Simple Noninterference from Parametricity

Simple Noninterference from Parametricity

MAXIMILIAN ALGEHED, Chalmers University of Technology, Sweden
JEAN-PHILIPPE BERNARDY, University of Gothenburg, Sweden

In this paper we revisit the connection between parametricity and noninterference. Our primary contribution is a proof of noninterference for a polyvariant variation of the Dependency Core Calculus of Abadi et al. in the Calculus of Constructions. The proof is modular: it leverages parametricity for the Calculus of Constructionsand the encoding of data abstraction using existential types.This perspective gives rise to simple and understandable proofs of noninterference from parametricity. All our contributions have been mechanised in the Agda proof assistant.

CCS Concepts: • **Security and privacy** → **Formal methods and theory of security**; **Logic and verification**; • **Theory of computation** → *Type theory*; • **Software and its engineering** → *Functional languages*; *Syntax*; *Semantics*.

Additional Key Words and Phrases: Security, Types, Parametricity, Noninterference

ACM Reference Format:
Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple Noninterference from Parametricity. *Proc. ACM Program. Lang.* 3, ICFP, Article 89 (August 2019), 22 pages. https://doi.org/10.1145/3341693

1 INTRODUCTION

Parametricity is a generic property of programming languages with polymorphism. It produces useful theorems about programs from nothing but their types. Because such results do not depend on the content of the program, but follow mechanically from their types, they have been dubbed "free theorems" by Wadler [1989]. For example, he shows how parametricity gives us the following theorem for any polymorphic list-transformation function `r`.

$$\text{if } \mathtt{r} : \forall a.\ [a] \to [a],\ \text{then } \forall f\ xs.\ \mathtt{map}\ f\ (\mathtt{r}\ xs) = \mathtt{r}\ (\mathtt{map}\ f\ xs)$$

This theorem tells us something useful and instructive about the way `r` interacts with its input: it works on the structure of the input list in a way which is independent of the elements of the list. Another way to look at this statement is that the elements of the list appear *secret* to `r`: it is not able to inspect the elements. Many of the theorems we get from parametricity are of this form; they tell us that parts of the input are opaque to our functions.

To further illustrate this point, we can draw a parallel with the notion of an *erasure* function from the information-flow control literature [Russo et al. 2008; Stefan et al. 2011; Vassena et al. 2018], whose role is to hide secrets from programs running under "unprivileged" levels. In the meta-theory of secure-by-construction programming languages, one will typically show that secure programs commute with the erasure function: applying erasure before running the program yields the same result as applying it after running the program. This commutation property is reminiscent of the

Authors' addresses: Maximilian Algehed, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, algehed@chalmers.se; Jean-Philippe Bernardy, Department of Philosophy, Linguistics, and Theory of Science, University of Gothenburg, Göteborg, Sweden, jean-philippe.bernardy@gu.se.

2475-1421/2019/8-ART89
https://doi.org/10.1145/3341693

way f commutes with r in the above example: if we instantiate the function f with an erasure function, namely $\lambda a.\ <>$, where <> is the only element of the unit type, and we instantiate the list xs with any list of secret values, we see that the `r` function commutes with erasure, and `r` behaves just as a secure program. In general, this sort of commutation property is used to prove *noninterference*, the property that public outputs of a program may not depend on secret inputs.

This example suggests a general connection between parametricity and noninterference. They talk about similar things, and in fact this connection has been observed several times in the literature [Abadi et al. 1999; Bowman and Ahmed 2015; Tse and Zdancewic 2004]. For example, Bowman and Ahmed [2015] develop a full abstraction result between the Dependency Core Calculus (DCC) [Abadi et al. 1999] and System F_ω [Barendregt et al. 2013] and use parametricity for one key part of their proof and get noninterference as a result.

In this paper we provide a fresh look at the connection between parametricity and noninterference. That is, instead of constructing a special-purpose free-theorem with custom logical relations (like in our illustration about lists, or in the works cited above), we mechanically apply the most generic parametricity result and specialize it to noninterference separately. Technically, we proceed as follows:

(1) We perform a shallow embedding of DCC into a lambda-calculus with dependent types (say the Calculus of Constructions — CC).
(2) We apply the generic parametricity result for CC to this encoding, obtaining a "free theorem" for *every* program written in the encoding.
(3) We show that noninterference is a consequence of such free theorems. Concretely, we pick the parametric interpretation of security levels, types and combinators to make them coincide exactly with the notion of independence found in the information flow control literature.
(4) Finally (and some may regard this conceptually simple step optional), we connect the shallow embedding to a deep embedding and show that noninterference carries over to this representation.

Even though step (3) is not a trivial consequence of step (2) (one must still correctly instantiate the relations of secret-preserving types), the complete proof becomes considerably simpler.

Such a simplification has several benefits:

(1) We can easily go beyond the state of the art and support the polyvariant variation of DCC, where functions are not constrained to act on pre-defined security levels.
(2) The pattern can be applied to several other calculi — only the step (3) is non-mechanical. One could even say that if a language is sufficiently expressive to embed a security library, then the noninterference properties of such libraries can be proven as a consequence of the parametricity theorem for said language.
(3) The technique is more amenable to formal verification. In fact all our proofs have been mechanised in the Agda proof assistant [Norell 2007] and are available online [1].

We consider two calculi in this paper, the Sec language of Russo et al. [2008], and the DCC language of Abadi et al. [1999]. Both calculi are given shallow embeddings in the Calculus of Constructions (CC) and we develop noninterference proofs for both of them using parametricity. Because DCC is strictly more expressive than Sec, we build our embedding of DCC as an extension of the Sec embedding. To connect the noninterference theorem for DCC to the actual DCC calculus, we show how to translate DCC terms into their corresponding embedding in CC.

