Translating Dependency into Parametricity

Stephen Tse Steve Zdancewic

University of Pennsylvania

Abstract

Abadi et al. introduced the *dependency core calculus* (DCC) as a unifying framework to study many important program analyses such as binding time, information flow, slicing, and function call tracking. DCC uses a lattice of monads and a nonstandard typing rule for their associated *bind* operations to describe the dependency of computations in a program. Abadi et al. proved a *noninterference* theorem that establishes the correctness of DCC's type system and thus the correctness of the type systems for the analyses above.

In this paper, we study the relationship between DCC and the Girard-Reynolds polymorphic lambda calculus (System F). We encode the recursion-free fragment of DCC into F via a type-directed translation. Our main theoretical result is that, following from the correctness of the translation, the parametricity theorem for F implies the noninterference theorem for DCC. In addition, the translation provides insights into DCC's type system and suggests implementation strategies of dependency calculi in polymorphic languages.

1. Introduction

Abadi et al. introduced the *dependency core calculus* (DCC) as a unifying framework to study many important program analyses such as binding time, information flow, slicing, and function call tracking. DCC uses a lattice of monads and a nonstandard typing rule for their associated *bind* operations to describe the dependency of computations in a program.

For example, information-flow security analyses prevent publicly visible outputs of a program from revealing information about confidential inputs [14]. Consider the program `e` with security type as follows:

$$\mathtt{e} : \mathtt{bool}_{\mathtt{H}} \times \mathtt{bool}_{\mathtt{L}} \rightarrow \mathtt{bool}_{\mathtt{H}} \times \mathtt{bool}_{\mathtt{L}}$$

Here, the label `H` indicates *high-security* or *confidential* data and the label `L` indicates *low-security* or *public* data. A correct information-flow analysis must ensure that the low-security output of the function depends only on the low-security input to the function. Or, more formally, if we fix the low-security input to be a constant $\mathtt{e}_0$, the second output does not depend on the first input. This is equivalent to requiring that the following is a constant function:

$$\lambda\mathtt{x{:}bool}_{\mathtt{H}}.\ \mathtt{prj}_2\ (\mathtt{e}\ \texttt{<}\mathtt{x}, \mathtt{e}_0\texttt{>}) : \mathtt{bool}_{\mathtt{H}} \rightarrow \mathtt{bool}_{\mathtt{L}} \qquad (1)$$

If there are no illegal dependencies, `e` is said to satisfy *noninterference*. Note that the high-security output of `e` may depend on either of the two inputs.

DCC describes dependency analyses by interpreting programs using a lattice of monads, rather than a single monad as in Moggi's computational lambda calculus [9]. The lattice order captures permissible dependencies: computation interpreted in a monad higher in the lattice is permitted to depend on data interpreted in a monad lower in the lattice, but not vice-versa. Intuitively, computation higher in the lattice is held abstract with respect to computation lower in the lattice. In our example above, the two-point lattice $\mathtt{L} \sqsubseteq \mathtt{H}$ suffices: `H` outputs may depend on `H` or `L` inputs, but `L` outputs may depend only on `L` inputs. DCC uses the type constructor $\mathtt{T}_\ell$ to denote the monad corresponding to lattice level ℓ, so the type of `e` in DCC is:

$$\mathtt{e} : \mathtt{T}_{\mathtt{H}}\ \mathtt{bool} \times \mathtt{T}_{\mathtt{L}}\ \mathtt{bool} \rightarrow \mathtt{T}_{\mathtt{H}}\ \mathtt{bool} \times \mathtt{T}_{\mathtt{L}}\ \mathtt{bool}$$

The denotational semantics for DCC extensively uses partial equivalence relations (PERs) indexed by the lattice elements, suggesting a connection to Reynolds' PER semantics [13] for the Girard–Reynolds polymorphic lambda calculus [4, 12] (System F). It is, therefore, natural to ask whether the parametric polymorphism in F can express the dependency in DCC.

This paper answers this question affirmatively by giving a type-directed translation of DCC into F in such that the parametricity theorem for F implies the correctness of the dependency analysis for DCC.

The key idea behind our translation is surprisingly simple. In DCC, the type $\mathtt{T}_{\mathtt{H}}\ \mathtt{bool}$ represents a boolean value that is visible only to `H` computations (computations that produce data with label `H`). `L` computations must treat such a value opaquely and hence cannot distinguish between `true` and `false` at the type $\mathtt{T}_{\mathtt{H}}\ \mathtt{bool}$. DCC models this situation via an observation relation that says these two values are *equal* at `L` (written $\eta_{\mathtt{H}}\ \mathtt{true} \sim_{\mathtt{L}} \eta_{\mathtt{H}}\ \mathtt{false} : \mathtt{T}_{\mathtt{H}}\ \mathtt{bool}$) but not at `H`. In fact, $\sim_{\mathtt{L}}$ relates every possible pair of booleans at the type $\mathtt{T}_{\mathtt{H}}\ \mathtt{bool}$, whereas the corresponding `H` observation relation $\sim_{\mathtt{H}}$ is the *identity* relation.

Our translation simulates DCC's observation relations by encoding $\mathtt{T}_{\mathtt{H}}\ \mathtt{bool}$ as $\alpha_{\mathtt{H}} \rightarrow \mathtt{bool}$. An `L` observer translates to a piece of code that does not have access to any values of type $\alpha_{\mathtt{H}}$, hence any function of type $\alpha_{\mathtt{H}} \rightarrow \mathtt{bool}$ may not be applied. Therefore, in such a context, the two functions $\lambda x:\alpha_H.$ `true` and $\lambda x:\alpha_H.$ `false`, which are the translations of η_H `true` and η_H `false`, are indistinguishable.

The contributions of this paper are:

- the development of this translation of DCC into F and proofs of the static and dynamic correctness of this translation;
- a proof of DCC's noninterference derived from F's parametricity theorem;
- a sound extension of DCC's type system suggested by the translation into F; and
- a sketch of how the translation can be used to implement DCC–style types in languages with parametric polymorphism.

Although the correctness of DCC's type system has been proved previously [1], we believe that the proof method proposed here is interesting in its own right. These results not only help to explain how the nonstandard type system in DCC relates to the well-understood abstract types in F, but also provide insights on how to make DCC's type system more expressive. This translation may also lead to implementation strategies of dependency calculi in polymorphic languages. We demonstrate this possibility by giving a Haskell implementation of the translation for the two-point lattice.

It is known that adding `fix` to a language weakens the parametricity theorem [21, 7, 6] and the noninterference theorem [5, 1] because programs may diverge. For most of this paper we focus on the terminating fragment of DCC in order to emphasize the connection between noninterference and parametricity. Section 6.2 discusses the extension to the translation for the full DCC with `fix`.

The remainder of this paper is organized as follows. The next section introduces the source language of our translation, the dependency core calculus. Section 3 presents the translation and proves the correctness of the translation. Section 4 shows that the parametricity theorem for F implies the noninterference theorem for DCC. Section 5 extends DCC's type system with a protection context to make it more expressive. The paper concludes with a prototype implementation in Haskell and a discussion of future and related work.

2. Dependency core calculus

This section describes the recursion-free fragment of dependency core calculus (DCC) and explains how dependency information is tracked in its type system [1]. The following grammar defines the syntax for DCC's contexts, types, values and terms:

$$\begin{array}{lcl}
\Gamma & ::= & \cdot \mid \Gamma, x{:}t \\
s & ::= & 1 \mid s \times s \mid s + s \mid s \rightarrow s \mid T_\ell\ s \\
v & ::= & \texttt{<>} \mid \texttt{<e,e>} \mid \texttt{inj}_i\ e \mid \lambda x{:}s.\ e \mid \eta_\ell\ e \\
e & ::= & v \mid \texttt{prj}_i\ e \mid \texttt{case}\ e\ v\ v \mid x \mid e\ e \mid \texttt{bind}\ x = e\ \texttt{in}\ e
\end{array}$$

DCC is a call-by-name, simply-typed lambda calculus with one additional language construct—a lattice of monads that restrict dependencies in the program. Let $\mathcal{L}$ be a lattice of dependency levels with join $\sqcup$ and order $\sqsubseteq$. We write $\mathcal{L}_\ell$ for the set of lattice labels and $\mathcal{L}_\sqsubseteq$ for the lattice order such that $\mathcal{L} = (\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$. For each element $\ell \in \mathcal{L}$ there is a monad T_ℓ and its corresponding unit η_ℓ and `bind` operations.

The typing rules for constructs other than the monad operations (η_ℓ and `bind`) are completely standard, so we omit them here. The following rules prevent low-level computation from depending on high-level computation:

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash \eta_\ell\ e : T_\ell\ s} \quad \text{(DT-Prot)}$$

$$\frac{\Gamma \vdash e_1 : T_\ell\ s_1 \quad \Gamma, x{:}s_1 \vdash e_2 : s_2 \quad s_2 \succeq \ell}{\Gamma \vdash \texttt{bind}\ x = e_1\ \texttt{in}\ e_2 : s_2} \quad \text{(DT-Bind)}$$

The $\eta_\ell\ e$ operation marks the computation e with the label ℓ, restricting how it interacts with the rest of the program (DT-Prot). The $\texttt{bind}\ x = e_1\ \texttt{in}\ e_2$ operation exposes the computation e_1 hidden inside the monad T_ℓ to the scope of e_2 (DT-Bind). Here, e_2 may depend on e_1, but the results eventually produced by the entire `bind` expression must still be protected from computation with label lower than (or incomparable to) ℓ in the lattice. Operationally, `bind` evaluates e_1 to a value $\eta_\ell\ e$ and then substitutes e for x in e_2:

$$\texttt{bind}\ x = \eta_\ell\ e\ \texttt{in}\ e_2 \longrightarrow e_2\{e/x\}$$

Protection rules $s \succeq \ell$ (type s protects information at level ℓ) enforce the restrictions on dependencies between computations at different levels of the lattice:

$$1 \succeq \ell \quad \text{(P-Unit)}$$

$$\frac{s_1 \succeq \ell \quad s_2 \succeq \ell}{s_1 \times s_2 \succeq \ell} \quad \text{(P-Pair)} \qquad \frac{\ell \not\sqsubseteq \ell' \quad s \succeq \ell}{T_{\ell'}\ s \succeq \ell} \quad \text{(P-Label1)}$$

$$\frac{s_2 \succeq \ell}{s_1 \rightarrow s_2 \succeq \ell} \quad \text{(P-Fun)} \qquad \frac{\ell \sqsubseteq \ell'}{T_{\ell'}\ s \succeq \ell} \quad \text{(P-Label2)}$$

No information can be transmitted by a value of type 1 because there is only one such value, so computations of type 1 protect any ℓ (P-Unit).[1] Information can be transmitted by a product only by examining its components, so a product type protects information when both of its components do (P-Pair). A function will protect the data as long as the return type of the function protects the data (P-Fun).

A monad at a *lower* (or incomparable) level of computation does not protect data at a *higher* monad in the lattice, unless the contents are already protected at the higher level (P-Label1). On the other hand, a monad at a *higher* (or equal) level in the lattice sufficiently protects the results at a lower level (P-Label2).

