsf{s}$. We can only state this condition since each $\hat{\alpha}_\ell$ is left free. That is, by leaving the types $\hat{\alpha}_\ell$ free, we are able to interpret each $\hat{\alpha}_\ell$ as new a base type encoding the lattice labels from DCC. Using these new base types, we can define a relational interpretation for $\alpha_\preceq$ that encodes the key property needed to show that noninterference is preserved.

These requirements turn into proof obligations to users of a protected expression: to produce related protection proofs, you must prove that low observers cannot distinguish protected terms by providing an $\mathbf{R}_\beta$ that relates all appropriately typed terms if the observer is too low. Hence, the existence of a protection proof corresponds to the DCC protection judgment.

Thus we have encoded, using parametricity, the requirement of noninterference in F_ω: a low observer cannot distinguish protected terms. That is, using this relational interpretation, any proof that a type is protected also proves that, if the observer is not permitted to see the protected type then any two terms of that type are indistinguishable.

6. Translation Preserves Semantics

To prove that the translation preserves semantics, like Tse and Zdancewic, we define a *cross-language logical relation* that relates source terms of type s to target terms of type s^+. The relation is defined by induction on source types s.

The cross-language relation is defined in Figure 12. The relation $\mathcal{V}^+_\zeta[\![\mathsf{s}]\!]^{\Sigma}_\delta$ specifies when a DCC value $\mathsf{v} : \mathsf{s}$ is related (*i.e.*, semantically equivalent) to an F_ω value $\Sigma \vdash \mathbf{u} : \delta(\mathsf{s}^+)$. The relation is parametrized by our open protection ADT Σ. (We write $\Sigma_\mathbf{D}$ to refer to the $\mathbf{D}_\ell$ component of Σ and $\Sigma_\mathbf{G}$ to refer to the components $\mathbf{G}_\ell, \mathbf{G}_\preceq$.) The type substitution δ maps types from our protection ADT to corresponding types in the open protection ADT. Note that the relation is also indexed by an observer ζ. This seems strange since clearly *semantic* equivalence should be independent of an observer. We discuss this issue below.

Most values are related in the obvious way: $\langle\rangle$ is related to $\langle\rangle$, pairs are related if they contain related components, and functions

$$\eta_{\mathbf{k}}^{\ell,\mathsf{s}} \stackrel{\text{def}}{=} \lambda \mathbf{y}{:}\mathsf{s}^+.\Lambda\boldsymbol{\beta}{::}*.\lambda \mathbf{x}{:}((\boldsymbol{\alpha}_{\preceq}\ \boldsymbol{\alpha}_\ell\ \boldsymbol{\beta}) \times (\mathsf{s}^+ \to \boldsymbol{\beta})).\ ((\mathbf{pr}\mathbf{j}_2\ \mathbf{x})\ \mathbf{y})$$

$$\begin{aligned}
Atom^+[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} &= \{(\mathsf{e},\mathbf{m}) \mid \cdot \vdash \mathsf{e} : \mathsf{s} \wedge \boldsymbol{\Sigma}_{\mathbf{D}} \vdash \boldsymbol{\delta}(\mathsf{s}^+) \wedge \\
&\qquad \boldsymbol{\Sigma}_{\mathbf{D}}; \boldsymbol{\Sigma}_{\mathbf{G}} \vdash \mathbf{m} : \boldsymbol{\delta}(\mathsf{s}^+)\} \\
&\vdots \\
\mathcal{V}^+_\zeta[\![\mathsf{s}' \to \mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} &= \{(\lambda \mathsf{x} : \mathsf{s}'.\, \mathsf{e}, \lambda \mathbf{x}{:}\boldsymbol{\delta}((\mathsf{s}')^+).\mathbf{m}) \in Atom^+[\![\mathsf{s}' \to \mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} \mid \\
&\qquad \forall \mathsf{e}', \mathbf{m}'.(\mathsf{e}', \mathbf{m}') \in \mathcal{E}^+_\zeta[\![\mathsf{s}']\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} \implies \\
&\qquad (\mathsf{e}[\mathsf{e}'/\mathsf{x}], \mathbf{m}[\mathbf{m}'/\mathsf{x}]) \in \mathcal{E}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}\} \\
\mathcal{V}^+_\zeta[\![\mathsf{T}_\ell\ \mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} &= \{(\eta_\ell\ \mathsf{e}, \Lambda\boldsymbol{\beta}{::}*.\mathbf{m}) \in Atom^+[\![\mathsf{T}_\ell\ \mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} \mid \\
&\quad \text{let } \boldsymbol{\rho} = [\![\mathcal{L}^+_\ell]\!]^{\boldsymbol{\Sigma}}_\zeta \wedge \mathbf{x}_{\mathbf{p}} = \mathbf{pf}[\![\ell \preceq \mathsf{T}_\ell\ \mathsf{s}]\!] \text{ in} \\
&\quad \exists \mathbf{m}'.\boldsymbol{\Sigma}_{\mathbf{D}}; \boldsymbol{\Sigma}_{\mathbf{G}} \vdash \mathbf{m}' : \boldsymbol{\delta}(\mathsf{s}^+) \wedge \\
&\quad (\mathbf{m}[(\mathsf{T}_\ell\ \mathsf{s})^+/\boldsymbol{\beta}]\ \langle \mathbf{x}_{\mathbf{p}}, \eta_{\mathbf{k}}^{\ell,\mathsf{s}} \rangle, \eta_{\mathbf{k}}^{\ell,\mathsf{s}}\ \mathbf{m}') \in \boldsymbol{\mathcal{E}}[\![(\mathsf{T}_\ell\ \mathsf{s})^+]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\rho}} \wedge \\
&\quad (\mathsf{e}, \mathbf{m}') \in \mathcal{E}^+_\zeta[\![\mathsf{s}^+]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}\} \\
\mathcal{E}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} &= \{(\mathsf{e}, \mathbf{m}) \in Atom^+[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} \mid \exists \mathsf{v}, \mathbf{u}. \\
&\qquad \mathsf{e} \longmapsto^* \mathsf{v} \wedge \mathbf{m} \longmapsto^* \mathbf{u} \wedge (\mathsf{v}, \mathbf{u}) \in \mathcal{V}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}\} \\
\mathcal{G}^+_\zeta[\![\cdot]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} &= \{\,(\emptyset, \emptyset)\,\} \\
\mathcal{G}^+_\zeta[\![\Gamma, \mathsf{x} : \mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} &= \{(\gamma\,[\mathsf{x} \mapsto \mathsf{e}]\,, \boldsymbol{\gamma}\,[\mathbf{x} \mapsto \mathbf{m}]) \mid \\
&\qquad (\gamma, \boldsymbol{\gamma}) \in \mathcal{G}^+_\zeta[\![\Gamma]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}} \wedge (\mathsf{e}, \mathbf{m}) \in \mathcal{E}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}\}
\end{aligned}$$

$$\Gamma \mid \boldsymbol{\Sigma} \vdash \mathsf{e} \simeq \mathbf{m} : \mathsf{s} \mid \boldsymbol{\delta} \stackrel{\text{def}}{=} \Gamma \vdash \mathsf{e} : \mathsf{s} \wedge \boldsymbol{\Sigma}_{\mathbf{D}}; \boldsymbol{\Sigma}_{\mathbf{G}}, \hat{\Gamma^+} \vdash \mathbf{m} : \boldsymbol{\delta}(\mathsf{s}^+) \wedge \forall \zeta, (\gamma, \boldsymbol{\gamma}) \in \mathcal{G}^+_\zeta[\![\Gamma]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}.(\gamma(\mathsf{e}), \boldsymbol{\delta}(\boldsymbol{\gamma}(\mathbf{m}))) \in \mathcal{E}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}$$

$$\begin{aligned}
\Gamma \vdash \mathsf{e} \simeq \mathbf{m} : \mathsf{s} \stackrel{\text{def}}{=}\ & \Gamma \vdash \mathsf{e} : \mathsf{s} \wedge \mathcal{L}^+_\ell; \mathcal{L}^+_\sqsubseteq, \preceq^+, \Gamma^+ \vdash \mathbf{m} : \mathsf{s}^+ \wedge \\
& \text{let } \boldsymbol{\delta} = \{\boldsymbol{\alpha}_\ell \mapsto \hat{\boldsymbol{\alpha}}_\ell \mid \ell \in \mathcal{L}_\ell\} \cup \{\boldsymbol{\alpha}_\preceq \mapsto \hat{\boldsymbol{\alpha}}_\preceq\} \wedge \\
& \quad \boldsymbol{\gamma}_\sqsubseteq = [\![\mathcal{L}^+_\sqsubseteq]\!] \wedge \boldsymbol{\gamma}_\preceq = [\![\preceq^+]\!] \text{ in} \\
& \quad \Gamma \mid \boldsymbol{\Sigma} \vdash \mathsf{e} \simeq \boldsymbol{\delta}(\boldsymbol{\gamma}_\sqsubseteq(\boldsymbol{\gamma}_\preceq(\mathbf{m}))) : \mathsf{s} \mid \boldsymbol{\delta}
\end{aligned}$$

Figure 12. DCC to F_ω: Cross-Language Logical Relation

are related if, given related inputs, they produce related outputs. But when should two values be related at type $\mathsf{T}_\ell\ \mathsf{s}$? Intuitively, $\eta_\ell\ \mathsf{e}$ should be related to $\Lambda\boldsymbol{\beta}{::}*.\mathbf{m}$ if e is related to the protected contents of $\mathbf{m}$, which we will denote with $\mathbf{m}'$. We could extract $\mathbf{m}'$ if we could instantiate $\boldsymbol{\beta}$ with s^+ and apply the resulting term to a protection proof of type $(\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \mathsf{s}^+)$ and the identity continuation. However, a term of type $(\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \mathsf{s}^+)$ does not exist in general, so we cannot use the identity continuation.