The contributions of this paper are the following:

[1] `https://github.com/MaximilianAlgehed/SimpleNoninterferenceFromParametricity`

(1) We develop label-polymorphic shallow embeddings of the Sec calculus of Russo et al. [2008] and the Dependency Core Calculus (DCC) of Abadi et al. [1999] in the Calculus of Constructions (CC), Sections 4 and 5.
(2) We show that parametricity for CC implies noninterference for our shallow embeddings, Sections 4 and 5. To the best of our knowledge, this constitutes the first parametricity-based proof of the soundness of a security library, of which there are many [Algehed and Russo 2017; Buiras et al. 2015; Russo 2015; Russo et al. 2008; Stefan et al. 2011]. Unlike previous work, however, our library is embedded in CC, not Haskell.
(3) We show that our shallow embedding of DCC in CC respects the operational semantics of DCC in its usual presentation. As a corollary of this we also obtain noninterference for this presentation of DCC, Section 6.
(4) We also implement our development as a shallow embedding of a small security library in Haskell, Section 7.

In this paper we use a notation similar to that of the Agda programming language [Norell 2007], with a few exceptions. The symbol $\star$ represents the universe of types, denoted `Set` in Agda. The notation $(a : A) \to B$ represents the dependent product type $\Pi(a : A)\ B$. We use a shorthand for multiple arguments of the same type as a comma-separated list of variables, $(a_0, a_1 : A) \to B$ is shorthand for $(a_0 : A) \to (a_1 : A) \to B$. As an example, in this notation the type of the identity function $\forall A.\ A \to A$ will be written $(A : \star) \to A \to A$.

2 PARAMETRICITY

Bernardy et al. [2012] have developed a notion of parametricity for Pure Type Systems (PTS) [Barendregt et al. 2013] based on a term-level translation function $[\![\_]\!] : \text{PTS} \to \text{PTS}$ which translates a PTS term into another PTS term, possibly in a more powerful PTS. (CC can be mapped to itself.) The key idea is to transform types into relations; and their inhabitants into proofs that they are related.

$$
\begin{aligned}
[\![\star]\!] &= \lambda A_0, A_1 : \star.\ A_0 \to A_1 \to \star \\
[\![x]\!] &= R_x \\
[\![(x : A) \to B]\!] &= \lambda f_0 : (x : A_0) \to B_0.\ \lambda f_1 : (x : A_1) \to B_1. \\
&\qquad (x_0 : A_0) \to (x_1 : A_1) \to (R_x : [\![A]\!]\ x_0\ x_1) \to [\![B]\!]\ (f_0\ x_0)\ (f_1\ x_1) \\
[\![f\ a]\!] &= [\![f]\!]\ a_0\ a_1\ [\![a]\!] \\
[\![\lambda x : A.\ e]\!] &= \lambda x_0 : A_0.\ \lambda x_1 : A_1.\ \lambda R_x : [\![A]\!]\ x_0\ x_1.\ [\![e]\!]
\end{aligned}
$$

Note that the translation of variables $[\![x]\!] = R_x$ assumes a variable R_x in scope which encodes information about the variable x, like the associated relation when x denotes a bound type variable. The variable R_x is in turn introduced by the translation of the binding constructs $(x : A) \to B$ and $\lambda x : A.\ e$, ensuring that the R_x associated with each bound variable x in an expression is also bound in the translation of that expression. The situation is similar for the variables related by R_x, namely x_1 and x_2. Furthermore, we extend the subscript notation for expressions: in the above, A_0 denotes the term A where every variable x is renamed to x_0 (likewise for A_1). The key result for this translation is the parametricity lemma (also known as the fundamental lemma of logical relations).

Lemma 1 (Parametricity). $\vdash f : t \Rightarrow \vdash [\![f]\!] : [\![t]\!]\ f\ f$.

The core statement of this lemma is that $[\![\_]\!]$ computes both propositions and proofs from PTS terms at the same time, and that any term f of an inhabited type t is related to itself.

Example: The `const` *function.* As an example of how to work with parametricity in this setting we briefly work through a proof that all functions $f : (A, B : \star) \to A \to B \to A$ ignore their last argument. Note that this statement already has somewhat of a security flavour to it, from the point of view of f the B argument is "secret" and therefore f may not make use of it. The statement we are trying to prove is the following:

$$\forall A, B : \star.\ \forall a : A.\ \forall b_0, b_1 : B. f\ A\ B\ a\ b_0 \equiv f\ A\ B\ a\ b_1$$

Which when cast in the setting of dependent types with types as propositions and programs as proofs means we need to give an inhabitant of the following type:

$$(A, B : \star) \to (a : A) \to (b_0, b_1 : B) \to f\ A\ B\ a\ b_0 \equiv f\ A\ B\ a\ b_1$$

To understand how the proof works, recall the parametricity lemma from above and what it says about f:

$$[\![f]\!] : [\![(A, B : \star) \to A \to B \to A]\!]\ f\ f$$

Expanding the brackets gives us:

$$\begin{aligned}
[\![f]\!] : &(A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to \\
&(B_0, B_1 : \star) \to (R_B : B_0 \to B_1 \to \star) \to \\
&(a_0 : A_0) \to (a_1 : A_1) \to (R_a : R_A\ a_0\ a_1) \to \\
&(b_0 : B_0) \to (b_1 : B_1) \to (R_b : R_B\ b_0\ b_1) \to \\
&R_A\ (f\ A_0\ B_0\ a_0\ b_0)\ (f\ A_1\ B_1\ a_1\ b_1)
\end{aligned}$$

As can be seen from this simple example, the expansion of types quickly becomes quite large. To ease reading of the expanded $[\![\_]\!]$ brackets we arrange the expansion of each function argument in the original type of f on its own line, with the result on the final line. When translating a specific function like f the lambdas in the translation of the dependent function space go away as the translation is immediately applied to f. This means that, like in the example above, the translation of a type argument is two types and a relation on the types, $[\![(A : \star) \to \ldots]\!]$ becomes $(A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to [\![\ldots]\!]$. Note that while $A_0 \to A_1 \to \star$ denotes a relation on types A_0 and A_1, it is also a proof that A_0 and A_1 are related at $[\![\star]\!]$. Similarly, the translation of what one might call "term" arguments like the last two arguments to f turn into two such arguments and a proof that they are related, $[\![A \to \ldots]\!]$ becomes $(a_0 : A_0) \to (a_1 : A_1) \to (R_a : R_A\ a_0\ a_1) \to [\![\ldots]\!]$. In general, the mantra to follow when expanding expressions like $[\![a \to \_]\!]$ is "two a's and a proof that they are related".