Note that a sum type, which transmits information via the injection tags, does not protect data at any level. To protect a sum in a `bind` expression, we must put the sum into a monad. For example, the program $\texttt{bind}\ x = \eta_H\ \texttt{inj}_1\ \texttt{<>}\ \texttt{in}\ x$ is insecure because it leaks the high-security value $\texttt{inj}_1\ \texttt{<>}$. The type system will reject the program as ill-typed because of the typing rule DT-Bind: x has type $1 + s$ but the sum type $1 + s$ does not protects information at the high level (that is, $1 + s \not\succeq H$).

To make examples in the rest of the paper more readable, we define some syntactic sugar for Boolean values: `bool` =

[1]Note that if this language permitted diverging computations, then information could be transmitted via termination. The full DCC calculus includes *lifted* types to distinguish between total and partial types.

$$\begin{array}{rcll}
[\![1 \succeq \ell]\!](x{:}t, m_1, m_2) & = & \texttt{<>} & \text{(LP-Unit)} \\
[\![s_1 \times s_2 \succeq \ell]\!](x{:}t, m_1, m_2) & = & \texttt{<}[\![s_1 \succeq \ell]\!](x{:}t, m_1, \mathtt{prj}_1\ m_2), [\![s_2 \succeq \ell]\!](x{:}t, m_1, \mathtt{prj}_2\ m_2)\texttt{>} & \text{(LP-Pair)} \\
[\![s_1 \to s_2 \succeq \ell]\!](x{:}t, m_1, m_2) & = & \lambda x_0{:}s_1^\dagger.\ [\![s_2 \succeq \ell]\!](x{:}t, m_1, m_2\ x_0) \quad (\text{fresh } x_0) & \text{(LP-Fun)} \\
[\![T_{\ell'}\ s \succeq \ell]\!](x{:}t, m_1, m_2) & = & \lambda x_0{:}\alpha_{\ell'}.\ [\![s \succeq \ell]\!](x{:}t, m_1, m_2\ x_0) \quad (\text{fresh } x_0,\ \ell \not\sqsubseteq \ell') & \text{(LP-Label1)} \\
[\![T_{\ell'}\ s \succeq \ell]\!](x{:}t, m_1, m_2) & = & \lambda x_0{:}\alpha_{\ell'}.\ (\lambda x{:}t.\ m_2\ x_0)\ (m_1\ (k_{\ell'\ell}\ x_0)) \quad (\text{fresh } x_0,\ \ell \sqsubseteq \ell') & \text{(LP-Label2)}
\end{array}$$

Figure 1: Protection translation

$1 + 1$, `true` $=$ $\mathtt{inj}_1\ m_1$ and `false` $= m_2$, where both m_1 and m_2 have type `unit`, and, for fresh variables x_1 and x_2,

$$\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 = \texttt{case } e\ (\lambda x_1.e_1)\ (\lambda x_2.e_2)$$

3. DCC to F

This section describes a type-directed translation that implements DCC's monads using parametric polymorphism. The target of the translation is the polymorphic lambda calculus (System F) extended with unit, products and sums. The following grammar defines the syntax for F's type contexts, term contexts, types, values and terms:

$$\begin{array}{rcl}
\Delta & ::= & \cdot \mid \Delta, \alpha \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}t \\
t & ::= & 1 \mid t \times t \mid t + t \mid t \to t \mid \alpha \mid \forall\alpha.\ t \\
u & ::= & \texttt{<>} \mid \texttt{<}m,m\texttt{>} \mid \mathtt{inj}_i\ m \mid \lambda x{:}t.\ m \mid \Lambda\alpha.\ m \\
m & ::= & u \mid \mathtt{prj}_i\ m \mid \mathtt{case}\ m\ u\ u \mid x \mid m\ m \mid m\ [t]
\end{array}$$

We use the standard type system and the call-by-name dynamic semantics for F [8, 10]. We write its typing judgment as $\Delta; \Gamma \vdash m : t$ and its evaluation as $m_1 \longrightarrow m_2$.

Although both languages have the same basic features, only DCC has the dependency constructs ($\eta_\ell\ e$ and $\mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2$) while F has the parametric polymorphism ($\Lambda\alpha.e$ and $e\ [t]$). The key observation for the translation is that a function whose input type is abstract cannot be applied without an argument of the appropriate type. Hence, if we keep that input type abstract throughout the translation, such functions can hide expressions in their body and the type system can ensure that other parts of the program do not depend on the expressions inside.[2]

3.1 Translation

Let us start by defining the type translation $s^\dagger = t$ and the term translation $e^* = m$. As both DCC and F have the same semantics for the basic constructs, the following rules simply recurse on the structure of the terms and do not perform any interesting translation:

$$\begin{array}{rcl}
1^\dagger & = & 1 \\
(s_1 \times s_2)^\dagger & = & s_1^\dagger \times s_2^\dagger \\
(s_1 + s_2)^\dagger & = & s_1^\dagger + s_2^\dagger \\
(s_1 \to s_2)^\dagger & = & s_1^\dagger \to s_2^\dagger
\end{array}$$

[2]Therefore, we are actually translating *in*-dependency into parametricity.

$$\begin{array}{rcl}
\texttt{<>}^* & = & \texttt{<>} \\
\texttt{<}e_1, e_2\texttt{>}^* & = & \texttt{<}e_1^*, e_2^*\texttt{>} \\
(\mathtt{prj}_i\ e)^* & = & \mathtt{prj}_i\ e^* \\
(\mathtt{inj}_i\ e)^* & = & \mathtt{inj}_i\ e^* \\
(\mathtt{case}\ e\ v_1\ v_2)^* & = & \mathtt{case}\ e^*\ v_1^*\ v_2^* \\
x^* & = & x \\
(\lambda x{:}s.\ e)^* & = & \lambda x{:}s^\dagger.\ e^* \\
(e_1\ e_2)^* & = & e_1^*\ e_2^*
\end{array}$$

Now, we translate a protection under label ℓ as follows:

$$\begin{array}{rcl}
(T_\ell\ s)^\dagger & = & \alpha_\ell \to s^\dagger \\
(\eta_\ell\ e)^* & = & \lambda x{:}\alpha_\ell.\ e^* \quad (\text{fresh } x)
\end{array}$$

We can understand a value of type α_ℓ as a key needed to access the data e. Since α_ℓ is abstract, the only way to access the data under protection is to apply the function closure to the right key. We will formalize this noninterference property of DCC in the next section. A privileged computation that has a key high in the lattice, however, should be able to use that key to access values lower in the lattice. We allow such downgrading of keys via coercion functions $k_{\ell\ell'}$ of type $\alpha_\ell \to \alpha_{\ell'}$. These keys and coercions are generated fresh from the translation of the lattice $\mathcal{L} = (\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$:

$$\begin{array}{rcl}
\mathcal{L}_\ell^\dagger & = & \{\alpha_\ell \mid \ell \in \mathcal{L}_\ell\} \\
\mathcal{L}_\sqsubseteq^\dagger & = & \{k_{\ell\ell'} : \alpha_\ell \to \alpha_{\ell'} \mid \ell' \sqsubseteq \ell \in \mathcal{L}_\sqsubseteq\}
\end{array}$$

The real work is in the translation of `bind`: we have to insert keys and key coercions such that they satisfy the type translation above. Recall that the typing rule of `bind` in DCC is

$$\frac{\Gamma \vdash e_1 : T_\ell\ s_1 \quad \Gamma, x{:}s_1 \vdash e_2 : s_2 \quad s_2 \succeq \ell}{\Gamma \vdash \mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2 : s_2}$$

The type system ensures that a protected expression is eliminated only if the result of the `bind` term protects the label ($s_2 \succeq \ell$). Therefore, we must translate `bind` into F terms that have the corresponding parametric property. Since this property depends on the protection derivation ($s_2 \succeq \ell$), our translation is actually directed by the typing derivation of the term. To simplify the presentation, we enrich the term $\mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2$ with its typing information and define the translation on the enriched term as follows:

$$(\mathtt{bind}\ x : s_1 = e_1\ \mathtt{in}\ e_2 : s_2 \succeq \ell)^* = [\![s_2 \succeq \ell]\!](x{:}s_1^\dagger, e_1^*, e_2^*)$$

Figure 1 shows the protection translation $[\![s \succeq \ell]\!](x : t, m_1, m_2)$, which is defined inductively on the protection derivation $s \succeq \ell$. The first four rules in the figure simply recurse into components using extensionality. Only the last rule reveals the value of e_1^* to e_2^*, which is sound because e_2

is also a protection type $\mathtt{T}_{\ell'}\ \mathtt{s}$ with $\ell \sqsubseteq \ell'$. Such dependency of $\mathtt{e}_2$ on $\mathtt{e}_1$ is translated as follows: we coerce a key of type $\alpha_{\ell'}$ into that of α_ℓ by applying the coercion $\mathtt{k}_{\ell'\ell}$ and then reveal the value of $\mathtt{m}_1 = \mathtt{e}_1^*$ to $\mathtt{m}_2 = \mathtt{e}_2^*$ by function application. Note that $\mathtt{x}$ of type $\mathtt{s}$ is free in $\mathtt{e}_2$; we keep track of the binding variable $\mathtt{x{:}t} = \mathtt{x{:}s}^\dagger$ and close $\mathtt{m}_2 = \mathtt{e}_2^*$ only after the key is applied $(\lambda\mathtt{x{:}t.}\ \mathtt{m}_2\ \mathtt{x}_0)$.