Note that we can always construct a protection proof of type $(\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ (\mathsf{T}_\ell\ \mathsf{s})^+)$. If we instantiate $\boldsymbol{\beta}$ with $(\mathsf{T}_\ell\ \mathsf{s})^+$ and provide the proof and a suitable continuation, then $\mathbf{m}$ must eventually call that continuation on the protected term $\mathbf{m}'$. We use the continuation $\eta_{\mathbf{k}}^{\ell,\mathsf{s}}$ given at the top of Figure 12. This continuation corresponds to $\lambda \mathsf{x} : \mathsf{s}.\, \eta_\ell\ \mathsf{x}$ in DCC, and simply protects its argument at label ℓ. We consider $\eta_\ell\ \mathsf{e}$ related to $\Lambda\boldsymbol{\beta}{::}*.\mathbf{m}$ when there exists an $\mathbf{m}'$ such that $\mathbf{m}[(\mathsf{T}_\ell\ \mathsf{s})^+/\boldsymbol{\beta}]\ \eta_{\mathbf{k}}^{\ell,\mathsf{s}}$ is equivalent to $\eta_{\mathbf{k}}^{\ell,\mathsf{s}}\ \mathbf{m}'$ (in the target language!), and e is related to $\mathbf{m}'$. This technique is reminiscent of the cross-language relation for CPS given by Chlipala [8].

To require equivalence of the two F_ω terms above, we use the F_ω logical relation $\boldsymbol{\mathcal{E}}$, which is indexed by $\boldsymbol{\rho}$. In Figure 12, we use the $\boldsymbol{\rho}$ defined for our protection ADT in §5.3. To generate this $\boldsymbol{\rho}$ we require an observer, therefore the value and term relations must be indexed by an observer. However, we quantify over all possible observers when we define the relation $\simeq$ for open terms.

Two terms e and $\mathbf{m}$ are related in $\mathcal{E}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}$ if they evaluate to values that are related in $\mathcal{V}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}$. Note that any target term in $\mathcal{E}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}$ must reduce to a value, since the type s^+ cannot be an abstract type $\boldsymbol{\alpha}$, whereas in $\boldsymbol{\mathcal{E}}[\![\boldsymbol{\alpha}]\!]^{\mathbf{D};\mathbf{G}}_{\boldsymbol{\rho}}$ terms may not reduce to a value. In the top-level relation $\Gamma \vdash \mathsf{e} \simeq \mathbf{m} : \mathsf{s}$, as in §5.3, we use $[\![\mathcal{L}^+_\sqsubseteq]\!]$ and $[\![\preceq^+]\!]$ from Figure 10 to implement $\mathcal{L}^+_\sqsubseteq$ and $\preceq^+$. We pick $\boldsymbol{\delta}$ using the same types from $[\![\mathcal{L}^+_\ell]\!]^{\boldsymbol{\Sigma}}_\zeta$. Unlike in the F_ω logical relation, we do not provide a relational interpretation for these types because terms of type $\hat{\boldsymbol{\alpha}}_\ell$ or $(\hat{\boldsymbol{\alpha}}_\preceq\ \hat{\boldsymbol{\alpha}}_\ell\ \mathbf{t})$ are never related by this logical relation. Such terms can only appear in larger terms of translation type.

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 $\mathbf{D}, \boldsymbol{\alpha} :: * \vdash \boldsymbol{\rho}_\mathbf{i}(\mathbf{t}_1) :: *$, $\mathbf{D} \vdash \boldsymbol{\rho}_\mathbf{1}(\mathbf{t}_g) :: *$, $\mathbf{D} \vdash \boldsymbol{\rho}_\mathbf{2}(\mathbf{t}_f) :: *$,
$\mathbf{D}; \mathbf{G} \vdash \mathbf{m} : \boldsymbol{\rho}_\mathbf{i}(\forall \boldsymbol{\alpha}{::}*.\ (\mathbf{t}_1 \times (\mathbf{t}_2 \to \boldsymbol{\alpha})) \to \boldsymbol{\alpha})$,
$\mathbf{D}; \mathbf{G} \vdash \mathbf{m}_f : \mathbf{t}_g \to \mathbf{t}_f$, $\mathbf{D}; \mathbf{G} \vdash \mathbf{m}_g : \mathbf{t}_2 \to \mathbf{t}_g$,
$(\mathbf{m}_a, \mathbf{m}'_a) \in \boldsymbol{\mathcal{E}}[\![\mathbf{t}_1]\!]^{\mathbf{D};\mathbf{G}}_{\boldsymbol{\rho}[\boldsymbol{\alpha} \mapsto (\mathbf{t}_g, \mathbf{t}_f, \mathbf{R})]}$
then $(\mathbf{m}_f\ (\mathbf{m}\ [\mathbf{t}_g]\ \langle \mathbf{m}_a, \mathbf{m}_g \rangle), \mathbf{m}\ [\mathbf{t}_f]\ \langle \mathbf{m}'_a, \mathbf{m}_f \circ \mathbf{m}_g \rangle) \in \boldsymbol{\mathcal{E}}[\![\mathbf{t}_f]\!]^{\mathbf{D};\mathbf{G}}_{\boldsymbol{\rho}}$

Lemma 6.2 below tells us that if a source term e_1 is related to $\mathbf{m}_1$, and $\mathbf{m}_1$ is related to $\mathbf{m}_2$ in the F_ω relation, e_1 is also related to $\mathbf{m}_2$ in the cross-language relation. This allows us to use reasoning in the target logical relation, *e.g.*, to use the parametricity condition, to reason about the cross-language relation.

Lemma 6.2 (Cross-language relation respects F_ω relation)
Let $\boldsymbol{\rho} = [\![\mathcal{L}^+_\ell]\!]^{\boldsymbol{\Sigma}}_\zeta$. *If* $(\mathsf{v}, \mathbf{u}) \in \mathcal{V}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}$ *and* $(\mathbf{u}_1, \mathbf{u}_2) \in \boldsymbol{\mathcal{E}}[\![\mathsf{s}^+]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\rho}}$ *then* $(\mathsf{v}, \mathbf{u}_2) \in \mathcal{V}^+_\zeta[\![\mathsf{s}]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}$

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 $\mathbf{m}$ runs to the same boolean value as e—follows from this theorem. The proof of Theorem 6.3 is by induction on $\Gamma \vdash \mathsf{e} : \mathsf{s} \rightsquigarrow \mathbf{m}$. The case for bind is the most interesting. We sketch this case by noting a series of equivalences. We decompose the translation of the bind continuation ($\lambda \mathbf{x}{:}\mathsf{s}_1^{\hat{+}}.\mathbf{m}_2$ in the proof below) into the $\eta_{\mathbf{k}}^{\ell,\mathsf{s}}$ continuation and the continuation $\mathbf{m}_f$ defined below. Then we use the parametricity condition and the definition of $\mathcal{V}^+_\zeta[\![\mathsf{T}_\ell\ \mathsf{s}_1]\!]^{\boldsymbol{\Sigma}}_{\boldsymbol{\delta}}$ to find the protected target value $\mathbf{m}'$ inside $\mathbf{m}_1$.

Theorem 6.3 (Translation preserves semantics)
If $\Gamma \vdash \mathsf{e} : \mathsf{s}$ *and* $\Gamma \vdash \mathsf{e} : \mathsf{s} \rightsquigarrow \mathbf{m}$ *then* $\Gamma \vdash \mathsf{e} \simeq \mathbf{m} : \mathsf{s}$

Proof Sketch:
Show $\mathsf{bind}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 : \mathsf{s}_2$ is related to
$\mathbf{m}_1\ [\mathsf{s}_2^+]\ \langle \mathbf{pf}[\![\ell \preceq \mathsf{s}_2]\!], \lambda \mathbf{x}{:}\mathsf{s}_1^+.\mathbf{m}_2 \rangle$
Let $\mathbf{x}_\mathbf{p} = \mathbf{pf}[\![\ell \preceq \mathsf{s}_2]\!]$
$\mathbf{m}_f = \lambda \mathbf{y}{:}(\mathsf{T}_\ell\ \mathsf{s}_1)^{\hat{+}}.\mathbf{y}\ [\mathsf{s}_2^{\hat{+}}]\ \langle \mathbf{x}_\mathbf{p}, \lambda \mathbf{x}{:}\mathsf{s}_1^{\hat{+}}.\mathbf{m}_2 \rangle : (\mathsf{T}_\ell\ \mathsf{s}_1)^{\hat{+}} \to \mathsf{s}_2^{\hat{+}}$.
Suppose $\mathsf{e}_1 \longmapsto^* \eta_\ell\ \mathsf{e}'_1$.
Suffices to show:

$\mathsf{e}_2[\mathsf{e}'_1/\mathsf{x}] \approx \mathbf{m}_1\ [\mathsf{s}_2^+]\ \langle \mathbf{pf}[\![\ell \preceq \mathsf{s}_2]\!], \lambda \mathbf{x}{:}\mathsf{s}_1^+.\mathbf{m}_2 \rangle$	*by evaluation*
$\mathsf{e}_2[\mathsf{e}'_1/\mathsf{x}] \approx \mathbf{m}_1\ [\mathsf{s}_2^+]\ \langle \mathbf{pf}[\![\ell \preceq \mathsf{s}_2]\!], \mathbf{m}_f \circ \eta_{\mathbf{k}}^{\ell,\mathsf{s}} \rangle$	*by Lemma 6.2*
$\mathsf{e}_2[\mathsf{e}'_1/\mathsf{x}] \approx \mathbf{m}_f\ (\mathbf{m}_1\ [\mathsf{s}_2^+]\ \langle \mathbf{pf}[\![\ell \preceq \mathsf{T}_\ell\ \mathsf{s}_1]\!], \eta_{\mathbf{k}}^{\ell,\mathsf{s}} \rangle)$	*by Lemma 6.1*
$\mathsf{e}_2[\mathsf{e}'_1/\mathsf{x}] \approx \mathbf{m}_f\ (\eta_{\mathbf{k}}^{\ell,\mathsf{s}}\ \mathbf{m}')$	*by IH on* e_1 *and* $\mathbf{m}_1$
$\mathsf{e}_2[\mathsf{e}'_1/\mathsf{x}] \approx \mathbf{m}_2[\mathbf{m}'/\mathbf{x}]$	*by evaluation*

Follows by IH on e_2 and $\mathbf{m}_2$

7. Translation Preserves Noninterference

In this section, we present our central result, that our translation from DCC to F_ω preserves noninterference. The proof requires a back-translation. Our back-translation is inspired by Ahmed and Blume'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 $\mathsf{s}^{\hat{+}}$ to denote s^+ with all instances of $\boldsymbol{\alpha}_\ell$ and $\boldsymbol{\alpha}_\preceq$ replaced with $\hat{\boldsymbol{\alpha}}_\ell$ and $\hat{\boldsymbol{\alpha}}_\preceq$, respectively. We analogously put the hat symbol $\hat{}$ on other notation that we have defined already, such as Γ^+ to $\hat{\Gamma^+}$, to indicate substitution of type and term variables in $\mathcal{L}^+_\ell; \mathcal{L}^+_\sqsubseteq, \preceq^+$ with those in $\mathbf{D}_\ell; \mathbf{G}_\ell, \mathbf{G}_\preceq$.

$\boxed{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}}$ where $\Sigma_\mathbf{D}; \Sigma_\mathbf{G}, \mathbf{G_k}, \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$ and $\Gamma \vdash \mathsf{e} : \mathsf{s}$

$$\frac{}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \langle\rangle : \mathbf{1} \uparrow \langle\rangle} \qquad \frac{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}_i^{\hat{+}} \uparrow \mathsf{e}}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{inj_i}\ \mathbf{m} : \mathsf{s}_1^{\hat{+}} + \mathsf{s}_2^{\hat{+}} \uparrow \mathsf{inj}_i\ \mathsf{e}} \qquad \frac{(\mathbf{x} : \mathsf{s}^{\hat{+}}) \in \Gamma^{\hat{+}}}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{x} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{x}} \qquad \frac{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}}, \mathbf{x} : \mathsf{s}^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}_2^{\hat{+}} \uparrow \mathsf{e}}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \lambda \mathbf{x}{:}\mathsf{s}_1^{\hat{+}}.\mathbf{m} : \mathsf{s}_1^{\hat{+}} \to \mathsf{s}_2^{\hat{+}} \uparrow \lambda \mathsf{x}{:}\mathsf{s}_1.\,\mathsf{e}}$$

$$\frac{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m_1} : \mathsf{s}_1^{\hat{+}} \uparrow \mathsf{e}_1 \qquad \Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m_2} : \mathsf{s}_2^{\hat{+}} \uparrow \mathsf{e}_2}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \langle \mathbf{m_1}, \mathbf{m_2} \rangle : \mathsf{s}_1^{\hat{+}} \times \mathsf{s}_2^{\hat{+}} \uparrow \langle \mathsf{e}_1, \mathsf{e}_2 \rangle} \qquad \text{FD-}\kappa\ \frac{\mathbf{k} : (\mathsf{s}^{\hat{+}} \to (\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}) \in \mathbf{G_k}}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{k} : \mathsf{s}^{\hat{+}} \to \mathsf{T}_\ell\ \mathsf{s}^{\hat{+}} \uparrow \lambda \mathsf{x}{:}\mathsf{s}.\,\eta_\ell\,\mathsf{x}}$$

$$\text{FD-}\eta\ \frac{\Sigma; \mathbf{G_k}, \mathbf{k} : (\mathsf{s}^{\hat{+}} \to (\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}); \Gamma^{\hat{+}} \vdash \mathbf{m}[(\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}/\beta]\ \langle \hat{\mathbf{pf}}[\![\ell \preceq \mathsf{T}_\ell\ \mathsf{s}]\!], \mathbf{k} \rangle : \mathsf{T}_\ell\ \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \Lambda\beta{::}*.\mathbf{m} : \mathsf{T}_\ell\ \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}}$$

$$\text{FD-BIND}\ \frac{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m_p} : (\alpha_{\preceq}\ \alpha_\ell\ \mathsf{s}^{\hat{+}}) \times (\mathsf{s}'^{\hat{+}} \to \mathsf{s}^{\hat{+}}) \qquad \Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m} : (\mathsf{T}_\ell\ \mathsf{s}')^{\hat{+}} \uparrow \mathsf{e} \qquad \Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{prj}_2\ \mathbf{m_p} : \mathsf{s}'^{\hat{+}} \to \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}'}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m}\ [\mathsf{s}^{\hat{+}}]\ \mathbf{m_p} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{bind}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}'\ \mathsf{x}}$$

$$\text{FD-SUBTERM}\ \frac{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{F} : \mathsf{s}_1^{\hat{+}} \Rightarrow \mathsf{s}_2^{\hat{+}} \uparrow \mathsf{F}' \qquad \Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}_1^{\hat{+}} \uparrow \mathsf{e}}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{F}[\mathbf{m}] : \mathsf{s}_2^{\hat{+}} \uparrow \mathsf{F}'[\mathsf{e}]} \qquad \text{FD-HOLE}^{\not\to}\ \frac{\mathbf{E}^{\not\to}[\mathbf{u}] \longmapsto \mathbf{m_1} \qquad \Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m_1} : \mathsf{s}_2^{\hat{+}} \uparrow \mathsf{e}}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{E}^{\not\to}[\mathbf{u}] : \mathsf{s}_2^{\hat{+}} \uparrow \mathsf{e}}$$

$$\text{FD-HOLE}^{+}\ \frac{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}_1^{\hat{+}} + \mathsf{s}_2^{\hat{+}} \qquad \Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{case}\ \mathbf{m}\ \mathbf{of}\ \mathbf{inj_1}\ \mathbf{x_1}.\,\mathbf{m_1} \parallel \mathbf{inj_2}\ \mathbf{x_2}.\,\mathbf{m_2} : \mathbf{t}\ \text{where}\ \nexists \mathsf{s}'.\mathbf{t} = \mathsf{s}'^{\hat{+}} \quad \Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{case}\ \mathbf{m}\ \mathbf{of}\ \mathbf{inj_1}\ \mathbf{y_1}.\,\mathbf{E}^{\not\to}[\mathbf{m_1}[\mathbf{y_1}/\mathbf{x_1}]] \parallel \mathbf{inj_2}\ \mathbf{y_2}.\,\mathbf{E}^{\not\to}[\mathbf{m_2}[\mathbf{y_2}/\mathbf{x_2}]] : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e} \qquad (\mathit{fresh}\ \mathbf{y_1}, \mathbf{y_2})}{\Sigma; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{E}^{\not\to}[\mathbf{case}\ \mathbf{m}\ \mathbf{of}\ \mathbf{inj_1}\ \mathbf{x_1}.\,\mathbf{m_1} \parallel \mathbf{inj_2}\ \mathbf{x_2}.\,\mathbf{m_2}] : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}}$$

Figure 13. F_ω to DCC: Term Back-Translation

7.1 Back-Translation

Recall from §3 that proving equivalence preservation for functions requires a back-translation. Specifically, proving that $(\mathsf{f}_1, \mathsf{f}_2) \in \mathcal{V}[\![\mathsf{s}' \to \mathsf{s}]\!]_\zeta$ implies $(\mathbf{f_1}, \mathbf{f_2}) \in \boldsymbol{\mathcal{V}}[\![\mathsf{s}'^{+} \to \mathsf{s}^{+}]\!]^{\Sigma}_{\rho}$ (where $\rho = [\![\mathcal{L}^{+}_{\ell}]\!]^{\Sigma}_{\zeta}$), requires that given $(\mathbf{m_1}, \mathbf{m_2}) \in \boldsymbol{\mathcal{E}}[\![\mathsf{s}'^{+}]\!]^{\Sigma}_{\rho}$ we can produce $(\mathsf{e}_1, \mathsf{e}_2) \in \mathcal{E}[\![\mathsf{s}']\!]_\zeta$. Note that $\mathbf{m_1}$ and $\mathbf{m_2}$ have the translation type $\mathsf{s}'^{\hat{+}}$ under Σ. So, at the "top-level" we need to back-translate terms $\mathbf{m}$ where $\Sigma \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$. However, consider the function $\lambda \mathbf{x}{:}\mathsf{s}'^{\hat{+}}.\mathbf{m} : \mathsf{s}'^{\hat{+}} \to \mathsf{s}^{\hat{+}}$. The obvious way to back-translate this is to back-translate the term $\Sigma, \mathbf{x} : \mathsf{s}^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$. Therefore, we will at least need to be able to back-translate terms $\mathbf{m}$ such that $\Sigma_\mathbf{D}; \Sigma_\mathbf{G}, \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$, where $\Gamma^{\hat{+}}$ is restricted to contain only variables of translation type. To do so, we will set up a back-translation judgment of the form $\Sigma; \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}$ which says that the target term $\mathbf{m}$, which has type $\mathsf{s}^{\hat{+}}$ under the open ADT Σ and the term context $\Gamma^{\hat{+}}$, back-translates to e of type s under context Γ. In fact, our back-translation, defined in Figure 13, will require an additional environment $\mathbf{G_k}$, but the reader should ignore that for now. We will introduce this extra environment as we explain Figure 13. First, we discuss the rules for back-translating values and then the more complex rules for back-translating expressions.