We are going to use $[\![f]\!]$ as the core of the proof; instantiating the argument relations R_A and R_B correctly is going to give us precisely the statement we are looking for. Because we are only dealing with two types A and B; we choose $A_0 = A_1 = A$ and likewise for B. Furthermore, because we know that the resulting type of `proof` is going to be equality at A for the applications of f it makes sense that R_A be precisely equality at A, $R_A\ a_0\ a_1 = a_0 \equiv a_1$. Next we turn to R_B, we want to apply f to two different and completely unrelated Bs, b_0 and b_1. This means that using, for example, equality at B for R_B cannot work, as we would then need to produce a proof that $b_0 \equiv b_1$ for arbitrary b_0 and b_1. The natural choice for R_B is then $R_B\ b_0\ b_1 = \top$, where $\top$ denotes the unit type with one element $<> : \top$. This definition of R_B corresponds to the full relation $B \times B$. With our intuition formed and relations defined we can give the full proof:

$$\begin{aligned}
&\texttt{proof} : (A, B : \star) \to (a : A) \to (b_0, b_1 : B) \to f\ A\ B\ a\ b_0 \equiv f\ A\ B\ a\ b_1 \\
&\texttt{proof}\ A\ B\ a\ b_0\ b_1 = [\![f]\!]\ A\ A\ (\equiv)\ B\ B\ (\lambda b_0 : B. \lambda b_1 : B.\ \top)\ a\ a\ (\texttt{refl}\ A\ a)\ b_0\ b_1\ <>
\end{aligned}$$

Here $\texttt{refl} : (A : \star) \to (a : A) \to a \equiv a$ is the standard witness of reflexivity for propositional equality in type theory. The proof is now complete, and we barely had to do any work other than analysing what our intuitions for the relations between the various mentioned variables are. This intuitive approach to the proofs is the bread-and-butter of proofs by parametricity and precisely what makes the technique appealing. It is the foundation of the next sections of this paper.

Sigma types. While the above definition is for a PTS with no constants, Bernardy et al. show how to give a parametricity translation for $\Sigma : (A : \star) \to (A \to \star) \to \star$ types, also known as dependent pairs where $(a, b) : \Sigma\ A\ B$ if and only if $a : A$ and $b : B\ a$. The type of $[\![\Sigma]\!]$ is given below:

$$\begin{aligned}[\![\Sigma]\!] : {} & (A_1, A_2 : \star) \to (R_A : A_1 \to A_2 \to \star) \to \\ & (B_1 : A_1 \to \star) \to (B_2 : A_2 \to \star) \to \\ & ((a_1 : A_1) \to (a_2 : A_2) \to R_A\ a_1\ a_2 \to B_1\ a_1 \to B_2\ a_2 \to \star) \to \\ & \Sigma\ A_1\ B_1 \to \Sigma\ A_2\ B_2 \to \star\end{aligned}$$

The actual translation of $[\![\Sigma]\!]$ is straight forward, at least when we allow for minor abuse of notation in pattern-matching on the last two arguments:

$$[\![\Sigma]\!]\ A_0\ A_1\ R_A\ B_0\ B_1\ R_B\ (a_0, b_0)\ (a_1, b_1) = \Sigma\ (R_A\ a_0\ a_1)\ (\lambda R_a.\ R_B\ a_0\ a_1\ R_a\ b_0\ b_1)$$

Note the meaning of this definition: two pairs (a_0, b_0) and (a_1, b_1) are related if the as and bs are individually related, with the b relation conditioned on the proof that a_0 and a_1 are related.

Abstract (Data) Types. The application of Σ types that interests us here is the encoding of Abstract Data Types using Modules. Let us assume a module M which exports an abstract type T and a number of functions $f_i : F_i(T)$ whose type may mention T. As explained by Cardelli and Wegner [1985], in the Calculus of Constructions (CC) [Coquand and Huet 1988] with Σ types the above can be represented as a sigma type $\mathcal{M} = \Sigma(T : \star)P(T)$, where $P(T)$ is the product of function types $P(T) = \prod F_i(T)$. The implementation of the module M is represented as a value m of type $\mathcal{M}$; and importing the module M from a module N corresponds to taking an argument of the same type $(n = \lambda(m : \mathcal{M}).t)$. Linking corresponds to function application.

Assuming $m : \mathcal{M} \vdash t : A$, parametricity tells us that $m_0 : \mathcal{M}, m_1 : \mathcal{M}, R_m : [\![\mathcal{M}]\!]\ m_0\ m_1 \vdash [\![t]\!] : [\![A]\!]\ t_0\ t_1$. The above judgement can be illustrated using the parable given by Reynolds [1983] in his seminal paper, where students learn about complex numbers from two different professors. Each professor uses a different representation for complex numbers (cartesian or polar) — but because the rest of the course uses the same interface the students are not confused when switching professors. We could let here m_0 be a module containing a polar representation of complex numbers, m_1 be a module containing a cartesian one; relate them in the appropriate way (R_m), and get that any program t using them satisfies the parametricity interpretation of its type.