Let us illustrate the translation rules with an example. Consider the typing $\Gamma \vdash \mathtt{e} : \mathtt{s}$ in DCC,

$$\vdash \lambda\mathtt{x}_1{:}\mathtt{T_L\ bool.\ bind\ x}_2 = \mathtt{x}_1\ \mathtt{in}\ \eta_\mathtt{H}\ \mathtt{x}_2 : \mathtt{T_L\ bool} \to \mathtt{T_H\ bool}$$

The translations of the type and the enriched term are:

$$\begin{array}{cl} & (\mathtt{T_L\ bool} \to \mathtt{T_H\ bool})^\dagger \\ = & (\mathtt{T_L\ bool})^\dagger \to (\mathtt{T_H\ bool})^\dagger \\ = & (\alpha_\mathtt{L} \to \mathtt{bool}) \to (\alpha_\mathtt{H} \to \mathtt{bool}) \end{array}$$

$$\begin{array}{cl} & (\lambda\mathtt{x}_1{:}\mathtt{T_L\ bool.} \\ & \mathtt{bind\ x}_2 : \mathtt{bool} = \mathtt{x}_1\ \mathtt{in}\ \eta_\mathtt{H}\ \mathtt{x}_2 : \mathtt{T_H\ bool} \succeq \mathtt{L})^* \\ = & \lambda\mathtt{x}_1{:}\alpha_\mathtt{L} \to \mathtt{bool.}\ [\![\mathtt{T_H\ bool} \succeq \mathtt{L}]\!](\mathtt{x}_2{:}\mathtt{bool}^\dagger, \mathtt{x}_1^*, (\eta_\mathtt{H}\ \mathtt{x}_2)^*) \\ = & \lambda\mathtt{x}_1{:}\alpha_\mathtt{L} \to \mathtt{bool.}\ [\![\mathtt{T_H\ bool} \succeq \mathtt{L}]\!](\mathtt{x}_2{:}\mathtt{bool}, \mathtt{x}_1, \lambda\mathtt{x}_3{:}\alpha_\mathtt{H}.\mathtt{x}_2) \\ = & \lambda\mathtt{x}_1{:}\alpha_\mathtt{L} \to \mathtt{bool.}\ \lambda\mathtt{x}_0{:}\alpha_\mathtt{H}.\ (\lambda\mathtt{x}_2{:}\mathtt{bool.}\ (\lambda\mathtt{x}_3{:}\alpha_\mathtt{H}.\ \mathtt{x}_2)\ \mathtt{x}_0) \\ & (\mathtt{x}_1\ (\mathtt{k_{HL}}\ \mathtt{x}_0)) \end{array}$$

We also need to translate the lattice to obtain $\mathcal{L}_\ell^\dagger = \alpha_\mathtt{H}, \alpha_\mathtt{L}$ and $\mathcal{L}_\sqsubseteq^\dagger = \mathtt{k_{HL}}{:}\alpha_\mathtt{H} \to \alpha_\mathtt{L}$ and use them to close the translated term to obtain the typing $\mathcal{L}_\ell^\dagger; \mathcal{L}_\sqsubseteq^\dagger, \Gamma^\dagger \vdash \mathtt{e}^* : \mathtt{s}^\dagger$ in F:[3]

$$\begin{array}{cl} & \alpha_\mathtt{H}, \alpha_\mathtt{L}; \mathtt{k_{HL}}{:}\alpha_\mathtt{H} \to \alpha_\mathtt{L} \\ \vdash & \lambda\mathtt{x}_1{:}\alpha_\mathtt{L} \to \mathtt{bool.}\ \lambda\mathtt{x}_0{:}\alpha_\mathtt{H}.\ (\lambda\mathtt{x}_2{:}\mathtt{bool.}\ (\lambda\mathtt{x}_3{:}\alpha_\mathtt{H}.\ \mathtt{x}_2)\ \mathtt{x}_0) \\ & (\mathtt{x}_1\ (\mathtt{k_{HL}}\ \mathtt{x}_0)) \\ : & (\alpha_\mathtt{L} \to \mathtt{bool}) \to (\alpha_\mathtt{H} \to \mathtt{bool}) \end{array}$$

3.2 Correctness

In this subsection we prove that the translation is correct with respect to its static and dynamic semantics. These correctness theorems are essential in proving the noninterference theorem in the next section.

The static correctness theorem below states that translated terms are well-typed. We only give proof sketches in this paper; complete proofs can be found in our companion technical report [19].

THEOREM 1 (STATIC CORRECTNESS). *If $\Gamma \vdash e : s$ with $(\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$, then*

$$\mathcal{L}_\ell^\dagger; \mathcal{L}_\sqsubseteq^\dagger, \Gamma^\dagger \vdash e^* : s^\dagger$$

PROOF. We prove the theorem by induction on the typing derivation and use the following lemma in case of `bind`. □

LEMMA 2. *If $\Delta; \Gamma \vdash m_1 : \alpha_\ell \to t$ and $\Delta; \Gamma, x{:}\,t \vdash m_2 : s^\dagger$, then $\Delta; \Gamma \vdash [\![s \succeq \ell]\!](x{:}\,t, m_1, m_2) : s^\dagger$.*

In order to relate the dynamic semantics of the two languages in a declarative style, we define the following translation relation $\mathtt{v} \rightharpoonup \mathtt{u} : \mathtt{t}$ to relate a DCC value $\mathtt{v}$ to an F value $\mathtt{u}$ at DCC type $\mathtt{t}$. It is a logical relation [8] that formalizes the behavioral equivalence of the dynamic semantics.

$$\mathtt{<>} \rightharpoonup \mathtt{<>} : 1 \qquad \text{(DF-Unit)}$$

$$\frac{\mathtt{e}_1 \Rightarrow \mathtt{m}_1 : \mathtt{s}_1 \quad \mathtt{e}_2 \Rightarrow \mathtt{m}_2 : \mathtt{s}_2}{\mathtt{<e_1, e_2>} \rightharpoonup \mathtt{<m_1, m_2>} : \mathtt{s}_1 \times \mathtt{s}_2} \qquad \text{(DF-Pair)}$$

$$\frac{\mathtt{e} \Rightarrow \mathtt{m} : \mathtt{s}_i}{\mathtt{inj}_i\ \mathtt{e} \rightharpoonup \mathtt{inj}_i\ \mathtt{m} : \mathtt{s}_1 + \mathtt{s}_2} \qquad \text{(DF-Inj)}$$

$$\frac{\forall(\mathtt{e} \Rightarrow \mathtt{m} : \mathtt{s}_1).\ \mathtt{v\ e} \Rightarrow \mathtt{u\ m} : \mathtt{s}_2}{\mathtt{v} \rightharpoonup \mathtt{u} : \mathtt{s}_1 \to \mathtt{s}_2} \qquad \text{(DF-Fun)}$$

$$\frac{\forall(\vdash \mathtt{m} : \mathtt{t}).\ \mathtt{e} \Rightarrow \mathtt{u\ m} : \mathtt{s}}{\eta_\ell\ \mathtt{e} \rightharpoonup \mathtt{u} : \mathtt{T}_\ell\ \mathtt{s}} \qquad \text{(DF-Prot)}$$

$$\frac{\mathtt{e} \longrightarrow^* \mathtt{v} \quad \mathtt{m} \longrightarrow^* \mathtt{u} \quad \mathtt{v} \rightharpoonup \mathtt{u} : \mathtt{s}}{\mathtt{e} \Rightarrow \mathtt{m} : \mathtt{s}} \qquad \text{(DF-Term)}$$

The last rule defines the translation relation $\mathtt{e} \Rightarrow \mathtt{m} : \mathtt{s}$ for terms by extending the corresponding relation for values $\mathtt{v} \rightharpoonup \mathtt{u} : \mathtt{t}$ with evaluation. We will similarly extend other logical relations for values to terms by evaluation in later sections.

Note that the relation above is defined for closed types and terms. Our translated types and terms, however, have free type variables α_ℓ for the lattice labels and free term variables $\mathtt{k}_{\ell\ell'}$ for the lattice order. By the static correctness theorem, we know that those types and terms are closed under $\mathcal{L}_\ell^\dagger$ and $\mathcal{L}_\sqsubseteq^\dagger$.

At runtime, we need to link the open term with a well-typed implementation of the keys and the coercions. We formalize this linking using term and type substitutions. Let γ denote a finite map from term variables to values and let δ denote a finite map from type variables to closed types:

$$\begin{array}{rcl} \gamma & ::= & \cdot \mid \mathtt{x} \mapsto \mathtt{v} \\ \delta & ::= & \cdot \mid \alpha \mapsto \mathtt{t} \end{array}$$

We define $\delta(\Gamma)$ as the pointwise application of δ to the range of Γ such that $(\delta(\Gamma))(\mathtt{x}) = \delta(\Gamma(\mathtt{x}))$. Also, we generalize the definition of the translation relation for values $\mathtt{v} \rightharpoonup \mathtt{u} : \mathtt{t}$ to term substitutions such that $\gamma \Rightarrow \gamma' : \delta(\Gamma)$ if $\gamma(\mathtt{x}) \Rightarrow \gamma'(\mathtt{x}) : \delta(\Gamma(\mathtt{x}))$ for all $\mathtt{x} \in \mathtt{dom}(\gamma) = \mathtt{dom}(\gamma') = \mathtt{dom}(\Gamma)$. Formally we state the typing requirements of the substitutions as:

$$\begin{array}{lcl} \gamma \models \Gamma & \text{iff} & \vdash \gamma(\mathtt{x}) : \Gamma(\mathtt{x}) \\ \delta \models \Delta & \text{iff} & \mathrm{dom}(\delta) = \Delta \end{array}$$

At last, the dynamic correctness theorem below states that if a term and a linking are well-typed, then the source and the target terms have the same dynamic behavior.

THEOREM 3 (DYNAMIC CORRECTNESS). *If $\Gamma \vdash e : s$ with $(\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$, $\delta_0 \models \mathcal{L}_\ell^\dagger$, $\gamma \Rightarrow \gamma' : \delta_0(\Gamma)$, and $\gamma_0 \models \mathcal{L}_\sqsubseteq^\dagger$, then*

$$\gamma(e) \Rightarrow \delta_0\gamma_0\gamma'(e^*) : s$$

PROOF. We prove the theorem by induction on the typing derivation and use the following lemma in case of `bind`. □

LEMMA 4. *If $\Delta; \Gamma, x{:}\,t \vdash m_2 : s$, $\delta_0 \models \Delta$, $\gamma_0 \models \Gamma$, $m_1\ m_0 \longrightarrow^* m$ and $e_2 \Rightarrow m_2\{m/x\} : s$, then $e_2 \Rightarrow \delta_0\gamma_0[\![s \succeq \ell]\!](x : t, m_1, m_2)$.*

[3]Actually, as the lattice order is reflexive, we should have $\mathcal{L}_\sqsubseteq^\dagger = \mathtt{k_{HL}}{:}\alpha_\mathtt{H} \to \alpha_\mathtt{L}, \mathtt{k_{HH}}{:}\alpha_\mathtt{H} \to \alpha_\mathtt{H}, \mathtt{k_{LL}}{:}\alpha_\mathtt{L} \to \alpha_\mathtt{L}$. We leave out the last two coercions to simplify the presentation in later sections.

4. Theorems

This section proves the main theoretical result of the paper: the parametricity theorem for F implies the noninterference theorem for DCC. Both theorems are defined in terms of logical equivalence of the dynamic semantics; therefore, we will use logical relations again to define related values for DCC and those for F. Then, we come up with a canonical implementation consisting of substitutions for terms, types and relations in F. Using these substitutions we can instantiate parametricity to prove noninterference.