Back-translating values The rules for back-translating values are in the top half of Figure 13. We back-translate $\langle\rangle$ to $\langle\rangle$. We back-translate a pair $\langle \mathbf{m_1}, \mathbf{m_2} \rangle : \mathsf{s}_1^{\hat{+}} \times \mathsf{s}_2^{\hat{+}}$ to $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ if $\mathbf{m_1} : \mathsf{s}_1^{\hat{+}}$ and $\mathbf{m_2} : \mathsf{s}_2^{\hat{+}}$ back-translate to e_1 and e_2, respectively. We back-translate sums and functions of translation type similarly by structural recursion since their subterms must be of translation type.

For type abstractions, we only need to be able to back-translate terms $\Lambda\beta{::}\kappa.\mathbf{m}$ of translation type, *i.e.*, of type $(\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}$, which is defined by rule FD-η. Note that $\Lambda\beta{::}\kappa.\mathbf{m}$ can only be of translation type when $\kappa = *$, so we only need to back-translate $\Lambda\beta{::}*.\mathbf{m}$. As discussed in §6, the term $\Lambda\beta{::}*.\mathbf{m}$ has some protected contents $\mathbf{m}'$. Roughly speaking, if we could back-translate $\mathbf{m}' : \mathsf{s}^{\hat{+}}$ to e', then $\Lambda\beta{::}*.\mathbf{m} : (\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}$ should back-translate to $\eta_\ell\,\mathsf{e}'$. To extract $\mathbf{m}'$, we need to provide a protection proof for $(\hat{\alpha}_{\preceq}\ \hat{\alpha}_\ell\ \mathsf{s}^{\hat{+}})$ which does not exist in general. Previously, we noted that we can always construct a protection proof for $(\hat{\alpha}_{\preceq}\ \hat{\alpha}_\ell\ (\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}})$ and used the continuation $\eta^{\ell,\mathsf{s}}_{\mathbf{k}}$. However, here if we use the continuation $\eta^{\ell,\mathsf{s}}_{\mathbf{k}}$ then the back-translation would not be well-founded. Instead, we introduce a continuation variable $\mathbf{k} : \mathsf{s}^{\hat{+}} \to (\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}$ and keep it in a separate environment $\mathbf{G_k}$. Then we back-translate $\Lambda\beta{::}*.\mathbf{m}$ to e by back-translating $\mathbf{m_k} = \mathbf{m}[\mathsf{T}_\ell\ \mathsf{s}^{\hat{+}}/\beta]\ \langle \hat{\mathbf{pf}}[\![\ell \preceq \mathsf{T}_\ell\ \mathsf{s}]\!], \mathbf{k} \rangle$ to e. Note that parametricity guarantees that the term $\mathbf{m_k}$ will return the result of the continuation $\mathbf{k}$ applied to the protected contents $\mathbf{m}'$. Thus, the back-translation of $\mathbf{m_k}$ will eventually back-translate $\mathbf{k}\,\mathbf{m}'$ to produce its result. We back-translate $\mathbf{k}$ via FD-κ to $\lambda \mathsf{x} : \mathsf{s}.\,\eta_\ell\,\mathsf{x}$ and back-translate $\mathbf{m}'$ to e'. Hence, we back-translate $\mathbf{k}\ \mathbf{m}'$ to $(\lambda \mathsf{x} : \mathsf{s}.\,\eta_\ell\,\mathsf{x})\ \mathsf{e}'$ which is beta-equivalent to $\eta_\ell\,\mathsf{e}'$.

Back-translating expressions When back-translating a term whose subterms are all of translation type, we proceed by structural recursion for most forms. When a term has subterms of non-translation type, we use partial evaluation to eliminate those subterms. Our partial evaluation is more involved than Ahmed and Blume's because our target language is not restricted to CPS form (*i.e.*, not all subterms of an expression are values). With a CPS restriction, every elimination form in evaluation position is a redex, but without this restriction one needs to look arbitrarily deep to find a redex.

The back-translation of bind expressions depends on the invariants imposed by the protection types, expressed in the FD-BIND rule. In the target language, bind appears as an expression like $\mathbf{m}'\ [\mathsf{s}^{\hat{+}}]\ \mathbf{m_p}$. That is, an expression of type $(\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}$ applied to a type and a protected continuation.

To illustrate back-translation of other expressions, let us consider how to back-translate the term $\mathbf{prj}_1\ \mathbf{m} : \mathsf{s}_1^{\hat{+}}$. There are two cases.

1. Subterms are of translation type In the simplest case, the subterm $\mathbf{m}$ is of translation type $\mathsf{s}_1^{\hat{+}} \times \mathsf{s}_2^{\hat{+}}$. We can back-translate $\mathbf{m}$ to e and $\mathbf{prj}_1\ \mathbf{m}$ to $\mathsf{prj}_1\ \mathsf{e}$.

The same idea applies to all elimination forms of translation type, such as application and case, and is captured by the FD-SUBTERM rule. To abstract this reasoning, we introduce a restricted evaluation context $\mathbf{F}$—the grammar appears in Figure 14. This context is restricted to be one level deep. Any target elimination form can be written as $\mathbf{F}[\mathbf{m}]$. If both the type of the hole and the result are translation types, then we can simply back-translate $\mathbf{F}$ to F and $\mathbf{m}$ to e, and produce $\mathsf{F}[\mathsf{e}]$. We omit the definition of context typing, as it is standard. We write $\mathbf{F} : \mathbf{t}_1 \Rightarrow \mathbf{t}_2$ to mean the hole of $\mathbf{F}$ has type $\mathbf{t}_1$ while the result has type $\mathbf{t}_2$, under type and term environments that are obvious from context. We also omit the back-translation of $\mathbf{F}$. Each

F_ω *ctxt* $\mathbf{F} ::= \mathbf{prj_i}\ [\cdot]_\mathbf{T} \mid [\cdot]_\mathbf{T}\ \mathbf{m} \mid [\cdot]_\mathbf{T}\ [\mathbf{t}] \mid \mathbf{case}\ [\cdot]_\mathbf{T}\ \mathbf{of\ inj_1\ y.\ m_1} \parallel \mathbf{inj_2\ y.\ m_2}$

DCC ctxt $\mathsf{F} ::= \mathsf{prj_i}\ [\cdot]_\mathsf{S} \mid [\cdot]_\mathsf{S}\ \mathsf{e} \mid \mathsf{bind\ x} = [\cdot]_\mathsf{S}\ \mathsf{in\ e} \mid \mathsf{case}\ [\cdot]_\mathsf{S}\ \mathsf{of\ inj_1\ y.\ e_1} \parallel \mathsf{inj_2\ y.\ e_2}$

$\boxed{\mathbf{E}^{\not=} = \mathbf{F}_0[\mathbf{F}_1[...\mathbf{F}_n]]}$ where

$$\mathbf{\Sigma_D}; \mathbf{\Sigma_G}, \mathbf{G_k}, \Gamma^{\hat{+}} \vdash \mathbf{F}_0 : \mathbf{t}_1 \Rightarrow \mathsf{s}^{\hat{+}} \qquad \forall i \in [1,n+1].\ \not\exists \mathsf{s_i}.\mathbf{t}_i = \mathsf{s}_i^{\hat{+}}$$

$$\forall i \in [1,n].\mathbf{\Sigma_D}; \mathbf{\Sigma_G}, \mathbf{G_k}, \Gamma^{\hat{+}} \vdash \mathbf{F}_i : \mathbf{t}_{i+1} \Rightarrow \mathbf{t}_i$$

Figure 14. F_ω to DCC: Back-Translation Contexts

$\mathbf{F}$ is back-translated by back-translating each of its subterms—which a simple case analysis shows must all be of translation type—and back-translating $[\cdot]_\mathbf{T}$ to $[\cdot]_\mathsf{S}$.

2. Subterms are not of translation type Next, consider the scenario where the subterm $\mathbf{m}$ is of non-translation type $\mathsf{s}_1^{\hat{+}} \times \mathbf{t}_2$. Let us consider the structure of $\mathbf{m}$. If $\mathbf{m}$ is a value $\mathbf{u}$ that is not a variable, then $\mathbf{u} = \langle \mathbf{m_1}, \mathbf{m_2} \rangle$. Hence, we can reduce $\mathbf{prj}_1\ \langle \mathbf{m_1}, \mathbf{m_2} \rangle \longmapsto \mathbf{m_1}$ and then back-translate $\mathbf{m_1} : \mathsf{s}_1^{\hat{+}}$. We will come back to the possibility of $\mathbf{u}$ being a variable shortly.