However, in this paper, we will focus on the case where there is a single implementation of the abstract module. Thus, we will have a single module m with a single type T and a single implementation of the functions f_i. According to the parametric interpretation of Σ, we then must pick a definition of $[\![T]\!]$ and ensure that they are related by constructing a proof of $[\![T]\!]\ T\ T$ and $[\![F_i]\!]\ f_i\ f_i$ for each i.

Example: Booleans. To understand how the choice of operators influences the parametricity interpretation of an abstract type we will try to define parametricity conditions for simple booleans.

We introduce the following interface, encoded in a Σ type as described above:

$$\begin{aligned}\texttt{Bool} &: \star \\ \texttt{true} &: \texttt{Bool} \\ \texttt{false} &: \texttt{Bool} \\ \texttt{if} &: (A : \star) \to \texttt{Bool} \to A \to A \to A\end{aligned}$$

Next we need to introduce a relation $R_{\texttt{Bool}}$, and proofs $R_{\texttt{true}}$, $R_{\texttt{false}}$, and $R_{\texttt{if}}$ each representing the parametricity conditions on their respective constant. This means that $R_{\texttt{Bool}} : [\![\star]\!]\ \texttt{Bool}\ \texttt{Bool}$, that is, by definition, $R_{\texttt{Bool}} : \texttt{Bool} \to \texttt{Bool} \to \star$. Notice that there is no strict requirement on $R_{\texttt{Bool}}$ yet. Right now we are free to pick any definition we want, say $R_{\texttt{Bool}}\ b_0\ b_1 = \top$.

Assuming this definition of $R_{\texttt{Bool}}$ for now, next we need to prove that $R_{\texttt{true}} : R_{\texttt{Bool}}\ \texttt{true}\ \texttt{true}$, which is easily done by $R_{\texttt{true}} = <>$, likewise for `false`. However, problems arise when we try to define $R_{\texttt{if}}$:

$$\begin{aligned}R_{\texttt{if}} : {} & (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \\ & (b_0, b_1 : \texttt{Bool}) \to R_{\texttt{Bool}}\ b_0\ b_1 \to \\ & (x_0 : A_0) \to (x_1 : A_1) \to R_A\ x_0\ x_1 \to \\ & (y_0 : A_0) \to (y_1 : A_1) \to R_A\ y_0\ y_1 \to \\ & R_A\ (\texttt{if}\ A_0\ b_0\ x_0\ y_0)\ (\texttt{if}\ A_1\ b_1\ x_1\ y_1)\end{aligned}$$

Here we have to show that given *any* two booleans (remember that we set $R_{\texttt{Bool}}\ b_0\ b_1$ to be $\top$) the two `if`s return related results. Specifically, we have to show:

$$R_A\ (\texttt{if}\ A_0\ \texttt{true}\ x_0\ y_0)\ (\texttt{if}\ A_1\ \texttt{false}\ x_1\ y_1)$$

which is the same as $R_A\ x_0\ y_1$. Furthermore by inspecting the symmetric case we see that we also need to show $R_A\ x_1\ y_0$. This is clearly not possible given the arguments to $R_{\texttt{if}}$. However, what is at fault is not the parametricity theory, but rather the choice of $R_{\texttt{Bool}}$. Instead we choose:

$$R_{\texttt{Bool}}\ b_0\ b_1 = b_0 \equiv_{\texttt{Bool}} b_1$$

The translation of `if` now becomes straightforward as we only need to consider the cases where both b_0 and b_1 are either `true` or `false` and we can rely on the two proofs of $R_A\ x_0\ y_0$ and $R_A\ x_1\ y_1$ respectively.

Then, every program using Booleans (*via* the interface spelled above) can then be associated a parametric interpretation, and thus to every type will correspond a free theorem.

3 INTUITION BEHIND THE PROOF OF NONINTERFERENCE FROM PARAMETRICITY

To prove noninterference using parametricity, we will formalise a very simple intuition present in many security libraries. Security libraries like MAC [Russo 2015] and LIO [Stefan et al. 2011] rely on defining secure types by hiding parts of the implementation from the user — that is, they effectively define an abstract type. An early example is that of Russo et al. [2008]:

```
module Sec (S, return, bind, up) where

data S (l :: Lattice) a = Hide a

return :: a -> S l a
return a = Hide a
```

```
up :: l `CanFlowTo` h => S l a -> S h a
up (Hide a) = Hide a

bind :: S l a -> (a -> S l b) -> S l b
Hide a `bind` f = f a
```

The above module makes the type `S` abstract by hiding the `Hide` constructor. The security lattice which encodes security levels is represented as a kind `Lattice`, where the order of the lattice ($\sqsubseteq$) is encoded using the type class `CanFlowTo`. The role of the `Lattice` type parameter in the type `S` is to track the "sensitivity" (for example, the confidentiality level) of the information which contributed to the computation. Russo et al. argue, by modelling the clients of this library as a stand-alone program calculus, that this is sufficient to ensure noninterference for programs constructed using this module. The idea is that any value of type `a` is boxed in a monadic type `S l a` and values can be promoted only to levels more secret than `l`. The `return` combinator simply boxes secret data at a level l. The `up` combinator promotes data from one level in the lattice to a higher level, as indicated by the `l `CanFlowTo` h` constraint, which represents the standard $l \sqsubseteq h$ lattice ordering. Finally the `bind` combinator takes a secret at level l, a type a and a continuation which computes a new secret at the same level (but of type b) based on the value of the first secret and gives back the result. Russo et al. prove that the clients of the above library are noninterfering by using a custom program calculus with multiple custom semantics. In our development, we instead use the standard parametric interpretation of abstract types (as explained in the previous section).