4.1 Logical equivalences

We formalize logical equivalences as logical relations. For DCC, they extend the identity relations, except that data in a monad higher in the lattice is opaque at levels lower in the lattice. We write $\mathtt{v} \sim_\zeta \mathtt{v}' : \mathtt{s}$ to denote two related DCC values $\mathtt{v}$ and $\mathtt{v}'$ at type $\mathtt{s}$ below observation bound ζ. These relations are defined as follows:

$$\mathtt{<>} \sim_\zeta \mathtt{<>} : 1 \qquad \text{(DR-Unit)}$$

$$\frac{\mathtt{e}_1 \approx_\zeta \mathtt{e}_1' : \mathtt{s}_1 \quad \mathtt{e}_2 \approx_\zeta \mathtt{e}_2' : \mathtt{s}_2}{\mathtt{<e_1, e_2>} \sim_\zeta \mathtt{<e_1', e_2'>} : \mathtt{s}_1 \times \mathtt{s}_2} \qquad \text{(DR-Pair)}$$

$$\frac{\mathtt{e} \approx_\zeta \mathtt{e}' : \mathtt{s}_i}{\mathtt{inj}_i\ \mathtt{e} \sim_\zeta \mathtt{inj}_i\ \mathtt{e}' : \mathtt{s}_1 + \mathtt{s}_2} \qquad \text{(DR-Inj)}$$

$$\frac{\forall(\mathtt{e} \approx_\zeta \mathtt{e}' : \mathtt{s}_1).\ \mathtt{v}\ \mathtt{e} \approx_\zeta \mathtt{v}'\ \mathtt{e}' : \mathtt{s}_2}{\mathtt{v} \sim_\zeta \mathtt{v}' : \mathtt{s}_1 \to \mathtt{s}_2} \qquad \text{(DR-Fun)}$$

$$\frac{\ell \not\sqsubseteq \zeta}{\eta_\ell\ \mathtt{e} \sim_\zeta \eta_\ell\ \mathtt{e}' : T_\ell\ \mathtt{s}} \qquad \text{(DR-Label1)}$$

$$\frac{\mathtt{e} \approx_\zeta \mathtt{e}' : \mathtt{s} \quad \ell \sqsubseteq \zeta}{\eta_\ell\ \mathtt{e} \sim_\zeta \eta_\ell\ \mathtt{e}' : T_\ell\ \mathtt{s}} \qquad \text{(DR-Label2)}$$

$$\frac{\mathtt{e} \longrightarrow^* \mathtt{v} \quad \mathtt{e}' \longrightarrow^* \mathtt{v}' \quad \mathtt{v} \sim_\zeta \mathtt{v}' : \mathtt{s}}{\mathtt{e} \approx_\zeta \mathtt{e}' : \mathtt{s}} \qquad \text{(DR-Term)}$$

Except for protection $\eta_\ell\ \mathtt{e}$, the definitions above are standard [8]. The parameter ζ is the context's observation label, capturing the intuition that the equivalence of two terms depends on the label of the observer [20, 5, 23, 15, 11]. For $T_\ell\ \mathtt{s}$, either the observer's label is below the bound ($\ell \not\sqsubseteq \zeta$) in which case all values are related (DR-Label1), or two protected terms are actually related recursively (DR-Label2).

Similarly, we define related values for F. However, instead of labels and bounds for DCC, two F values are related at type $\mathtt{t}$ under relation substitution ρ (which maps type variables to relations), written $\mathtt{v} \sim \mathtt{v} : \mathtt{t} \mid \rho$. These relations are defined as follows:

$$\mathtt{<>} \sim \mathtt{<>} : 1 \mid \rho \qquad \text{(FR-Unit)}$$

$$\frac{\mathtt{m}_1 \approx \mathtt{m}_1' : \mathtt{t}_1 \mid \rho \quad \mathtt{m}_2 \approx \mathtt{m}_2' : \mathtt{t}_2 \mid \rho}{\mathtt{<m_1, m_2>} \sim \mathtt{<m_1', m_2'>} : \mathtt{t}_1 \times \mathtt{t}_2 \mid \rho} \qquad \text{(FR-Pair)}$$

$$\frac{\mathtt{m} \approx \mathtt{m}' : \mathtt{t}_i \mid \rho}{\mathtt{inj}_i\ \mathtt{m} \sim \mathtt{inj}_i\ \mathtt{m}' : \mathtt{t}_1 + \mathtt{t}_2 \mid \rho} \qquad \text{(FR-Inj)}$$

$$\frac{\forall(\mathtt{m} \approx \mathtt{m}' : \mathtt{t}_1 \mid \rho).\ \mathtt{u}\ \mathtt{m} \approx \mathtt{u}'\ \mathtt{m}' : \mathtt{t}_2 \mid \rho}{\mathtt{u} \sim \mathtt{u}' : \mathtt{t}_1 \to \mathtt{t}_2 \mid \rho} \qquad \text{(FR-Fun)}$$

$$\frac{(\mathtt{u}, \mathtt{u}') \in \mathtt{R}}{\mathtt{u} \sim \mathtt{u}' : \alpha \mid (\rho, \alpha \mapsto \mathtt{R})} \qquad \text{(FR-Var)}$$

$$\frac{\begin{array}{l}\forall(\mathtt{R} \in \mathtt{t}_2 \leftrightarrow \mathtt{t}_2').\\ \mathtt{u}\ [\mathtt{t}_2] \approx \mathtt{u}'\ [\mathtt{t}_2'] : \mathtt{t}_1 \mid (\rho, \alpha \mapsto \mathtt{R})\end{array}}{\mathtt{u} \sim \mathtt{u}' : \forall\alpha.\ \mathtt{t}_1 \mid \rho} \qquad \text{(FR-All)}$$

$$\frac{\mathtt{m} \longrightarrow^* \mathtt{u} \quad \mathtt{m}' \longrightarrow^* \mathtt{u}' \quad \mathtt{u} \sim \mathtt{u}' : \mathtt{t} \mid \rho}{\mathtt{m} \approx \mathtt{m}' : \mathtt{t} \mid \rho} \qquad \text{(FR-Term)}$$

The $\approx$ relation is the usual logical relation for System F [21]. Rule FR-All says that ρ maps type variable α to *any* relation respecting the types ($\mathtt{R} \in \mathtt{t}_2 \leftrightarrow \mathtt{t}_2'$). We write $\mathtt{t}_2 \leftrightarrow \mathtt{t}_2'$ to denote the set of all binary relations over values of the types $\mathtt{t}_2$ and $\mathtt{t}_2'$. For example, the empty relation $\emptyset$ and the diagonal relation $\{(\mathtt{<>}, \mathtt{<>})\}$ for unit (which we will use in later sections) are both in the set of all binary relations over `unit` and `unit`.

We can also derive the related values at `bool` in F as follows. The encoding of Boolean in Section 2 and the rules FR-Unit and FR-Inj imply the following equivalence:

$$\frac{\mathtt{m} \approx \mathtt{m}' : 1 \mid \rho}{\mathtt{inj}_i\ \mathtt{m} \sim \mathtt{inj}_i\ \mathtt{m}' : 1 + 1 \mid \rho}$$

That is, at `bool`, logical equivalence implies β-equivalence modulo the definitions of `true` and `false` in Section 2. In particular, the definition does not relate the values `true` and `false`. The same is true for Booleans in DCC.

4.2 Parametricity

System F's parametricity theorem (below) states that, independent of substitutions for terms, types and relations, a well-typed term is related to itself. We write $\rho \in \delta \leftrightarrow \delta'$ as the pointwise extension of $\mathtt{R} \in \mathtt{t}_2 \leftrightarrow \mathtt{t}_2'$ (which is defined in the last subsection) such that $\rho(\alpha) \in \delta(\alpha) \leftrightarrow \delta'(\alpha)$ for all $\alpha \in \mathtt{dom}(\rho) = \mathtt{dom}(\delta) = \mathtt{dom}(\delta')$.

THEOREM 5 (PARAMETRICITY). *If* $\Delta; \Gamma \vdash m : t$ *and* $\delta, \delta' \models \Delta$ *and* $\gamma \approx \gamma' : \Gamma \mid \rho$ *and* $\rho \in \delta \leftrightarrow \delta'$, *then*

$$\delta\gamma(m) \approx \delta'\gamma'(m) : t \mid \rho$$

PROOF. Proofs can be found in standard references [13, 21]. We have a full proof in our notation in the technical report and we use the following substitution lemma when the induction case is type application. □

LEMMA 6. *If* $m \approx m' : t_1 \mid (\rho, \alpha \mapsto [\![t_2]\!]_\rho)$, *then* $m \approx m' : t_1\{t_2/\alpha\} \mid \rho$ *where* $[\![t_2]\!]_\rho = \{(u, u') \mid u \sim u' : t_2 \mid \rho\}$.

To motivate our general proof of noninterference from this parametricity theorem, we demonstrate a simple application using Wadler's *free theorems* [21]. Recall from the introduction, that a DCC term of type $\mathtt{T_H\ bool} \to \mathtt{T_L\ bool}$ translates to a term $\mathtt{m}$ of type $(\alpha_\mathtt{H} \to \mathtt{bool}) \to (\alpha_\mathtt{L} \to \mathtt{bool})$. We use a free theorem to demonstrate that $\mathtt{m}$ is a constant function, which is required for noninterference..

In this instance, we pick the following substitutions for types, terms and relations:

$$\begin{array}{rcl} \delta_0 & = & \alpha_\mathtt{H} \mapsto \mathtt{1}, \alpha_\mathtt{L} \mapsto \mathtt{1} \\ \gamma_0 & = & \mathtt{k_{HL}} \mapsto \lambda\mathtt{x}{:}\mathtt{1.x} \\ \rho_0 & = & \alpha_\mathtt{H} \mapsto \emptyset, \alpha_\mathtt{L} \mapsto \{(\mathtt{<>},\mathtt{<>})\} \end{array}$$

By the parametricity theorem and the typing $\mathcal{L}_\ell^\dagger; \mathcal{L}_\sqsubseteq^\dagger \vdash \mathtt{m} : (\alpha_\mathtt{H} \to \mathtt{bool}) \to (\alpha_\mathtt{L} \to \mathtt{bool})$, we have

$$\delta_0\gamma_0(\mathtt{m}) \approx \delta_0\gamma_0(\mathtt{m}) : (\alpha_\mathtt{H} \to \mathtt{bool}) \to (\alpha_\mathtt{L} \to \mathtt{bool}) \mid \rho_0$$

Applying FR-Fun two times, we have $\delta_0\gamma_0(\mathtt{m})\ \mathtt{m_1}\ \mathtt{m_2} \approx \delta_0\gamma_0(\mathtt{m})\ \mathtt{m}_1'\ \mathtt{m}_2' : \mathtt{bool} \mid \rho_0$ for all $\mathtt{m_1} \approx \mathtt{m}_1' : \alpha_\mathtt{H} \to \mathtt{bool} \mid \rho_0$ and for all $\mathtt{m_2} \approx \mathtt{m_2} : \alpha_\mathtt{L} \mid \rho_0$. Since ρ_0 maps $\alpha_\mathtt{H}$ to the empty relation, $\mathtt{m_1}$ and $\mathtt{m}_1'$ can be different terms (such as $\lambda\mathtt{x}{:}\mathtt{1.\ true}$ and $\lambda\mathtt{x : 1.\ false}$). Yet ρ_0 maps $\alpha_\mathtt{L}$ to the diagonal relation. As the relation $\approx$ of related values at $\mathtt{bool}$ implies β-equivalence, $\delta_0\gamma_0(\mathtt{m})\ \mathtt{m_1}\ \mathtt{m_2}$ is equivalent to $\delta_0\gamma_0(\mathtt{m})\ \mathtt{m}_1'\ \mathtt{m}_2'$ despite the difference in the arguments. In other words, $\mathtt{m}$ is a constant function, or its argument does not interfere with its output.