But what if $\mathbf{m}$ is not a value? Intuitively, there must be *some* redex $\mathbf{F}[\mathbf{u}]$ in $\mathbf{m}$. 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 $\mathbf{m}$ is the redex $\mathbf{F}[\mathbf{u}]$, then we can reduce $\mathbf{prj}_1\ \mathbf{F}[\mathbf{u}] \longmapsto \mathbf{prj}_1\ \mathbf{m'}$ and continue back-translating $\mathbf{prj}_1\ \mathbf{m'} : \mathsf{s}_1^{\hat{+}}$. 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 $\mathbf{F}_0[\mathbf{F}_1[\mathbf{F}_2[\ldots \mathbf{F}_n[\mathbf{u}]]]]$ (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 $\mathbf{E}^{\not=}[\mathbf{u}] = \mathbf{F}_0[\mathbf{F}_1[\mathbf{F}_2[\ldots \mathbf{F}_n[\mathbf{u}]]]]$, where the result of $\mathbf{F}_0$ is a translation type and the hole and result types of all other $\mathbf{F}_i$ are non-translation types. We refer to the redex $\mathbf{F}_n[\mathbf{u}]$ as a non-translation redex. We use $\mathbf{E}^{\not=}$ to denote such a sequence of $\mathbf{F}_i$ contexts, formally defined in Figure 14. In this case, the rule FD-HOLE applies. We perform one step of evaluation, eliminating the redex $\mathbf{F}_n[\mathbf{u}]$, then continue back-translating the resulting term. We might worry that here $\mathbf{u}$ can be a variable, and thus $\mathbf{E}^{\not=}[\mathbf{u}]$ is stuck. However, recall by assumption that the hole of $\mathbf{F}_n$ must be of non-translation type. Let us consider which variables can appear in the hole of $\mathbf{F}_n$.

During back-translation, only certain variables are free. Free variables can be either the coercion functions $\mathbf{c}_{\ell'\ell}$, the proof constructors $\hat{\mathbf{p}}_\mathbf{t}$, the variables $\mathbf{x} : \mathsf{s}^{\hat{+}}$ from functions of translation type, or the variables $\mathbf{k} : \mathsf{s}^{\hat{+}} \to (\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}}$ introduced by the back-translation. Clearly a variable of translation type cannot appear in the hole, so $\mathbf{u}$ cannot be one of the variables from the final two cases.

Suppose $\mathbf{u}$ is a coercion function $\mathbf{c}_{\ell'\ell} : \hat{\boldsymbol{\alpha}}_{\ell'} \to \hat{\boldsymbol{\alpha}}_\ell$. Then $\mathbf{F}_n[\mathbf{u}]$ has type $\hat{\boldsymbol{\alpha}}_\ell$. However, a term of type $\hat{\boldsymbol{\alpha}}_\ell$ cannot appear in any evaluation context. This follows by considering the type of each evaluation context in F_ω. Since there must be at least an outer $\mathbf{F}_0$ whose result is of translation type, $\mathbf{u}$ cannot be a coercion function.

Finally, suppose $\mathbf{u}$ is a proof constructor, for instance, $\hat{\mathbf{p}}_\mathbf{1}$. Then $\mathbf{F}_n[\mathbf{u}] = \hat{\mathbf{p}}_\mathbf{1}\ [\mathbf{t}]$ has type $(\hat{\boldsymbol{\alpha}}_\preceq\ \hat{\boldsymbol{\alpha}}_\ell\ \mathbf{1})$. Again, terms of this type cannot appear in any evaluation context due to the types of evaluation context in F_ω. Similar reasoning applies to all proof constructors since after wrapping them in some number of $\mathbf{F}$ contexts the final result will be a term of protection type which cannot appear in any evaluation context. So $\mathbf{u}$ cannot be a proof constructor. Hence, we conclude that it is impossible for a value $\mathbf{u}$ in $\mathbf{E}^{\not=}[\mathbf{u}]$ to be a variable.

2b. There does not exist a non-translation redex Note that the FD-HOLE$^{\not=}$ rule requires that the hole of each $\mathbf{F}_i$ 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 $\mathbf{E}^{\not=}[\mathbf{F}_i[\mathbf{m}]] = \mathbf{F}_0[\mathbf{F}_1[\mathbf{F}_2[\ldots \mathbf{F}_i[\mathbf{m}]]]]$, where $\mathbf{m}$ is the first term (going outside in) that has translation type, but the result of $\mathbf{F}_i$ is of non-translation type. In this case $\mathbf{E}^{\not=}[\mathbf{F}_i]$ is not a valid $\mathbf{E}_\mathbf{1}^{\not=}$ and there is no non-translation redex. This could happen, for instance, if $\mathbf{m}$ is a variable of translation type, which we can only rule out when $\mathbf{E}^{\not=}[\mathbf{F}_i]$ *is* a valid $\mathbf{E}_\mathbf{1}^{\not=}$. Let us analyze what $\mathbf{F}_i$ could be.

If $\mathbf{F}_i$ is $\mathbf{prj_i}\ [\cdot]_\mathbf{T}$, then $\mathbf{m}$ must have type $\mathsf{s}^{\hat{+}} \times \mathsf{s'}^{\hat{+}}$. But then the result of $\mathbf{F}_i$ is a translation type, so $\mathbf{F}_i$ cannot be a projection.

If $\mathbf{F}_i$ is $[\cdot]_\mathbf{T}\ \mathbf{m'}$, then $\mathbf{m'}$ must have type $\mathsf{s}^{\hat{+}} \to \mathsf{s'}^{\hat{+}}$. But then the result of $\mathbf{F}_i$ is a translation type, so $\mathbf{F}_i$ cannot be an application.

If $\mathbf{F}_i$ is $[\cdot]_\mathbf{T}\ [\mathbf{t}]$, then $\mathbf{m}$ must have type $(\mathsf{T}_\ell\ \mathsf{s'})^{\hat{+}}$. But then $\mathbf{F}_i[\mathbf{m}]$ must be $\mathbf{m'}\ [\mathbf{t}]$, and this must appear in at least one higher context $\mathbf{m'}\ [\mathbf{t}]\ \mathbf{m_p}$. But $\mathbf{m_p} : (\boldsymbol{\alpha}_\preceq\ \boldsymbol{\alpha}_\ell\ \mathsf{s}^{\hat{+}}) \times \mathsf{s'}^{\hat{+}} \to \mathsf{s}^{\hat{+}}$—since we can only construct protection proofs for translation types—so $\mathbf{t} = \mathsf{s}^{\hat{+}}$. Therefore, $\mathbf{m'}\ [\mathbf{t}]\ \mathbf{m_p}$ is of translation type. Recall that we assumed that $\mathbf{m}$ is the first term of translation type, so $\mathbf{F}_i$ cannot be a type instantiation.

Finally, if $\mathbf{F}_i$ is $\mathbf{case}\ [\cdot]_\mathbf{T}\ \mathbf{of\ inj_1\ x_1.\ m_1} \parallel \mathbf{inj_2\ x_2.\ m_2}$, then $\mathbf{m}$ must have type $\mathsf{s}^{\hat{+}} + \mathsf{s'}^{\hat{+}}$, yet the result of $\mathbf{F}_i$ *can* be of non-translation type! That is, we are trying to back-translate the following expression:

$$\cdots \vdash \mathbf{E}^{\not=}[\mathbf{case\ m\ of\ inj_1\ x_1.\ m_1} \parallel \mathbf{inj_2\ x_2.\ m_2}] : \mathsf{s}_1^{\hat{+}}$$

We could back-translate $\mathbf{E}^{\not=}[\mathbf{case\ m\ of\ inj_1\ x_1.\ m_1} \parallel \mathbf{inj_2\ x_2.\ m_2}]$ if we could rewrite this expression into one in which all the subterms are of translation type. Recall that we know the result of $\mathbf{E}^{\not=}$ must result in a translation type, and by assumption $\mathbf{m}$ is of translation type. Therefore, we *can* rewrite the term! Specifically, we rewrite the term into:

$$\cdots \vdash \mathbf{case\ m\ of\ inj_1\ x_1.\ E}^{\not=}[\mathbf{m_1}] \parallel \mathbf{inj_2\ x_2.\ E}^{\not=}[\mathbf{m_2}] : \mathsf{s}_1^{\hat{+}}$$

This scenario is captured by the rule FD-HOLE$^+$. 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_ω term under $\mathbf{\Sigma}$ 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 $\mathbf{m}$ to belong to our logical relation if there exists a term e such that $\mathbf{m}$ back-translates to e. Back-translation performs partial evaluation, but the "normalization" done during back-translation differs from evaluation using the F_ω dynamic semantics, for instance, because we perform reduction under λ.