The interface of the module, converted to a Σ-type using the recipe of Cardelli and Wegner [1985], is the following:

$$\begin{aligned}\mathsf{Sec} = \Sigma\ (S : \mathcal{L} \to \star \to \star)\\ (\ \mathtt{return} : (A : \star) \to (\ell : \mathcal{L}) \to A \to S\ \ell\ A\\ ,\ \mathtt{bind} : (A, B : \star) \to (\ell : \mathcal{L}) \to S\ \ell\ A \to (A \to B) \to S\ \ell\ B\\ ,\ \mathtt{up} : (A : \star) \to (\ell, \ell' : \mathcal{L}) \to \ell \sqsubseteq \ell' \to S\ \ell\ A \to S\ \ell'\ A)\end{aligned}$$

This type can be parsed either as nested Σ-types or as a single Σ containing S followed by simple products — this makes no essential difference. The above type goes a bit beyond the usual abstract data type. Indeed, instead of hiding a single type, it hides a family of types $S\ \ell\ A$, one for each ℓ and A. However the essence of the idea remains: we provide $S : \mathcal{L} \to \star \to \star$, together with a signature of functions operating on it, but without providing any access to the implementation.

Any code written using this module is modeled as a function parameterised over the Sec type. To see how this works consider the code below:

```
example :: Sec.S L Bool -> Sec.S H Bool
example s = S.up (S.bind s (\x -> S.return (not x)))
```

In CC, this corresponds to a function:

$$\begin{aligned}&\mathtt{example} : (m : \mathsf{Sec}) \to (S\ m)\ \mathsf{L}\ \mathtt{Bool} \to (S\ m)\ \mathsf{H}\ \mathtt{Bool}\\ &\mathtt{example} = \ldots\end{aligned}$$

(the body of the function makes no difference). We also have:

$$\begin{aligned}&S : \mathsf{Sec} \to (\mathcal{L} \to \star \to \star)\\ &S\ (s, \_) = s\end{aligned}$$

Similarly the functions in the module can be extracted from the module:

$$\begin{aligned}&\texttt{return} : (m : \texttt{Sec}) \to (A : \star) \to (\ell : \mathcal{L}) \to A \to (S\ m)\ \ell\ A\\&\texttt{return}\ (\_, (r, \_)) = r\end{aligned}$$

The upcoming parts of this paper will make extensive use of the techniques presented above. Encoding modules and information-hiding as functions which are parametric in the implementation of the module allows us to build upon the standard parametric interpretation of abstract types. Doing so shows that noninterference is a consequence of the standard notion of abstract data type. Additionally, and as it will become apparent in Sections 4 and 5 this structure gives rise to short and simple proofs of noninterference.

4 NONINTERFERENCE FROM PARAMETRICITY

In this section we make use of the encoding of modules in Section 3 alongside parametricity for Σ types described in Section 2 to give a simple proof of noninterference for a language similar to the **Sec** language of Russo et al. [2008]. In the following development we assume a type $\mathcal{L}$ denoting the security lattice. We take the parametric interpretation of $\mathcal{L}$ to be point-wise equality $[\![\mathcal{L}]\!] = (\equiv)$. This decision is in the interest of simplicity: we do not need any more complicated relation. One benefit of this definition of $[\![\mathcal{L}]\!]$ is that in any place where the parametricity condition states that we have two related ℓ, $(\ell_0, \ell_1 : \mathcal{L}) \to [\![\mathcal{L}]\!]\ \ell_0\ \ell_1 \to T$ we can substitute $(\ell : \mathcal{L}) \to [\ell/\ell_i]T$, where $[\ell/\ell_i]T$ denotes the capture-free substitution of ℓ for ℓ_0 and ℓ_1 in T, without loss of generality. We will use $(\sqsubseteq) : \mathcal{L} \to \mathcal{L} \to \star$ as the ordering on $\mathcal{L}$. As standard we expect this relation to come equipped with transitivity and reflexivity. The inhabitants of $\ell_0 \sqsubseteq \ell_1$ are proofs and carry no useful computational content for our purposes, so we take the corresponding parametricity relation to be pointwise equality, with the same justification as for $\mathcal{L}$. Finally, we assume a designated level $\hat{\ell} : \mathcal{L}$, called the *observer level*. Data at level ℓ such that $\ell \sqsubseteq \hat{\ell}$ can be thought of as public whereas data labeled above $\hat{\ell}$ is private.

Because we are aiming at proving the implementation of the **Sec** module above correct, we restate the type of the module here for clarity:

$$\begin{aligned}\texttt{Sec} = \Sigma\ (&S : \mathcal{L} \to \star \to \star)\\&(\ \texttt{return} : (A : \star) \to (\ell : \mathcal{L}) \to A \to S\ \ell\ A\\&,\ \texttt{bind} : (A, B : \star) \to (\ell : \mathcal{L}) \to S\ \ell\ A \to (A \to S\ \ell\ B) \to S\ \ell\ B\\&,\ \texttt{up} : (A : \star) \to (\ell, \ell' : \mathcal{L}) \to \ell \sqsubseteq \ell' \to S\ \ell\ A \to S\ \ell'\ A)\end{aligned}$$

Next we need to tackle the implementation, which will be a simple translation of the Haskell implementation into CC. Following Russo et al. we take the implementation of $S\ \ell\ A$ to be simply A — ignoring ℓ. Consequently, the implementation of the functions in the module simply ignores

the security levels and the proofs that data can flow from one level to another:

$$
\begin{aligned}
&S : \mathcal{L} \to \star \to \star \\
&S\ \ell\ A = A \\
&\mathsf{return} : (A : \star) \to (\ell : L) \to A \to S\ \ell\ A \\
&\mathsf{return}\ A\ \ell\ a = a \\
&\mathsf{bind} : (A, B : \star) \to (\ell : \mathcal{L}) \to S\ \ell\ A \to (A \to S\ \ell\ B) \to S\ \ell\ B \\
&\mathsf{bind}\ A\ B\ \ell\ s\ f\ = f\ s \\
&\mathsf{up} : (A : \star) \to (\ell, \ell' : \mathcal{L}) \to \ell \sqsubseteq \ell' \to S\ \ell\ A \to S\ \ell'\ A \\
&\mathsf{up}\ A\ B\ \ell\ \ell'\ p\ s = s
\end{aligned}
$$