The key point is that ρ_0 maps the type variable for the high view to be the empty relation ($\alpha_\mathtt{H} \mapsto \emptyset$) and the variable for the low view to be the diagonal relation on $\mathtt{1}$ ($\alpha_\mathtt{L} \mapsto \{(\mathtt{<>},\mathtt{<>})\}$). Because we translate a protection type $\mathtt{T_\ell\ s}$ into a function type $\alpha_\ell \to \mathtt{s}^\dagger$, we have the complete relation over $\mathtt{s}^\dagger$ when $\ell = \mathtt{H}$ and we have the relation defined by $\mathtt{s}^\dagger$ when $\ell = \mathtt{L}$. The complete relation formally captures the requirement that the observer cannot distinguish values in the high view.

Another way to motivate the construction of ρ_0 is the following two isomorphisms of relations: $\emptyset \to \mathtt{R} \cong \top$ and $\mathtt{1} \to \mathtt{R} \cong \mathtt{R}$. The former isomorphism says that a function relation from an empty relation $\emptyset$ to an arbitrary relation $\mathtt{R}$ is isomorphic to the total relation $\top$. Thus, setting $\alpha_\mathtt{H} \mapsto \emptyset$ implies the total relation for the high view, meaning all values are related (see DR-Label1 in Section 4.1). On the other hand, the latter isomorphisms says that a function relation from a diagonal relation $\mathtt{1}$ and an arbitrary relation $\mathtt{R}$ is isomorphism to the relation $\mathtt{R}$ itself. Thus, setting $\alpha_\mathtt{L} \mapsto \mathtt{1}$ implies the relation $\mathtt{R}$ itself for the low view, meaning values are related if they are related recursively (see DR-Label2).

4.3 Noninterference

We generalize the approach above to arbitrary lattices and terms to prove the noninterference theorem. First, we define an appropriate implementation of keys based on the lattice $\mathcal{L}$:

$$\begin{array}{rcl} [\![\mathcal{L}_\ell]\!]_1 & = & \{\alpha_\ell \mapsto \mathtt{1} \mid \ell \in \mathcal{L}_\ell\} \\ [\![\mathcal{L}_\sqsubseteq]\!]_1 & = & \{\mathtt{k}_{\ell'\ell} \mapsto \lambda\mathtt{x}{:}\mathtt{1.x} \mid \ell \sqsubseteq \ell' \in \mathcal{L}_\sqsubseteq\} \\ [\![\mathcal{L}_\ell]\!]_\zeta & = & \left\{ \begin{array}{ll} \alpha_\ell \mapsto \emptyset & \text{if } \ell \not\sqsubseteq \zeta \\ \alpha_\ell \mapsto \{(\mathtt{<>},\mathtt{<>})\} & \text{if } \ell \sqsubseteq \zeta \end{array} \right. \end{array}$$

This canonical implementation substitutes unit for keys and identity functions for key coercions. The relation substitution is the diagonal relation if the observer's label is higher than the data's label, or the empty relation otherwise. This property of $[\![\mathcal{L}_\ell]\!]_\zeta$ is critical in proving Lemma 8 and Lemma 9 below.

The noninterference theorem states that a well-typed term cannot distinguish substitutions of different values higher than the observer's bound. We define γ^* as $\gamma^*(\mathtt{x}) = \mathtt{e}^*$ iff $\gamma(\mathtt{x}) = \mathtt{e}$.

Theorem 7 (Noninterference). *If $\Gamma \vdash \mathtt{e} : \mathtt{s}$ with $(\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$ and $\gamma \approx_\zeta \gamma' : \Gamma$, then*

$$\gamma(\mathtt{e}) \approx_\zeta \gamma'(\mathtt{e}) : \mathtt{s}$$

Proof. By Theorem 1 (Static correctness), the translated term is well-typed: $\mathcal{L}_\ell^\dagger; \mathcal{L}_\sqsubseteq^\dagger, \Gamma^\dagger \vdash \mathtt{e}^* : \mathtt{s}^\dagger$. By the definition of $\delta_0 = [\![\mathcal{L}_\ell]\!]_1$, we have $\delta_0 \models [\![\mathcal{L}_\ell]\!]^\dagger$. By Lemma 8 (Relation correctness), the translated substitutions are related: $\gamma_0\gamma^* \approx \gamma_0\gamma'^* : \mathcal{L}_\sqsubseteq^\dagger, \Gamma^\dagger \mid \rho_0$. Since $\rho_0 = [\![\mathcal{L}_\ell]\!]_\zeta$ is either the empty relation or the diagonal relation, $\rho_0 \in \delta_0 \leftrightarrow \delta_0$. Then, by Theorem 5 (Parametricity), the translated terms are related: $\delta_0\gamma_0\gamma^*(\mathtt{e}^*) \approx \delta_0\gamma_0\gamma'^*(\mathtt{e}^*) : \mathtt{s}^\dagger \mid \rho_0$.

By the definition of $\gamma_0 \models [\![\mathcal{L}_\sqsubseteq]\!]_1$, we have $\gamma_0 \models \mathcal{L}_\sqsubseteq^\dagger$. The result then follows by Theorem 3 (Dynamic correctness) and Lemma 9 (Adequacy). □

Lemma 8 (Relation correctness).

1. *If $\gamma \approx_\zeta \gamma' : \Gamma$ with $(\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$ and $\gamma_0 \models \mathcal{L}_\sqsubseteq$, then $\gamma_0\gamma^* \approx \gamma_0\gamma'^* : \mathcal{L}_\sqsubseteq^\dagger, \Gamma^\dagger \mid [\![\mathcal{L}_\ell]\!]_\zeta$.*
2. *If $\mathtt{e} \approx_\zeta \mathtt{e}' : \mathtt{s}$ with $(\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$ and $\gamma_0 \models \mathcal{L}_\sqsubseteq$, then $\gamma_0(\mathtt{e}^*) \approx \gamma_0(\mathtt{e}'^*) : \mathtt{s}^\dagger \mid [\![\mathcal{L}_\ell]\!]_\zeta$.*

Proof. We prove part 2 by induction on the derivation of the related terms and thus, by inversion, the derivation of the related values $\mathtt{v} \sim_\zeta \mathtt{v}' : \mathtt{s}$. The important case is $\mathtt{s} = \mathtt{T_\ell\ s_0}$. When $\ell \not\sqsubseteq \zeta$ and thus $[\![\mathcal{L}_\ell]\!]_\zeta$ is the empty relation, there do not exist $\mathtt{m}$ and $\mathtt{m}'$ such that $\mathtt{m} \approx \mathtt{m}' : \alpha_\ell \mid [\![\mathcal{L}_\ell]\!]_\zeta$. The substituted values are related at the function types $(\mathtt{T_\ell\ s_0})^\dagger = \alpha_\ell \to \mathtt{s}_0^\dagger$ because the premise $\forall(\mathtt{m} \approx \mathtt{m}' : \alpha_\ell \mid [\![\mathcal{L}_\ell]\!]_\zeta).\ \mathtt{u\ m} \approx \mathtt{u}'\ \mathtt{m}' : \mathtt{s}_0^\dagger \mid [\![\mathcal{L}_\ell]\!]_\zeta$ is vacuously true. When $\ell \sqsubseteq \zeta$, we use the induction hypothesis. □

Lemma 9 (Adequacy). *If $\mathtt{e} \Rightarrow \delta_0(\mathtt{m}) : \mathtt{s}$, $\mathtt{e}' \Rightarrow \delta_0(\mathtt{m}') : \mathtt{s}$ with $(\mathcal{L}_\ell, \mathcal{L}_\sqsubseteq)$, $\delta_0 \models \mathcal{L}_\ell^\dagger$ and $\mathtt{m} \approx \mathtt{m}' : \mathtt{s}^\dagger \mid [\![\mathcal{L}_\ell]\!]_\zeta$, then $\mathtt{e} \approx_\zeta \mathtt{e}' : \mathtt{s}$.*

Proof. We prove the lemma by induction on the type $\mathtt{s}$. When $\mathtt{s} = \mathtt{T_\ell\ s_0}$ and $\ell \sqsubseteq \zeta$ and thus $[\![\mathcal{L}_\ell]\!]_\zeta$ is the diagonal relation, $\mathtt{m} \approx \mathtt{m}' : \alpha_\ell \mid [\![\mathcal{L}_\ell]\!]_\zeta$ for all $\mathtt{m},\mathtt{m}'$. Since $(\mathtt{T_\ell\ s_0})^\dagger = \alpha_\ell \to \mathtt{s}_0^\dagger$ and thus $\mathtt{m}$ and $\mathtt{m}'$ are related at the function types, we have $\mathtt{u\ m} \approx \mathtt{u}'\ \mathtt{m}' : \mathtt{s}^\dagger \mid [\![\mathcal{L}_\ell]\!]_\zeta$ given that $\mathtt{m} \longrightarrow^* \mathtt{u}$ and $\mathtt{m}' \longrightarrow^* \mathtt{u}'$. We can then use induction hypothesis to satisfy DR-Label2. □

In the last subsection, we used the parametricity theorem to prove that a term $\mathtt{m}$ of type $(\mathtt{T_H\ bool} \to \mathtt{T_L\ bool})^\dagger = (\alpha_\mathtt{H} \to \mathtt{bool}) \to (\alpha_\mathtt{L} \to \mathtt{bool})$ is a constant function in F. As an exercise, we want to prove, directly from the noninterference theorem, that a term $\mathtt{e}$ of type $\mathtt{T_H\ bool} \to \mathtt{T_L\ bool}$ is also a constant function in DCC. Suppose $\vdash \mathtt{e} : \mathtt{T_H\ bool} \to \mathtt{T_L\ bool}$. By the noninterference theorem, $\mathtt{e} \approx_\zeta \mathtt{e} : \mathtt{T_H\ bool} \to \mathtt{T_L\ bool}$. Expanding the definition of DR-Fun, we have $\mathtt{e\ e_2} \approx_\zeta \mathtt{e\ e}_2' : \mathtt{T_L\ bool}$ for all $\mathtt{e_2} \approx_\zeta \mathtt{e}_2' : \mathtt{T_H\ bool}$. Since there are only two points in the lattice, we have $\mathtt{L} \sqsubseteq \zeta$. Let us pick $\zeta = \mathtt{L}$ such that $\mathtt{H} \not\sqsubseteq \zeta$. Hence $\mathtt{e}_2$ and $\mathtt{e}_2'$ can be different terms (such as η_ℓ **true** and η_ℓ **false**) but still related by DR-Label1. Yet $\mathtt{e}\ \mathtt{e}_2$ and $\mathtt{e}\ \mathtt{e}_2'$ are related by DR-Label2, implying that both terms evaluate to related Boolean values despite their arguments. Because the relation $\approx$ at type bool implies β-equivalence (up to the definition of bool), it follows that e is a constant function.

5. Extending DCC

Having shown that $\mathtt{T_H\ bool} \to \mathtt{T_L\ bool}$ is a constant function, it is natural to ask: what about functions of other combinations of high booleans and low booleans? We can easily construct both the constant functions and the identity functions of types $\mathtt{T_L\ bool} \to \mathtt{T_L\ bool}$ and $\mathtt{T_H\ bool} \to \mathtt{T_H\ bool}$; therefore, functions of these two types are unconstrained.