We briefly present the logical relation at a high-level. Full definitions and proofs are available in our online technical appendix [7].

$$
\begin{array}{ll}
Atom^{\hat{\uparrow}}[\mathbf{t}]^{\Sigma}_{\delta} & = \{\,((\mathbf{G_k};\Gamma^{\hat{+}}),\mathbf{m}) \mid \Sigma_{\mathbf{D}};\Sigma_{\mathbf{G}},\mathbf{G_k},\Gamma^{\hat{+}} \vdash \mathbf{m} : \delta(\mathbf{t})\,\} \\
Atom^{\hat{\uparrow}\mathrm{ctx}}\left[\mathbf{t},\mathbf{s}^{\hat{+}}\right]^{\Sigma}_{\delta} & = \{\,((\mathbf{G_k};\Gamma^{\hat{+}}),\mathbf{E}^{\not\lambda}) \mid \Sigma_{\mathbf{D}};\Sigma_{\mathbf{G}},\mathbf{G_k},\Gamma^{\hat{+}} \vdash \mathbf{E}^{\not\lambda} : \delta(\mathbf{t}) \Rightarrow \mathbf{s}^{\hat{+}}\,\} \\
\mathcal{O}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta} & = \{(\mathbf{W},\mathbf{m}) \mid (\mathbf{W},\mathbf{m}) \in Atom^{\hat{\uparrow}}\left[\mathbf{s}^{\hat{+}}\right]^{\Sigma}_{\delta} \wedge \exists \mathbf{e}.\Sigma;\mathbf{W_k};\mathbf{W}_\Gamma \vdash \mathbf{m} : \mathbf{s}^{\hat{+}} \uparrow \mathbf{e}\} \\
Wf^{\hat{\uparrow}\mathrm{ctx}\ \Sigma}[\mathbf{t},\mathbf{s}^{\hat{+}}] & = \{(\mathbf{W},\mathbf{E}^{\not\lambda}) \in Atom^{\hat{\uparrow}\mathrm{ctx}}\left[\mathbf{t},\mathbf{s}^{\hat{+}}\right]^{\Sigma}_{\emptyset} \mid \forall \mathbf{W}',\mathbf{u}.(\mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}',\mathbf{u}) \in Atom^{\hat{\uparrow}\mathrm{val}}[\mathbf{t}]^{\Sigma}_{\emptyset}) \implies (\mathbf{W}',\mathbf{E}^{\not\lambda}[\mathbf{u}]) \in \mathcal{O}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\emptyset}\} \\
Rel^{\hat{\uparrow}\ \Sigma}_{*} & = \{(\mathbf{s}^{\hat{+}},\mathbf{R}) \mid \mathbf{R} \subseteq \mathcal{O}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\emptyset} \wedge \forall \mathbf{m},\mathbf{W}'.(\mathbf{W},\mathbf{m}) \in \mathbf{R} \wedge \mathbf{W}' \supseteq \mathbf{W}) \implies (\mathbf{W}',\mathbf{m}) \in \mathbf{R}\} \\
 & \quad \cup \{(\mathbf{t},\mathbf{R}) \mid \mathbf{R} \subseteq Atom^{\hat{\uparrow}}[\mathbf{t}]^{\Sigma}_{\emptyset} \wedge \nexists \mathbf{s}.\mathbf{t} = \mathbf{s}^{\hat{+}} \wedge \forall (\mathbf{W},\mathbf{m}) \in \mathbf{R}, \forall \mathbf{W}' \supseteq \mathbf{W}.(\mathbf{W}',\mathbf{m}) \in \mathbf{R} \wedge \\
 & \qquad \forall \mathbf{E}^{\not\lambda},\mathbf{s}'.(\mathbf{W},\mathbf{E}^{\not\lambda}) \in Wf^{\hat{\uparrow}\mathrm{ctx}\ \Sigma}[\mathbf{t},\mathbf{s}'^{\hat{+}}] \implies (\mathbf{W},\mathbf{E}^{\not\lambda}[\mathbf{m}]) \in \mathcal{O}^{\hat{\uparrow}}[\![\mathbf{s}'^{\hat{+}}]\!]^{\Sigma}_{\emptyset}\} \\
Rel^{\hat{\uparrow}\ \Sigma}_{\kappa_1 \to \kappa_2} & = \{(\mathbf{t}_1,\mathbf{R_1}) \mid \forall (\mathbf{t}_2,\mathbf{R_2}) \in Rel^{\hat{\uparrow}\ \Sigma}_{\kappa_1}.((\mathbf{t}_1\,\mathbf{t}_2),\mathbf{R_1}\ (\mathbf{t}_2,\mathbf{R_2})) \in Rel^{\hat{\uparrow}\ \Sigma}_{\kappa_2} \wedge \\
 & \qquad \forall (\mathbf{t}'_2,\mathbf{R}'_2) \in Rel^{\hat{\uparrow}\ \Sigma}_{\kappa_1}.(\mathbf{t}_2,\mathbf{R_2}) \equiv^{\Sigma}_{\kappa_1} (\mathbf{t}'_2,\mathbf{R}'_2) \implies (\mathbf{R_1}\ (\mathbf{t}_2,\mathbf{R_2})) \equiv^{\Sigma}_{\kappa_2} (\mathbf{R_1}\ (\mathbf{t}'_2,\mathbf{R}'_2))\}
\end{array}
$$

Figure 15. F_ω to DCC: Well-Formed Relations

The logical relation considers terms of translation type and non-translation type separately. The essence of the relation is that a term of translation type belongs to the relation $\mathcal{E}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta}$ if it is back-translatable, while a term of non-translation type belongs to the relation $\mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta}$ if plugging the term into a valid $\mathbf{E}^{\not\lambda}$ context results in a term that is back-translatable. We use a $\top\top$-closed-style relation to formalize this. In particular, we say that $\mathbf{E}^{\not\lambda}$ belongs to the continuation relation $\mathcal{K}^{\hat{\uparrow}}[\![\mathbf{t},\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta}$ if, given any $\mathbf{u} \in \mathcal{V}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta}$, $\mathbf{E}^{\not\lambda}[\mathbf{u}]$ 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 $\mathbf{E}^{\not\lambda}[\mathbf{case\ m\ of\ inj_1\ x_1.\ m_1 \parallel inj_2\ x_2.\ m_2}]$. There is no $\mathcal{V}^{\hat{\uparrow}}[\![\mathbf{s}^{+}]\!]^{\Sigma}_{\delta}$ relation for translation types. The relation $\mathcal{V}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta}$ for non-translation types is completely standard and ensures that subterms belong to the $\mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta}$ 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.*, $\mathbf{k} : \mathbf{s}^{\hat{+}} \to \mathsf{T}_\ell\ \mathbf{s}^{\hat{+}}$. Since target terms can only be back-translated under certain term environments—*i.e.*, the environments $\mathbf{G_k}$ and $\Gamma^{\hat{+}}$—we use worlds $\mathbf{W}$ to keep track of these environments as they are extended during the back-translation.

We define $Rel^{\hat{\uparrow}\ \Sigma}_{\kappa}$ 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 $\mathbf{t}$ at kind $*$, a well-formed relation $\mathbf{R}$ guarantees that for all contexts $\mathbf{E}^{\not\lambda}$, if filling $\mathbf{E}^{\not\lambda}$ with a value of type $\mathbf{t}$ results in a back-translatable term, so does filling $\mathbf{E}^{\not\lambda}$ with elements of $\mathbf{R}$. 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_ω definition of relation equivalence.

Finally, since this logical relation is for F_ω terms, we require an interpretation of the kinding judgment as is the case in our F_ω relation in §5.2. These relations, defined in Figure 17, are standard.

The definition of the top-level back-translation logical relation, $\mathbf{D}_\ell,\Delta;\mathbf{G}_\ell,\mathbf{G}_{\preceq},\mathbf{G_k},\Gamma^{\hat{+}},\mathbf{G_k},\Gamma \models^{\hat{\uparrow}} \mathbf{m} : \mathbf{t}$, first closes all free variables in Δ and Γ. We omit the definitions of $\mathcal{D}^{\hat{\uparrow}}[\![\Delta]\!]^{\Sigma}$ and $\mathcal{G}^{\hat{\uparrow}}[\![\Gamma]\!]^{\Sigma}_{\delta}$, which are standard. We also permit closing some of the variables in $\Gamma^{\hat{+}}$. 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-HOLE$^{\not\lambda}$, 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-typed F_ω term, as long as it is open with respect to Σ and the other environments required by the back-translation, belongs to the logical relation.