The complete implementation of the module $sec : \mathsf{Sec}$ consists of a tuple of the above:

$$
\begin{aligned}
&sec : \mathsf{Sec} \\
&sec = (S, \mathsf{return}, \mathsf{bind}, \mathsf{up})
\end{aligned}
$$

Our goal is to show that object functions written using this module cannot leak any information. The first key observation is that the the parametricity translation of these object functions will express that each of them satisfies the non-interference property. Let us review the Parametricity condition for functions taking a Sec argument. If $o : (imp : \mathsf{Sec}) \to A$ we have:

$$[\![o]\!] : (imp_0, imp_1 : \mathsf{Sec}) \to (R_{imp} : [\![\mathsf{Sec}]\!]\ imp_0\ imp_1) \to [\![A]\!]$$

Because we are concerned with o applied to the specific implementation sec, we must provide an $R_{sec} : [\![\mathsf{Sec}]\!]\ sec\ sec$ denoting the interpretation of the sec module. Having this gives us the following:

$$[\![o]\!]\ sec\ sec\ R_{sec} : [sec/imp_i][\![A]\!]$$

To see how to construct R_{sec} to allow us to prove noninterference, recall that Sec is a Σ type and that the translation of a Σ type is a Σ of translations.

$$R_{sec} = (R_S, R_{\mathsf{return}}, R_{\mathsf{bind}}, R_{\mathsf{up}})$$

where each of these R_x relates the terms of the same type as the corresponding members of the sec module.

$$
\begin{aligned}
&R_S : [\![\mathcal{L} \to \star \to \star]\!]\ S\ S \\
&R_{\mathsf{return}} : [\![(A : \star) \to (\ell : L) \to A \to S\ \ell\ A]\!]\ \mathsf{return}\ \mathsf{return} \\
&R_{\mathsf{bind}} : [\![(A, B : \star) \to (\ell : \mathcal{L}) \to S\ \ell\ A \to (A \to S\ \ell\ B) \to S\ \ell\ B]\!]\ \mathsf{bind}\ \mathsf{bind} \\
&R_{\mathsf{up}} : [\![(A : \star) \to (\ell, \ell' : \mathcal{L}) \to \ell \sqsubseteq \ell' \to S\ \ell\ A \to S\ \ell'\ A]\!]\ \mathsf{up}\ \mathsf{up}
\end{aligned}
$$

What we have in $[\![o]\!]\ sec\ sec\ R_{sec}$, then, is a proof that o respects the relations and proofs we instantiate R_x with. This realisation is the key fact that underlies the rest of this paper. Parametricity asks us to provide the security conditions and in return we get the proof that the code we write obeys them — for free. In other words, what we are saying is that the combinators in the language are all *locally* secure and parametricity gives us the proof that their resulting combination is secure. In other words, this proof technique is entirely compositional.

The rest of this section is concerned with the construction of these conditions; the definition of each R_x. The first and most important one of these is R_S. To see clearly what is going on here we

expand the type of R_S:

$$R_S : (\ell : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to S\ \ell\ A_0 \to S\ \ell\ A_1 \to \star$$

Recall that we implement $S\ \ell\ A$ as A. So, a natural candidate for $R_S\ \ell\ A_0\ A_1\ R_A$ is simply R_A. However, if we choose this definition, we will not be hiding any information and no proof of noninterference could ever go through. The key is to encode the idea of observer-sensitive equivalence in this relation. Thus, we condition the relation R_A by $\ell \sqsubseteq \hat{\ell}$, recall that $\hat{\ell}$ is the fixed observer level. That is, when $\ell \not\sqsubseteq \hat{\ell}$, no information can be extracted from the relation, encoding the property that secret inputs are opaque to the observer.

$$R_S\ \ell\ A_0\ A_1\ R_A\ s_0\ s_1 = (\ell \sqsubseteq \hat{\ell}) \to R_A\ s_0\ s_1$$

This definition is the root of non-interference. It will be expanded in the types of the other combinators, ensuring that they all satisfy it. Our task will be to fulfill the corresponding condition R_x for each combinator x. We start with the straightforward implementation of $R_{\texttt{return}}$:

$$\begin{aligned} R_{\texttt{return}}\ :\ & (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to \\ & \mathcal{L} \to (a_0 : A_0) \to (a_1 : A_1) \to (R_a : R_A\ a_0\ a_1) \to \\ & R_S\ \ell\ A_0\ A_1\ R_A\ (\texttt{return}\ A_0\ \ell\ a_0)\ (\texttt{return}\ A_0\ \ell\ a_1) \\ R_{\texttt{return}}\ & A_0\ A_1\ R_A\ \ell\ a_0\ a_1\ R_a = \lambda(p : \ell \sqsubseteq \hat{\ell}).\ R_a \end{aligned}$$

We get the proof that the two secrets are related precisely because the values being made secret are related in the first place.