The other interesting case is $\mathtt{T_L\ bool} \to \mathtt{T_H\ bool}$. We claim that, from the high observer's view ($\zeta = \mathtt{H}$), $\mathtt{T_L\ bool} \to \mathtt{T_H\ bool}$ is isomorphic to $\mathtt{bool} \to \mathtt{bool}$. This isomorphism gives us confidence that adding labels to a language does not reduce the expressiveness of the language: if we have a function of type $\mathtt{bool} \to \mathtt{bool}$, we can always write an equivalent function of $\mathtt{T_L\ bool} \to \mathtt{T_H\ bool}$ such that the dependency information is explicit.

But, because of typing, we fail to prove such an isomorphism in DCC. This section describes how we instead prove the isomorphism in F and, using insights of the proof, how we extend DCC to type-check the mediating functions for the isomorphism.

5.1 Excursion to parametricity

First, we propose the following mediating functions in DCC for the isomorphism of $\mathtt{s} = \mathtt{T_L\ bool} \to \mathtt{T_H\ bool}$ and $\mathtt{bool} \to \mathtt{bool}$:

$$\begin{array}{lcl}
\mathtt{f_1} & : & (\mathtt{bool} \to \mathtt{bool}) \to \mathtt{s} \\
 & = & \lambda x_1 : \mathtt{bool} \to \mathtt{bool}.\ \lambda x_2 : \mathtt{T_L\ bool}. \\
 & & \mathtt{bind}\ x_3 = x_2\ \mathtt{in}\ \eta_\mathtt{H}\ (x_1\ x_3) \\
\mathtt{g_1} & : & \mathtt{s} \to (\mathtt{bool} \to \mathtt{bool}) \\
 & = & \lambda x_4 : \mathtt{s}.\ \lambda x_5 : \mathtt{bool}.\ \mathtt{bind}\ x_6 = x_4\ (\eta_\mathtt{L}\ x_5)\ \mathtt{in}\ x_6
\end{array}$$

But $\mathtt{g_1}$ does not type-check. We cannot show $\mathtt{bool} \succeq \mathtt{H}$ in type-checking $x_4 : \mathtt{s}, x_5 : \mathtt{bool} \vdash \mathtt{bind}\ x_6 = x_4\ (\eta_\mathtt{L}\ x_5)\ \mathtt{in}\ x_6 : \mathtt{bool}$ with DT-Bind.

Instead we will show that $\mathtt{t}$ is isomorphic to $(\mathtt{bool} \to \mathtt{bool})^\dagger = \mathtt{bool} \to \mathtt{bool}$ in F where

$$\mathtt{t} = \forall \alpha_\mathtt{H}.\ \forall \alpha_\mathtt{L}.\ (\alpha_\mathtt{H} \to \alpha_\mathtt{L}) \to (\alpha_\mathtt{L} \to \mathtt{bool}) \to (\alpha_\mathtt{H} \to \mathtt{bool})$$

First, we construct these mediating functions in F:

$$\begin{array}{lcl}
\mathtt{f_2} & : & (\mathtt{bool} \to \mathtt{bool}) \to \mathtt{t} \\
 & = & \lambda x_1 : \mathtt{bool} \to \mathtt{bool}.\ \Lambda \alpha_\mathtt{H}.\ \Lambda \alpha_\mathtt{L}.\ \lambda x_2 : \alpha_\mathtt{H} \to \alpha_\mathtt{L}. \\
 & & \lambda x_3 : \alpha_\mathtt{L} \to \mathtt{bool}.\ \lambda x_4 : \alpha_\mathtt{H}.\ x_1\ (x_3\ (x_2\ x_4)) \\
\mathtt{g_2} & : & \mathtt{t} \to (\mathtt{bool} \to \mathtt{bool}) \\
 & = & \lambda x_5 : \mathtt{t}.\ \lambda x_6 : \mathtt{bool}.\ x_5\ [\mathtt{1}]\ [\mathtt{1}] \\
 & & (\lambda x_7 : \mathtt{1}.\ x_7)\ (\lambda x_8 : \mathtt{1}.\ x_6)\ \mathtt{<>}
\end{array}$$

Now we need to prove that $\mathtt{g_2} \circ \mathtt{f_2}$ and $\mathtt{f_2} \circ \mathtt{g_2}$ are identity functions. The first identity $\mathtt{g_2} \circ \mathtt{f_2}$ is straightforward from $\beta\eta$-reductions and we use $=$ to denote $\beta\eta$-equivalence. For all $\vdash \mathtt{m_0} : \mathtt{bool} \to \mathtt{bool}$,

$$\begin{array}{cl}
 & (\mathtt{g_2} \circ \mathtt{f_2})\ \mathtt{m_0} \\
= & \lambda x_6 : \mathtt{bool}.\ (\lambda x_2 : \mathtt{1} \to \mathtt{1}.\ \lambda x_3 : \mathtt{1} \to \mathtt{bool}.\ \lambda x_4 : \mathtt{1}. \\
 & \mathtt{m_0}\ (x_3\ (x_2\ x_4)))\ (\lambda x_7 : \mathtt{1}.\ x_7)\ (\lambda x_8 : \mathtt{1}.\ x_6)\ \mathtt{<>} \\
= & \lambda x_6 : \mathtt{bool}.\ (\lambda x_4 : \mathtt{1}.\ \mathtt{m_0}\ x_6)\ \mathtt{<>} \\
= & \mathtt{m_0}
\end{array}$$

The second identity $\mathtt{f_2} \circ \mathtt{g_2}$ requires the parametricity theorem: for all $\vdash \mathtt{m_1} : \forall \alpha_\mathtt{H}.\ \forall \alpha_\mathtt{L}.\ (\alpha_\mathtt{H} \to \alpha_\mathtt{L}) \to (\alpha_\mathtt{H} \to \mathtt{bool}) \to (\alpha_\mathtt{L} \to \mathtt{bool})$:

$$\begin{array}{cl}
 & (\mathtt{f_2} \circ \mathtt{g_2})\ \mathtt{m_1} \\
= & \Lambda \alpha_\mathtt{H}.\ \Lambda \alpha_\mathtt{L}.\ \lambda x_2 : \alpha_\mathtt{H} \to \alpha_\mathtt{L}.\ \lambda x_3 : \alpha_\mathtt{L} \to \mathtt{bool}.\ \lambda x_4 : \alpha_\mathtt{H}. \\
 & (\lambda x_6 : \mathtt{bool}.\ \mathtt{m_1}\ [\mathtt{1}]\ [\mathtt{1}]\ (\lambda x_7 : \mathtt{1}.\ x_7) \\
 & (\lambda x_8 : \mathtt{1}.\ x_6)\ \mathtt{<>})\ (x_3\ (x_2\ x_4)) \\
= & \Lambda \alpha_\mathtt{H}.\ \Lambda \alpha_\mathtt{L}.\ \lambda x_2 : \alpha_\mathtt{H} \to \alpha_\mathtt{L}.\ \lambda x_3 : \alpha_\mathtt{L} \to \mathtt{bool}.\ \lambda x_4 : \alpha_\mathtt{H}. \\
 & \mathtt{m_1}\ [\mathtt{1}]\ [\mathtt{1}]\ (\lambda x_7 : \mathtt{1}.\ x_7)\ (\lambda x_8 : \mathtt{1}.\ (x_3\ (x_2\ x_4)))\ \mathtt{<>}
\end{array}$$

We appear to be stuck at this point: there are no more $\beta\eta$-reductions to apply because $\mathtt{m_1}$ is abstract. However, $\mathtt{m_1}$ has a polymorphic type $\mathtt{t}$ and, by the parametricity theorem, $\mathtt{m_1} \approx \mathtt{m_1} : \forall \alpha_\mathtt{H}.\ \forall \alpha_\mathtt{L}.\ (\alpha_\mathtt{H} \to \alpha_\mathtt{L}) \to (\alpha_\mathtt{H} \to \mathtt{bool}) \to (\alpha_\mathtt{L} \to \mathtt{bool}) \mid \rho$. Expanding the definitions of FR-All and FR-Fun (together with FR-Term) for a few times, we obtain

$$\mathtt{m_1}\ [\mathtt{t_1}]\ [\mathtt{t_2}]\ \mathtt{m_2}\ \mathtt{m_3}\ \mathtt{m_4} \approx \mathtt{m_1}\ [\mathtt{t_1'}]\ [\mathtt{t_2'}]\ \mathtt{m_2'}\ \mathtt{m_3'}\ \mathtt{m_4'} : \mathtt{bool} \mid \rho'$$

as long as the following conditions are satisfied:

$$\begin{array}{rcl}
\rho' & = & \rho, \alpha_\mathtt{H} \mapsto \mathtt{R_1}, \alpha_\mathtt{L} \mapsto \mathtt{R_2} \\
\mathtt{R_1} & \in & \mathtt{t_1} \leftrightarrow \mathtt{t_1'} \\
\mathtt{R_2} & \in & \mathtt{t_1} \leftrightarrow \mathtt{t_1'} \\
\mathtt{m_2} \approx \mathtt{m_2'} & : & \alpha_\mathtt{H} \to \alpha_\mathtt{L} \mid \rho' \\
\mathtt{m_3} \approx \mathtt{m_3'} & : & \alpha_\mathtt{L} \to \mathtt{bool} \mid \rho' \\
\mathtt{m_4} \approx \mathtt{m_4'} & : & \alpha_\mathtt{H} \mid \rho'
\end{array}$$

We pick the types, relations and terms as follows:

$$\begin{array}{lcl@{\qquad}lcl}
\mathtt{t_1} & = & \mathtt{1} & \mathtt{t_1'} & = & \alpha_\mathtt{H} \\
\mathtt{t_2} & = & \mathtt{1} & \mathtt{t_2'} & = & \alpha_\mathtt{L} \\
\mathtt{R_1} & = & \lambda x_1' : \mathtt{1}.\ x_4 & & & \\
\mathtt{R_2} & = & \lambda x_2' : \mathtt{1}.\ x_2\ x_4 & & & \\
\mathtt{m_2} & = & \lambda x_7 : \mathtt{1}.\ x_7 & \mathtt{m_2'} & = & x_2 \\
\mathtt{m_3} & = & \lambda x_8 : \mathtt{1}.\ x_3\ (x_2\ x_4) & \mathtt{m_3'} & = & x_3 \\
\mathtt{m_4} & = & \mathtt{<>} & \mathtt{m_4'} & = & x_4
\end{array}$$

But we need to check that these definitions satisfy those conditions above. First, since $\mathtt{R_1}$ is a function of type $\mathtt{1} \to \alpha_\mathtt{H}$, we have $\mathtt{R_1} \in \mathtt{1} \leftrightarrow \alpha_\mathtt{H}$ (a binary relation of $\mathtt{1}$ and $\alpha_\mathtt{H}$). The same goes for $\mathtt{R_2}$ of type $\mathtt{1} \to \alpha_\mathtt{L}$.