$$
\begin{array}{ll}
\mathcal{E}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta} & = \mathcal{O}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta} \\
\mathcal{V}^{\hat{\uparrow}}[\![\mathbf{t}' \to \mathbf{t}]\!]^{\Sigma}_{\delta} & = \{(\mathbf{W},\lambda\mathbf{x}{:}\mathbf{t}'.\mathbf{m}) \in Atom^{\hat{\uparrow}}[\mathbf{t}' \to \mathbf{t}]^{\Sigma}_{\delta} \mid \\
 & \qquad \forall \mathbf{W}',\mathbf{m}'.\mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}',\mathbf{m}') \in \mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}']\!]^{\Sigma}_{\delta} \implies \\
 & \qquad (\mathbf{W}',\mathbf{m}[\mathbf{m}'/\mathbf{x}]) \in \mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta}\} \\
\mathcal{V}^{\hat{\uparrow}}[\![\forall\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{t}]\!]^{\Sigma}_{\delta} & = \{(\mathbf{W},\boldsymbol{\Lambda}\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{m}) \in Atom^{\hat{\uparrow}}[\forall\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{t}]^{\Sigma}_{\delta} \mid \\
 & \qquad \forall \mathbf{W}',\mathbf{t}',\mathbf{R}.\mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{t}',\mathbf{R}) \in Rel^{\hat{\uparrow}\ \Sigma}_{\boldsymbol{\kappa}} \implies \\
 & \qquad (\mathbf{W}',\mathbf{m}[\mathbf{t}'/\boldsymbol{\alpha}]) \in \mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta[\boldsymbol{\alpha} \mapsto (\mathbf{t}',\mathbf{R})]}\} \\
\mathcal{V}^{\hat{\uparrow}}[\![\mathbf{t}\ \mathbf{t}']\!]^{\Sigma}_{\delta} & = (\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t} :: \boldsymbol{\kappa}' \to *]\!]^{\Sigma}_{\delta}\ (\boldsymbol{\delta}_1(\mathbf{t}'),\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t}' :: \boldsymbol{\kappa}']\!]^{\Sigma}_{\delta})) \\
\mathcal{E}^{\hat{\uparrow}}[\![\boldsymbol{\alpha}]\!]^{\Sigma}_{\delta} & = \boldsymbol{\delta}_{\mathbf{R}}(\boldsymbol{\alpha}) \\
\mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta} & = \{(\mathbf{W},\mathbf{m}) \in Atom^{\hat{\uparrow}}[\mathbf{t}]^{\Sigma}_{\delta} \mid \forall \mathbf{W}',\mathbf{E}^{\not\lambda},\mathbf{s}^{\hat{+}}. \\
 & \qquad \mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}',\mathbf{E}^{\not\lambda}) \in \mathcal{K}^{\hat{\uparrow}}[\![\mathbf{t},\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta} \implies \\
 & \qquad (\mathbf{W}',\mathbf{E}^{\not\lambda}[\mathbf{m}]) \in \mathcal{O}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta}\} \\
\mathcal{K}^{\hat{\uparrow}}[\![\mathbf{t},\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta} & = \{(\mathbf{W},\mathbf{E}^{\not\lambda}) \in Atom^{\hat{\uparrow}\mathrm{ctx}}\left[\mathbf{t},\mathbf{s}^{\hat{+}}\right]^{\Sigma}_{\delta} \mid \forall \mathbf{W}',\mathbf{u}. \\
 & \qquad \mathbf{W}' \supseteq \mathbf{W} \wedge (\mathbf{W}',\mathbf{u}) \in \mathcal{V}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta} \implies \\
 & \qquad (\mathbf{W}',\mathbf{E}^{\not\lambda}[\mathbf{u}]) \in \mathcal{O}^{\hat{\uparrow}}[\![\mathbf{s}^{\hat{+}}]\!]^{\Sigma}_{\delta}\}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{D}_\ell,\Delta;\mathbf{G}_\ell,\mathbf{G}_{\preceq},\mathbf{G_k},\Gamma^{\hat{+}},\Gamma \models^{\hat{\uparrow}} \mathbf{m} : \mathbf{t} \stackrel{\mathrm{def}}{=} \\
\quad \forall \Gamma^{\hat{+}}_1,\Gamma^{\hat{+}}_2,\delta,\gamma,\boldsymbol{\gamma}.\ \Gamma^{\hat{+}}_1 \uplus \Gamma^{\hat{+}}_2 = \Gamma^{\hat{+}} \wedge \delta \in \mathcal{D}^{\hat{\uparrow}}[\![\Delta]\!]^{\mathbf{D}_\ell} \wedge \\
\qquad ((\mathbf{G_k};\Gamma^{\hat{+}}_2),\gamma) \in \mathcal{G}^{\hat{\uparrow}}[\![\Gamma^{\hat{+}}_1]\!]^{\mathbf{D}_\ell;\mathbf{G}_\ell,\mathbf{G}_{\preceq}}_{\delta} \wedge \\
\qquad ((\mathbf{G_k};\Gamma^{\hat{+}}_2),\gamma) \in \mathcal{G}^{\hat{\uparrow}}[\![\Gamma]\!]^{\mathbf{D}_\ell;\mathbf{G}_\ell,\mathbf{G}_{\preceq}}_{\delta} \implies \\
\qquad\quad ((\mathbf{G_k};\Gamma^{\hat{+}}_2),\delta(\gamma(\mathbf{m}))) \in \mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\mathbf{D}_\ell;\mathbf{G}_\ell,\mathbf{G}_{\preceq}}_{\delta}
\end{array}
$$

Figure 16. F_ω to DCC: Back-Translation Logical Relation

$$
\begin{array}{ll}
\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t} :: *]\!]^{\Sigma}_{\delta} & = \mathcal{E}^{\hat{\uparrow}}[\![\mathbf{t}]\!]^{\Sigma}_{\delta} \\
\quad \text{if } \mathbf{t} \in \{\mathbf{1},\boldsymbol{\alpha},\mathbf{t}_1 + \mathbf{t}_2,\mathbf{t}_1 \times \mathbf{t}_2,\mathbf{t}_1 \to \mathbf{t}_2,\forall\boldsymbol{\alpha}{::}\boldsymbol{\kappa}.\mathbf{t}\} & \\
\mathcal{T}^{\hat{\uparrow}}[\![\boldsymbol{\alpha} :: \boldsymbol{\kappa}_1 \to \boldsymbol{\kappa}_2]\!]^{\Sigma}_{\delta} & = \boldsymbol{\delta}_{\mathbf{R}}(\boldsymbol{\alpha}) \\
\mathcal{T}^{\hat{\uparrow}}[\![\boldsymbol{\lambda}\boldsymbol{\alpha}{::}\boldsymbol{\kappa}_1.\mathbf{t} :: \boldsymbol{\kappa}_1 \to \boldsymbol{\kappa}_2]\!]^{\Sigma}_{\delta} & = \lambda_R\tau.\{\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t} :: \boldsymbol{\kappa}_2]\!]^{\Sigma}_{\delta[\boldsymbol{\alpha}::\boldsymbol{\kappa}_1 \mapsto \tau]}\} \\
\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t}_1\,\mathbf{t}_2 :: \boldsymbol{\kappa}_2]\!]^{\Sigma}_{\delta} & = (\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t}_1 :: \boldsymbol{\kappa}_1 \to \boldsymbol{\kappa}_2]\!]^{\Sigma}_{\delta} \\
 & \qquad (\boldsymbol{\delta}(\mathbf{t}_2),\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t}_2 :: \boldsymbol{\kappa}_1]\!]^{\Sigma}_{\delta}))
\end{array}
$$

Figure 17. F_ω to DCC: Kinding Interpretation

Lemma 7.1 (Type interpretation is well-formed)
If $\Delta \vdash \mathbf{t} :: \boldsymbol{\kappa}$ *and* $\delta \in \mathcal{D}^{\hat{\uparrow}}[\![\Delta]\!]^{\Sigma}$ *then*

1. $(\boldsymbol{\delta}(\mathbf{t}),\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t} :: \boldsymbol{\kappa}]\!]^{\Sigma}_{\delta}) \in Rel^{\hat{\uparrow}\ \Sigma}_{\boldsymbol{\kappa}}$
2. *If* $\delta' \in \mathcal{D}^{\hat{\uparrow}}[\![\Delta]\!]^{\Sigma}$ *such that* $\delta \equiv^{\mathbf{D};\mathbf{G}} \delta'$, *then* $\mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t} :: \boldsymbol{\kappa}]\!]^{\Sigma}_{\delta} \equiv^{\Sigma}_{\boldsymbol{\kappa}} \mathcal{T}^{\hat{\uparrow}}[\![\mathbf{t} :: \boldsymbol{\kappa}]\!]^{\Sigma}_{\delta'}$

Lemma 7.2 (Fundamental property of logical relation)
If $\mathbf{D}_\ell, \mathbf{\Delta}; \mathbf{G}_\ell, \mathbf{G}_{\preceq}, \mathbf{G_k}, \Gamma^{\hat{+}}, \mathbf{G_k}, \mathbf{\Gamma} \vdash \mathbf{m} : \mathsf{t}$ *then*
$\mathbf{D}_\ell, \mathbf{\Delta}; \mathbf{G}_\ell, \mathbf{G}_{\preceq}, \mathbf{G_k}, \Gamma^{\hat{+}}, \mathbf{G_k}, \mathbf{\Gamma} \models^{\hat{\uparrow}} \mathbf{m} : \mathsf{t}$

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

Corollary 7.3 (Back-translation exists (1))
If $\mathbf{\Sigma_D}; \mathbf{\Sigma_G}, \mathbf{G_k}, \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$ *then* $\exists \mathsf{e}.\ \mathbf{\Sigma}; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}$.

Corollary 7.4 (Back-translation exists (2))
If $\mathbf{\Sigma_D}; \mathbf{\Sigma_G} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$ *then* $\exists \mathsf{e}.\ \mathbf{\Sigma}; \cdot; \cdot \vdash \mathbf{m} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}$.

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 Corrollary 7.6. Note that in order to have a strong enough induction hypothesis we first prove Lemma 7.5, which quantifies over arbitrary $\mathbf{G_k}$ environments. We substitute variables in $\mathbf{G_k}$ with appropriate $\eta_{\mathbf{k}}^{\ell,\mathsf{s}}$.