Next we attack $R_{\texttt{bind}}$. When expanding the type of $R_{\texttt{bind}}$ we get the following:

$$\begin{aligned} R_{\texttt{bind}} : & (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to \\ & (B_0, B_1 : \star) \to (R_B : B_0 \to B_1 \to \star) \to \\ & (\ell : \mathcal{L}) \to (s_0 : S\ \ell\ A_0) \to (s_1 : S\ \ell\ A_1) \to \\ & R_S\ \ell\ A_0\ A_1\ R_A\ s_0\ s_1 \to \\ & (f_0 : A_0 \to S\ \ell\ B_0) \to (f_1 : A_1 \to S\ \ell\ B_1) \to \\ & ((a_0 : A_0) \to (a_1 : A_1) \to R_A\ a_0\ a_1 \to R_S\ \ell\ B_0\ B_1\ R_B\ (f_0\ a_0)\ (f_1\ a_1)) \to \\ & R_S\ \ell\ B_0\ B_1\ R_B\ (\texttt{bind}\ A_0\ B_0\ \ell\ s_0\ f_0)\ (\texttt{bind}\ A_1\ B_1\ \ell\ s_1\ f_1) \end{aligned}$$

This expression may look big and intimidating. But, the most important takeaway is that given two related inputs s_0 and s_1 and functions which preserve relatedness f_0 and f_1, we must produce a proof that the outputs of `bind` applied to s_i and f_i are related at the output relation $R_S \ldots R_B$. After expanding the definition of `bind`, the target type becomes simply $R_S\ \ell\ B_0\ B_1\ R_B\ (f_0\ s_0)\ (f_1\ s_1)$. In this light we can give the surprisingly simple implementation of $R_{\texttt{bind}}$:

$$R_{\texttt{bind}}\ A_0\ A_1\ R_A\ B_0\ B_1\ R_B\ \ell\ s_0\ s_1\ R_s\ f_0\ f_1\ R_f = \lambda(p : \ell \sqsubseteq \hat{\ell}).\ R_f\ s_0\ s_1\ (R_s\ p)\ p$$

Finally we approach $R_{\texttt{up}}$. This proof makes use of the transitivity of $(\sqsubseteq)$ to prove that if $\ell \sqsubseteq \ell'$ and $\ell' \sqsubseteq \hat{\ell}$ then $\ell \sqsubseteq \hat{\ell}$. For this to work $(\sqsubseteq)$ needs to come equipped with transitivity:

$$\texttt{trans-flow} : (\ell, \ell', \ell'' : \mathcal{L}) \to (\ell \sqsubseteq \ell') \to (\ell' \sqsubseteq \ell'') \to (\ell \sqsubseteq \ell'')$$

With this in place the definition of $R_{\texttt{up}}$ is simple:

$$R_{\texttt{up}}\ A_0\ A_1\ R_A\ \ell\ \ell'\ p\ s_0\ s_1\ R_s = \lambda p' : (\ell' \sqsubseteq \hat{\ell}).\ R_s\ (\texttt{trans-flow}\ \ell\ \ell'\ \hat{\ell}\ p\ p')$$

We are now done with the core of the *sec* module as introduced by Russo et al..

Having established a parametric interpretation of the Sec module which encodes observer-sensitive equivalence, we can state the Noninterference theorem. We prove the theorem here in the same form provided by Russo et al. for the calculus which inspires Sec.

Theorem 2 (Noninterference). *Given $A : \star$ and $\ell : \mathcal{L}$ such that $\ell \not\sqsubseteq \hat{\ell}$ for any $f : (sec : \mathtt{Sec}) \to (S\ sec)\ \ell\ A \to (S\ sec)\ \hat{\ell}\ \mathtt{Bool}$ and any two $a_0, a_1 : A$ we have $f\ sec\ a_0 \equiv f\ sec\ a_1$ where sec is the module defined in this section.*

Proof. By Lemma 1 (Parametricity) we have:

$$[\![f]\!]\ sec\ sec\ R_{sec} : (a_0, a_1 : S\ \ell\ A) \to R_S\ \ell\ A\ A\ [\![A]\!]\ a_0\ a_1 \to R_S\ \hat{\ell}\ \mathtt{Bool}\ \mathtt{Bool}\ [\![\mathtt{Bool}]\!]\ (f\ sec\ a_0)\ (f\ sec\ a_1)$$

Let p be the witness of $\ell \not\sqsubseteq \hat{\ell}$. By the definition of S and R_S we have, for any $a_0, a_1 : A$:

$$[\![f]\!]\ sec\ sec\ R_{sec}\ a_0\ a_1\ (\lambda q.\ \mathtt{ex\text{-}falso}\ (p\ q)\ ([\![A]\!]\ a_0\ a_1))\ (\mathtt{refl\text{-}flow}\ \hat{\ell})\ : f\ sec\ a_0 \equiv f\ sec\ a_1$$

Where $\mathtt{ex\text{-}falso} : \bot \to (A : \star) \to A$ is the standard eliminator for the empty type $\bot$ and $\mathtt{refl\text{-}flow} : (\ell : \mathcal{L}) \to \ell \sqsubseteq \ell$ is the witness of reflexivity for $\sqsubseteq$ □

5 SHALLOW EMBEDDING OF DEPENDENCY CORE CALCULUS

The small language of protected values presented in Section 4 is simple, yet forms the core of other languages with similar capabilities. Most notably, the language is one primitive short of replicating the Dependency Core Calculus (DCC) of Abadi et al. [1999]. The terminating fragment of DCC is a simply typed lambda calculus with one special construct, a family of monads T indexed by elements of a lattice of security levels ($T : \mathcal{L} \to \star \to \star$). This construct plays the same role as S: values of type $T\ \ell\ A$ are constructed using the constructor $\eta_\ell : A \to T\ \ell\ A$ and combined using a primitive bind with a special typing rule:

$$\frac{\Gamma \vdash e : T\ \ell\ A \quad \Gamma, x : A \vdash e' : B \quad \ell \preceq B}{\Gamma \vdash \mathtt{bind}\ x = e\ \mathtt{in}\ e' : B}$$

There are two things going on in this definition: a binding and a side condition. The binding of the variable x plays the role of binding the secret value inside e in the expression e'. The idea of the side-condition $\ell \preceq B$, read "B is protected at ℓ", is to ensure that it is safe for information to flow from a value in a secure context at ℓ into a value of type B, as long as the type B protects secrets at level at least ℓ. The operational semantics of bind bring nothing new compared to Sec: there is a congruence rule in the bound expression and a rule for extracting secret values from an η_ℓ.

$$\frac{}{\mathtt{bind}\ x = \eta_\ell\ a\ \mathtt{in}\ e \longrightarrow [a/x]e}$$

It remains to explain how $\ell \preceq B$ captures the property that B protects secrets at level ℓ, or in other words, when the structure of B cannot leak any information to an observer below ℓ.