Then, we check that $\mathtt{m_2} \approx \mathtt{m_2'} : \alpha_\mathtt{H} \to \alpha_\mathtt{L} \mid \rho'$. By FR-Fun, it is equivalent to check that $\mathtt{m_2}\ \mathtt{m_5} \approx \mathtt{m_2'}\ \mathtt{m_5'} : \alpha_\mathtt{L} \mid \rho'$ for all $\mathtt{m_5} \approx \mathtt{m_5'} : \alpha_\mathtt{H}$. By FR-Var, $\mathtt{m_5'} = \mathtt{R_1}\ \mathtt{m_5} = x_4$ and it is equivalent to check that $\mathtt{R_2}\ (\mathtt{m_2}\ \mathtt{m_5}) = \mathtt{m_2'}\ \mathtt{m_5'}$. Indeed, $\mathtt{R_2}\ (\mathtt{m_2}\ \mathtt{m_5}) = (\lambda x_1' : \mathtt{1}.\ x_2\ x_4)\ ((\lambda x_7 : \mathtt{1}.\ x_7)\ \mathtt{m_5}) = x_2\ x_4$ and $\mathtt{m_2'}\ \mathtt{m_5'} = x_2\ (\mathtt{R_1}\ \mathtt{m_5}) = x_2\ x_4$.

Similarly, we check that $\mathtt{m_3} \approx \mathtt{m_3'} : \alpha_\mathtt{L} \to \mathtt{bool} \mid \rho'$. For all $\mathtt{m_6} \approx \mathtt{m_6'} : \alpha_\mathtt{L} \mid \rho'$ such that $\mathtt{m_6'} = \mathtt{R_2}\ \mathtt{m_6} = x_2\ x_4$, we have $\mathtt{m_3}\ \mathtt{m_6} = (\lambda x_8 : \mathtt{1}.\ x_3\ (x_2\ x_4))\ \mathtt{m_6} = x_3\ (x_2\ x_4)$ and $\mathtt{m_3'}\ \mathtt{m_6'} = x_3\ (x_2\ x_4)$. At last, we check that $\mathtt{m_4} \approx \mathtt{m_4'} : \alpha_\mathtt{H} \mid \rho'$ which follows from $\mathtt{R_1}\ \mathtt{m_4} = x_4$ and $\mathtt{m_4'} = x_4$.

Since the relation $\approx$ of related values at bool implies β-equivalence, the parametricity theorem implies that

$$\begin{aligned} & \mathtt{m_1\ [1]\ [1]\ (\lambda x_7{:}1.\ x_7)\ (\lambda x_8{:}1.\ (x_3\ (x_2\ x_4)))\ <>} \\ = \ & \mathtt{m_1\ [\alpha_H]\ [\alpha_L]\ x_2\ x_3\ x_4} \end{aligned}$$

which we can use to finish the proof:

$$\begin{aligned} & \mathtt{(f_2 \circ g_2)\ m_1} \\ = \ & \cdots \\ = \ & \Lambda\alpha_H.\ \Lambda\alpha_L.\ \lambda x_2{:}\alpha_H \to \alpha_L.\ \lambda x_3{:}\alpha_L \to \mathtt{bool}.\ \lambda x_4{:}\alpha_H. \\ & \mathtt{m_1\ [\alpha_H]\ [\alpha_L]\ x_2\ x_3\ x_4} \\ = \ & \mathtt{m_1} \end{aligned}$$

5.2 Protection contexts

The excursion to F and parametricity shows that our intuition is right: $T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}$ is isomorphic to $\mathtt{bool} \to \mathtt{bool}$ (from the high observer's view) in a richer system like F, even though their mediating function does not type-check in DCC. Using insights from the translation, we now show how to extend DCC's type system to allow such an isomorphism.

From the translation, we observe that the *protection context* is explicit in the typing rule of functions in F but missing in the typing rule of bind in DCC. In F, protections $\eta_\ell\ e$ are translated into functions and the type system uses the typing context Γ to keep track of the variable assumptions and thus the protection assumptions. In DCC, the type system does not make such protection assumptions of a term. For example, $\eta_H\ (\mathtt{bind}\ x = \eta_H\ \mathtt{true}\ \mathtt{in}\ x)$ should type-check, because the result of the bind is protected by η_H in the outside context, but this term is not well-typed in DCC.

Therefore, we propose to extend the typing judgment of DCC to be $\Gamma;\pi \vdash e : s$ where $\pi \in \mathcal{L}_\ell$ is the protection context. This is similar to the *program counter labels* that provide contextual information found in security-typed languages [5, 24, 18]. The function types $s_1 \to s_2$ now becomes $[\pi]\ s_1 \to s_2$ to account for the protection context. The new typing rules are:

$$\frac{\Gamma, x{:}s_1;\pi_2 \vdash e : s_2}{\Gamma;\pi_1 \vdash \lambda x{:}s_1.\ e : [\pi_2]\ s_1 \to s_2}$$

$$\frac{\Gamma;\pi_1 \vdash e_1 : [\pi_2]\ s_1 \to s_2 \quad \Gamma;\pi_1 \vdash e_2 : s_1 \quad \pi_2 \sqsubseteq \pi_1}{\Gamma;\pi_1 \vdash e_1\ e_2 : s_2}$$

$$\frac{\Gamma;\pi \sqcup \ell \vdash e : s}{\Gamma;\pi \vdash \eta_\ell\ e : T_\ell\ s}$$

$$\frac{\Gamma;\pi \vdash e_1 : T_\ell\ s_1 \quad \Gamma, x{:}s_1;\pi \vdash e_2 : s_2 \quad \pi \vdash s_2 \succeq \ell}{\Gamma;\pi \vdash \mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2 : s_2}$$

The other typing rules are essentially the same but now pass around the protection context. We add the following new protection rules that make use of the protection context, or call the old rules otherwise:

$$\frac{\ell \sqsubseteq \pi}{\pi \vdash s \succeq \ell} \qquad \frac{s \succeq \ell \quad \ell \not\sqsubseteq \pi}{\pi \vdash s \succeq \ell}$$

It appears to be straightforward to extend the translation for this augmented version of DCC by using the following type translation for functions: $([\pi]\ s_1 \to s_2)^\dagger = (\alpha_\pi \times s_1^\dagger) \to s_2^\dagger$. Protection contexts are also useful for other types whose values may contain delayed computations. For example, in call-by-name calculi like DCC, the protection context of a product type $([\pi]\ t_1 \times t_2)$ expresses the constraint that a projection of the product can be made only in a context higher or equal to π. The typing and translation rules for protection contexts of other types are similar to those for function types; we do not investigate further here.

5.3 Isomorphism

Given the modified type system for DCC, we now go back to show the isomorphism of $s = T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}$ and $\mathtt{bool} \to \mathtt{bool}$ by proving that $g_1 \circ f_1$ and $f_1 \circ g_1$ are identity functions from the high observer's view ($\zeta = H$). The technique is similar to proving the identities of f_2 and g_2 using the parametricity theorem in Section 5.1, but instead uses the noninterference theorem.

First of all, the mediating functions now type-check and have the following types. With the additional constraint [H] on the protection context of g_1, the term $\mathtt{bind}\ x_6 = x_4\ (\eta_L\ x_5)\ \mathtt{in}\ x_6$ satisfies the new rule above because $H \vdash \mathtt{bool} \succeq H$.

$$\begin{aligned} f_1 &: [L]\ ([L]\ \mathtt{bool} \to \mathtt{bool}) \to ([L]\ T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}) \\ g_1 &: [H]\ ([L]\ T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}) \to ([L]\ \mathtt{bool} \to \mathtt{bool}) \end{aligned}$$

As before, we use $\beta\eta$-reductions in DCC to prove the identities, using these equivalences:

$$\begin{aligned} \mathtt{bind}\ x = \eta_\ell\ e\ \mathtt{in}\ x \quad &=_\beta \quad e \\ \eta_\ell\ (\mathtt{bind}\ x = e\ \mathtt{in}\ x) \quad &=_\eta \quad e \end{aligned}$$

The first identity $g_1 \circ f_1$ is straightforward: for all $\vdash e_0 : [L]\ \mathtt{bool} \to \mathtt{bool}$, we have $(g_1 \circ f_1)\ e_0 = \lambda x_5{:}\mathtt{bool}.\ \mathtt{bind}\ x_6 = \eta_H\ (e_0\ x_5)\ \mathtt{in}\ x_6 = e_0$.

The second identity $f_1 \circ g_1$ requires the noninterference theorem: for all $\vdash e_1 : [L]\ T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}$, we have $(f_1 \circ g_1)\ e_1 = \lambda x_2{:}T_L\ \mathtt{bool}.\ \mathtt{bind}\ x_3 = x_2\ \mathtt{in}\ e_1\ (\eta_L\ x_3)$. There are no more $\beta\eta$-reductions to apply because e_1 is abstract. But, using the noninterference theorem, we will prove the following to order to finish the proof:

$$\begin{aligned} & \lambda x_2{:}T_L\ \mathtt{bool}.\ \mathtt{bind}\ x_3 = x_2\ \mathtt{in}\ e_1\ (\eta_L\ x_3) \\ = \ & \lambda x_2{:}T_L\ \mathtt{bool}.\ e_1\ x_2 \end{aligned}$$

Consider the applications of these two functions with two related terms. For all $m_2 \approx_H m_2' : T_L\ \mathtt{bool}$ such that $m_2 \longrightarrow^* \eta_L\ m_3$ and $m_2' \longrightarrow^* \eta_L\ m_3'$ and $m_3 \approx_H m_3' : \mathtt{bool}$, we have $(\lambda x_2 : T_L\ \mathtt{bool}.\ \mathtt{bind}\ x_3 = x_2\ \mathtt{in}\ e_1\ (\eta_L\ x_3))\ m_2 \longrightarrow^* e_1\ (\eta_L\ m_3)$ and $(\lambda x_2 : T_L\ \mathtt{bool}.\ e_1\ x_2)\ m_3' \longrightarrow^* e_1\ (\eta_L\ m_3')$. By DR-Label2, $\eta_L\ m_3 \approx_H \eta_L\ m_3' : T_H\ \mathtt{bool}$.

By noninterference and the typing $\vdash e_1 : T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}$, we have $e_1 \approx_H e_1 : T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}$. Expanding the definition of DR-Fun, we have $e_1\ (\eta_L\ m_3) \approx_H e_1\ (\eta_L\ m_3') : T_L\ \mathtt{bool}$. Hence, $\lambda x_2{:}T_L\ \mathtt{bool}.\ \mathtt{bind}\ x_3 = x_2\ \mathtt{in}\ e_1\ (\eta_L\ x_3) \approx_H \lambda x_2 : T_L\ \mathtt{bool}.\ e_1\ x_2 : T_L\ \mathtt{bool} \to T_H\ \mathtt{bool}$, which is the $\beta\eta$-equivalence from the high observer's view.