Lemma 7.5
Let $\gamma_k = \{\mathbf{k} \mapsto \eta_{\mathbf{k}}^{\ell,\mathsf{s}} \mid \mathbf{k} : \mathsf{s}^{\hat{+}} \to (\mathsf{T}_\ell\ \mathsf{s})^{\hat{+}} \in \mathbf{G_k}\}$
$\delta = \{\boldsymbol{\alpha}_\ell \mapsto \hat{\boldsymbol{\alpha}}_\ell \mid \ell \in \mathcal{L}_\ell\} \cup \{\boldsymbol{\alpha}_{\preceq} \mapsto \hat{\boldsymbol{\alpha}}_{\preceq}\}$.
If $\mathbf{\Sigma_D}; \mathbf{\Sigma_G}, \mathbf{G_k}, \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$ *then*
$\exists \mathsf{e}. \mathbf{\Sigma}; \mathbf{G_k}; \Gamma^{\hat{+}} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}$ *and* $\Gamma \mid \mathbf{\Sigma} \vdash \mathsf{e} \simeq \gamma_k(\mathbf{m}) : \mathsf{s} \mid \delta$.

Corollary 7.6 (Back-translation preserves semantics)
If $\mathbf{\Sigma_D}; \mathbf{\Sigma_G} \vdash \mathbf{m} : \mathsf{s}^{\hat{+}}$ *then*
$\exists \mathsf{e}. \mathbf{\Sigma}; \cdot; \cdot \vdash \mathbf{m} : \mathsf{s}^{\hat{+}} \uparrow \mathsf{e}$ *and* $\cdot \mid \mathbf{\Sigma} \vdash \mathsf{e} \simeq \mathbf{m} : \mathsf{s} \mid \delta$

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 $\mathbf{m_1}$ and $\mathbf{m_2}$, 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 ($\approx_\zeta$ preservation and reflection)
Let $\Gamma \vdash \mathsf{e}_1 : \mathsf{s} \rightsquigarrow \mathbf{m_1}$ *and* $\Gamma \vdash \mathsf{e}_2 : \mathsf{s} \rightsquigarrow \mathbf{m_2}$.
1. *If* $\Gamma \vdash \mathsf{e}_1 \approx_\zeta \mathsf{e}_2 : \mathsf{s}$, *then* $\mathcal{L}_\ell^+; \mathcal{L}_{\sqsubseteq}^+, \preceq^+, \Gamma^+ \vdash \mathbf{m_1} \approx_\zeta \mathbf{m_2} : \mathsf{s}^+$.
2. *If* $\mathcal{L}_\ell^+; \mathcal{L}_{\sqsubseteq}^+, \preceq^+, \Gamma^+ \vdash \mathbf{m_1} \approx_\zeta \mathbf{m_2} : \mathsf{s}^+$, *then* $\Gamma \vdash \mathsf{e}_1 \approx_\zeta \mathsf{e}_2 : \mathsf{s}$.

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

Theorem 5.2 (Noninterference)
If $\Gamma \vdash \mathsf{e} : \mathsf{s}$ *then* $\forall \zeta.\ \Gamma \vdash \mathsf{e} \approx_\zeta \mathsf{e} : \mathsf{s}$.
Indirect Proof of Noninterference:
By correctness of the translation:
$\Gamma \vdash \mathsf{e} : \mathsf{s} \rightsquigarrow \mathbf{m}$, $\Gamma \vdash \mathsf{e} \simeq \mathbf{m} : \mathsf{s}$, and $\mathcal{L}_\ell^+; \mathcal{L}_{\sqsubseteq}^+, \preceq^+, \Gamma^+ \vdash \mathbf{m} : \mathsf{s}^+$.
By parametricity, $\mathcal{L}_\ell^+; \mathcal{L}_{\sqsubseteq}^+, \preceq^+, \Gamma^+ \vdash \mathbf{m} \approx \mathbf{m} : \mathsf{s}^+$.
Since $\mathcal{L}_\ell^+; \mathcal{L}_{\sqsubseteq}^+, \preceq^+, \Gamma^+ \vdash \mathbf{m} \approx \mathbf{m} : \mathsf{s}^+$ quantifies over *all* ρ, γ, γ', the *particular* implementations we use in the observer-sensitive definition work, so: $\mathcal{L}_\ell^+; \mathcal{L}_{\sqsubseteq}^+, \preceq^+, \Gamma^+ \vdash \mathbf{m} \approx_\zeta \mathbf{m} : \mathsf{s}^+$.
By reflection, $\Gamma \vdash \mathsf{e} \approx_\zeta \mathsf{e} : \mathsf{s}$. □

8. Related Work, Future Work, and Conclusion

In §3, we examined the counterexample Shikuma and Igarashi [17, 18] gave to show that Tse and Zdancewic's translation fails to preserve observer-sensitive equivalence. In that work, they also gave a noninterference-preserving translation, much like 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 λ-calculus with operations for sealing at level ℓ and unsealing—which they proved equivalent to a variant of DCC introduced by Tse and Zdancewic [19] called $\mathrm{DCC}_{\mathrm{pc}}$ for "DCC with protection contexts." $\mathrm{DCC}_{\mathrm{pc}}$ has a lattice of monads like DCC and associated η_ℓ and bind operations but a different type system. $\mathrm{DCC}_{\mathrm{pc}}$ typing judgments have the form $\Gamma; \pi \vdash \mathsf{e} : \mathsf{s}$ where $\pi ::= \cdot \mid \pi, \ell$ for $\ell \in \mathcal{L}_\ell$ is the *protection context*. The typing rules for η_ℓ and bind are:

$$\frac{\Gamma; \pi, \ell \vdash \mathsf{e} : \mathsf{s}}{\Gamma; \pi \vdash \eta_\ell\ \mathsf{e} : \mathsf{T}_\ell\ \mathsf{s}} \qquad \frac{\Gamma; \pi \vdash \mathsf{e}_1 : \mathsf{T}_\ell\ \mathsf{s}_1 \quad \Gamma, \mathsf{x} : \mathsf{s}_1; \pi \vdash \mathsf{e}_2 : \mathsf{s}_2 \quad \pi \vdash \ell \preceq \mathsf{s}_2}{\Gamma; \pi \vdash \mathsf{bind}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 : \mathsf{s}_2}$$

Note that the premise $\pi \vdash \ell \preceq \mathsf{s}_2$ means that either $\ell \preceq \mathsf{s}_2$ as usual *or* $\ell \sqsubseteq \ell'$ for some $\ell' \in \pi$. Thus, the following term—which is ill typed in DCC—is well typed in $\mathrm{DCC}_{\mathrm{pc}}$:

$$\mathsf{e} = \eta_\ell\ (\lambda \mathsf{y} : \mathsf{T}_\ell\ \mathsf{bool}.\ \mathsf{bind}\ \mathsf{x} = \mathsf{y}\ \mathsf{in}\ \mathsf{x}) : \mathsf{T}_\ell\ ((\mathsf{T}_\ell\ \mathsf{bool}) \to \mathsf{bool})$$

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 $\mathrm{DCC}_{\mathrm{pc}}$ 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.[4]

In addition to weakening their source language, Shikuma and Igarashi also strengthen their target language: it admits fewer terms compared to System F since their target is a simply-typed λ-calculus, extended with base types to represent each $\ell \in \mathcal{L}_\ell$. They rightly note that Tse and Zdancewic's translation does not use polymorphism in an essential way—that is, the translation makes use of abstract types $\boldsymbol{\alpha}_\ell$ introduced "globally" (at top level) that may easily be replaced with base types as in Shikuma and Igarashi's work. By comparison, our translation does make essential use of polymorphism in the encoding of the monadic type because a continuation's answer type $\boldsymbol{\beta}$ is locally polymorphic. Moreover, we use higher-order parametricity to require the property given by Lemma 5.1.

Our back-translation improves upon Shikuma and Igarashi's "inverse translation" technique [17, 18] in several ways. Our target language F_ω is more expressive than the source—*e.g.*, we can encode arithmetic operations in F_ω 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_ω and DCC. Our back-translation is based on Ahmed and Blume's [5] but is more

[4] 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.

Recursion 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 [5, 12, 14] for DCC and F_ω 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 F_ω 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

[1] M. Abadi. Protection in programming-language translations. In ICALP 1998.

[2] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In POPL 1999.

[3] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In ESOP 2006.

[4] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In ICFP 2008.

[5] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In ICFP 2011.

[6] J. Berdine, P. O'Hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher Order Symbol. Comput.*, 15(2-3):181–208, 2002.

[7] W. J. Bowman and A. Ahmed. Noninterference for free (technical appendix). June 2015. URL `https://perma.cc/RJ9N-B5ZQ`.

[8] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In PLDI 2007

[9] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In POPL 2013.

[10] N. C. Heintze and J. G. Riecke. The SLam Calculus: Programming with secrecy and integrity. In POPL 1998.

[11] A. Kennedy. Securing the .NET programming model. In *APPSEM II Workshop, Industrial Applications Session*, Sept. 2005.

[12] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In POPL 2007.

[13] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[14] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In ESOP 2014.

[15] B. C. Pierce. *Types and Programming Languages*, chapter 30: Higher-Order Polymorphism. MIT Press, 2002.

[16] J. C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[17] N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. In *11th Asian conference on advances in computer science*, 2007.

[18] N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. *Logical Methods in Computer Science*, 4(3:10):1–31, 2008.

[19] S. Tse and S. Zdancewic. Translating dependency into parametricity. In ICFP 2004.

[20] D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Programming*, 20(2):175–210, Mar. 2010.

[21] P. Wadler. Theorems for free! In *ACM Symp. on Functional Programming Languages and Computer Architecture (FPCA)*, Sept. 1989.

[22] J. Zhao, Q. Zhang, and S. Zdancewic. Relational parametricity for a polymorphic linear lambda calculus. In APLAS 2010.