6. Discussion

We have shown in this paper that dependency analyses embodied by DCC's type system can be translated into F's parametricity. This embedding uses only a small subset of F—for instance it does not require any nontrivial use of type abstraction or application. DCC therefore appears to be much weaker than F. One possible future direction is to investigate additional extensions that preserve DCC's spirit

```
data H                          ;     data L = Khl H          -- Lattice
eta :: s -> T l s               ;     eta e = T (\x -> e)     -- Monad injection
newtype T l s = T (l -> s)      ;     t (T x) = x             -- T constructor/destructor

class Bind l s2 where bind :: T l s1 -> (s1 -> s2) -> s2
instance Bind l () where                                      -- P-Unit, LP-Unit
  bind m1 m2 = ()
instance (Bind l s1, Bind l s2) => Bind l (s1, s2) where      -- P-Pair, LP-Pair
  bind m1 m2 = (bind m1 (fst . m2), bind m1 (snd . m2))
instance (Bind l s2) => Bind l (s1 -> s2) where               -- P-Fun, LP-Fun
  bind m1 m2 = \x0 -> bind m1 (\x -> m2 x x0)
instance (Bind H s) => Bind H (T L s) where                   -- P-Label1, LP-Label1 with H </= L
  bind m1 m2 = T (\x0 -> bind m1 (\x -> t (m2 x) x0))

instance Bind L (T H s) where                                 -- P-Label2, LP-Label2 with L <= H
  bind (T m1) m2 = T (\x0 -> t (m2 (m1 (Khl x0))) x0)
instance Bind l (T l s) where                                 -- P-Label2, LP-Label0, reflexive
  bind (T m1) m2 = T (\x0 -> t (m2 (m1 x0)) x0)
```

Figure 2: Prototype implementation in Haskell for the two-point lattice

while admitting more of the expressiveness of F. For instance, generating new labels dynamically may correspond to first-class type abstraction in F. This connection is already being explored by Fluet and Morrisett, who are investigating the relationship between region-based memory management, Haskell-style monadic effects, and polymorphism [?].

6.1 Haskell implementation

Another future direction is to use the ideas behind this translation to provide implementations of dependency analyses in polymorphic languages. Figure 2 shows a prototype implementation in Haskell[4] for DCC with the two-point lattice $\mathtt{L} \sqsubseteq \mathtt{H}$.

Abstract datatypes `H` and `L` along with the key coercion function `Khl :: H -> L` encode the lattice. The type of the function `eta :: s -> T l s` corresponds to the typing rule DT-Prot in Section 2:

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash \eta_\ell\ e : T_\ell\ s} \quad \text{(DT-Prot)}$$

The definitions of the type `newtype T l s = T (l -> s)` and the function `eta e = T (\x -> e)` correspond to the translation rule for protection under label ℓ in Section 3.1:

$$\begin{array}{rcl} (T_\ell\ s)^\dagger & = & \alpha_\ell \to s^\dagger \\ (\eta_\ell\ e)^* & = & \lambda x{:}\alpha_\ell.\ e^* \quad \text{(fresh } x\text{)} \end{array}$$

Finally the destructor `t (T x) = x` encodes the evaluation rule of `bind`:

$$\mathtt{bind}\ x = \eta_\ell\ e\ \mathtt{in}\ e_2 \longrightarrow e_2\{e/x\}$$

The last part of the implementation is to use the typeclass `Bind` to encapsulate the typing rule DT-Bind and its protection rules $s \succeq \ell$ for the `bind` operations in Section 2. Each instance of the `Bind` typeclass corresponds to a translation rule in Figure 1, except that we also have the instance for the implicit reflexive cases LP-Label0 when $\mathtt{H} \sqsubseteq \mathtt{H}$ and $\mathtt{L} \sqsubseteq \mathtt{L}$.

The example in Section 3.1 type-checks in Haskell:

```
e :: T L Bool -> T H Bool
e x1 = bind x1 (\x2 -> eta x2)
```

On the other hand, if `e` is constrained to have the type `e :: T H Bool -> T L Bool`, Haskell will correctly report a type error because of the illegal dependency of a low output on the high input. If we omit the type annotation, Haskell can even do *label inference*:

```
e :: forall s l1 l.
     (Bind l (T l1 s)) => T l s -> T l1 s
```

Unfortunately, this implementation requires $O(n^2)$ typeclass instances to encode the `bind` translation for a lattice containing n labels. We could have factored the lattice edges into two classes, class `LE` ($\sqsubseteq$, less than or equal to) and class `GT` ($\not\sqsubseteq$, greater than), such that the number of class instances is shifted to these two classes, and `Bind` would only have two instances (one for `LE` and one for `GT`). This construction still has the same complexity of class instances and it requires non-standard Haskell extensions for checking non-overlapping and complementary instances, but it may be a more modular approach.

It would also be interesting to see whether other clever programming tricks, such as *phantom types* [3], could reduce this overhead. Another alternative is to use explicit subtyping and bounded quantification, such as $F_{<:}$ [10], which would simplify the description of the label lattice considerably.

6.2 Fixpoints and termination

It is known that adding `fix` (and side-effects in general) to System F weakens the parametricity theorem because programs may diverge [21, 7, 6]. A similar phenomenon arises in noninterference, which may be defined to be termination sensitive [5, 1].

This paper focuses on the terminating fragment of DCC, which helps to emphasize the connection between noninterference and parametricity. Like the original DCC, we would need to add pointed types and a `fix` operator to F to account for termination. We expect that all of the results presented here carry through to the case where `fix` is added to both

[4]We use Glasgow Haskell Compiler (GHC) 6.2 with flags `-fglasgow-exts` and `-fallow-undecidable-instances`.

DCC and F, assuming that noninterference and parametricity are suitably weakened to account for the possibility of diverging computations [6]. However, the translation should be slightly altered to prevent "forged" keys from being used to access protected data.

The problem is that having `fix` in the language allows arbitrary keys or coercions to be created, because all types are inhabited. For example, the following term

$$\mathtt{fix}\ (\lambda \mathtt{x}{:}\alpha_{\mathtt{H}}.\ \mathtt{x}) : \alpha_{\mathtt{H}}$$

can be used to unlock any high-security data. Note that, however, this term is non-terminating. To prevent such keys from being used by a bad F context, we can simply force the evaluation of the key by using a construct similar to `seq` in Haskell. Essentially, we need to modify the translation rules for protection LP-Label1 and LP-Label2 in Figure 1 as follows:

$$\begin{array}{rll} & [\![\mathtt{T}_{\ell'}\ \mathtt{s} \succeq \ell]\!](\mathtt{x{:}t}, \mathtt{m}_1, \mathtt{m}_2) & \text{where } \ell \not\sqsubseteq \ell' \\ = & \lambda \mathtt{x}_0{:}\alpha_{\ell'}.\ \mathtt{seq}\ \mathtt{x}_0\ ([\![\mathtt{s} \succeq \ell]\!](\mathtt{x{:}t}, \mathtt{m}_1, \mathtt{m}_2\ \mathtt{x}_0)) & \\ & & \\ & [\![\mathtt{T}_{\ell'}\ \mathtt{s} \succeq \ell]\!](\mathtt{x{:}t}, \mathtt{m}_1, \mathtt{m}_2) & \text{where } \ell \sqsubseteq \ell' \\ = & \lambda \mathtt{x}_0{:}\alpha_{\ell'}.\ \mathtt{seq}\ \mathtt{x}_0\ ((\lambda \mathtt{x{:}t}.\ \mathtt{m}_2\ \mathtt{x}_0)\ (\mathtt{m}_1\ (\mathtt{k}_{\ell'\ell}\ \mathtt{x}_0))) & \end{array}$$

Further investigating the interactions between side effects and relational parametricity is an important area of ongoing research.

6.3 Related work

The ideas behind our encoding of DCC into F are similar to Sumii and Pierce's work on the cryptographic lambda calculus [16]. They use logical relations (and, more recently, bisimulations [17]) to capture the effects of encryption as an information-hiding mechanism. Knowledge of a cryptographic key is sufficient to reveal the encrypted data. In this paper, our translation uses function closures with types of the form $\alpha_\ell \to \mathtt{t}$ to represent hidden data of type `t`. Possession of a key of type α_ℓ allows the computation to obtain the data hidden in the closure. Unlike Sumii and Pierce's work, our translation must also account for DCC's lattice order using key coercions.

Sabelfeld and Sands used PER models to prove noninterference properties in the context of information-flow security [15]. Benton uses logical relations to prove program transformations related to information-flow and dependency correct [2]. Both of these approaches, as we do here, use complete relations to represent abstract views of data.

Other researchers have studied languages via translation into F. For example, Washburn and Weirich [22] use parametric polymorphism to encode higher-order abstract syntax. There, parametricity is also essential to show that the encoding is adequate.

7. Conclusion

This paper has shown how to encode the dependency core calculus, which is useful for modeling many kinds of program analyses, into System F. This encoding provides both novel insights into the relationship between dependency and parametricity and an alternate proof of DCC's noninterference theorem. We also give a prototype implementation of this translation in Haskell.

Acknowledgments

The authors thank Eijiro Sumii, Stephanie Weirich, and the anonymous reviewers for their comments on drafts of this paper. Stephanie also provided the Haskell prototype.

This research was supported in part by National Science Foundation grants CNS-0346939 (*CAREER: Language-based Distributed System Security*) and CCR-0311204 (*Dynamic Security Policies*).

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, Jan. 1999.

[2] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. 31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 14–25. ACM Press, 2004.

[3] M. Fluet and G. Morrisett. Monadic regions. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2004.

[4] M. Fluet and R. Pucella. Phantom types and subtyping. In *Proc. of the 2nd IFIP International Conference on Theoretical Computer Science*, 2002.

[5] J.-Y. Girard. *Interprétation Functionelle et Élimination des Coupures dans l'Arithmétique d'Order Supérieure.* PhD thesis, Université Paris VII, 1972.

[6] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, 1998.

[7] P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Proc. 31st ACM Symp. on Principles of Programming Languages (POPL)*, 2004.

[8] J. Launchbury and R. Paterson. Parametricity and unboxing with pointed types. In *Proc. of the 9th European Symposium on Programming*, 1996.

[9] J. C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

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

[11] B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[12] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[13] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science.* Springer-Verlag, 1974.

[14] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing.* Elsevier Science Publishers B.V., 1983.

[15] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[16] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, Mar. 2001.

[17] E. Sumii and B. C. Pierce. Logical relations for encryption. In *Proc. of the 14th IEEE Computer*

Security Foundations Workshop, 2001.

[18] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *Proc. 31st ACM Symp. on Principles of Programming Languages (POPL)*. ACM Press, 2004.

[19] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *Proc. IEEE Symposium on Security and Privacy*, 2004. To Appear.

[20] S. Tse and S. Zdancewic. Translating dependency into parametricity. Technical report, University of Pennsylvania, Jan. 2004.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[22] P. Wadler. Theorems for free! In *Proceedings of the 4th International Symposium on Functional Programming and Computer Architecture*, Sept. 1989.

[23] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proc. of the 8th ACM SIGPLAN International Conference on Functional Programming*, Upsala, Sweden, Aug. 2003.

[24] S. Zdancewic. *Programming Languages for Information Security.* PhD thesis, Cornell University, 2002.

[25] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *Proc. of the 10th European Symposium on Programming*, 2001.