The relation $\ell \preceq B$ is defined inductively by the following rules:

$$\mathtt{protect}_T\ \frac{\ell \sqsubseteq \ell'}{\ell \preceq T\ \ell'\ A} \qquad \mathtt{protect}_{T'}\ \frac{\ell \preceq A}{\ell \preceq T\ \ell'\ A} \qquad \mathtt{protect}_\times\ \frac{\ell \preceq A \quad \ell \preceq B}{\ell \preceq A \times B}$$

$$\mathtt{protect}_\to\ \frac{\ell \preceq B}{\ell \preceq A \to B}$$

The first rule in this four-part definition tells us that moving up the lattice preserves secrets: $\ell \preceq T\ \ell' A$ holds as long as $\ell \sqsubseteq \ell'$ holds as well. The second rule says that wrapping something in a $T\ \ell$ cannot make it less secret. The third rule says that the structure of a product cannot leak any information if the two types in the product do not leak. The intuition for why this makes sense

is that given a value of type $T\ \ell\ (A \times B)$ we know that it must contain precisely two secrets, one of type A and one of type B, likewise $T\ \ell\ A \times T\ \ell\ B$ also contains precisely two secrets. Finally, functions producing secret things are also secret-preserving. Most notable in this definition is that there are no rules which allow us to derive things like $\ell \preceq \texttt{Bool}$, because booleans can be used to leak one bit of information. Similarly, there is no way to form $\ell \preceq A + B$ or $\ell \preceq \texttt{List}\ A$, for any A or B. The reason for this is that the structure of co-products and lists can be enough to leak information. For example, the length of a list can encode information regardless of the contents of the list.

In this section we extend the module given in the previous section (implementing the `Sec` language) to an implementation of a shallow embedding of DCC, as a `DCC` module. In keeping with the terminology of DCC, we rename the S type constructor to T and the R_S relation to R_T. Instead of implementing the DCC `bind` primitive directly, we consider two slightly simpler but equally expressive `map` and `join` primitives.

$$\texttt{join} : (\ell : \mathcal{L}) \to (A : \star) \to \ell \preceq A \to T\ \ell\ A \to A$$
$$\texttt{map} : (\ell : \mathcal{L}) \to (A, B : \star) \to (A \to B) \to T\ \ell\ A \to T\ \ell\ B$$

Indeed, using these, `bind` can be straightforwardly implemented:

$$\texttt{bind} : (\ell : \mathcal{L}) \to (A\ B : \star) \to (\ell \preceq B) \to T\ \ell\ A \to (A \to B) \to B$$
$$\texttt{bind}\ \ell\ A\ B\ p\ a\ k = \texttt{join}\ \ell\ B\ p\ (\texttt{map}\ \ell\ A\ B\ k\ a)$$

Note that this definition represents the variable binding in the DCC `bind` primitive using a higher-order function instead of implementing it directly in the typing relation. The two formulations are however equivalent; a program using the higher-order formulation corresponds to a program under the other formulation and vice-verse. Next we need to add the relation $(\preceq) : \mathcal{L} \to \star \to \star$ to the *dcc* module, as well as the rules that define it.

$$\texttt{protect}_T : (\ell, \ell' : \mathcal{L}) \to (A : \star) \to (\ell \sqsubseteq \ell') \to \ell \preceq (T\ \ell'\ A)$$
$$\texttt{protect}_{T'} : (\ell, \ell' : \mathcal{L}) \to (A : \star) \to (\ell \preceq A) \to \ell \preceq (T\ \ell'\ A)$$
$$\texttt{protect}_\times : (\ell : \mathcal{L}) \to (A, B : \star) \to (\ell \preceq A) \to (\ell \preceq B) \to \ell \preceq (A \times B)$$
$$\texttt{protect}_\to : (\ell : \mathcal{L}) \to (A, B : \star) \to (\ell \preceq B) \to \ell \preceq (A \to B)$$

To sum up, instead of the `Sec` type from Section 4 we have now a `DCC` type:

$$\begin{aligned}
\texttt{DCC} = \Sigma\ (&T : \mathcal{L} \to \star \to \star) \\
&(\ (\preceq) : \mathcal{L} \to \star \to \star \\
&,\ \texttt{return} : (A : \star) \to (\ell : \mathcal{L}) \to A \to T\ \ell\ A \\
&,\ \texttt{map} : (A, B : \star) \to (\ell : \mathcal{L}) \to (A \to B) \to T\ \ell\ A \to T\ \ell\ B \\
&,\ \texttt{join} : (\ell : \mathcal{L}) \to (A : \star) \to (\ell \preceq A) \to T\ \ell\ A \to A \\
&,\ \texttt{protect}_T : (\ell, \ell' : \mathcal{L}) \to (A : \star) \to (\ell \sqsubseteq \ell') \to \ell \preceq (T\ \ell'\ A) \\
&,\ \texttt{protect}_{T'} : (\ell, \ell' : \mathcal{L}) \to (A : \star) \to (\ell \preceq A) \to \ell \preceq (T\ \ell'\ A) \\
&,\ \texttt{protect}_\times : (\ell : \mathcal{L}) \to (A, B : \star) \to (\ell \preceq A) \to (\ell \preceq B) \to \ell \preceq (A \times B) \\
&,\ \texttt{protect}_\to : (\ell : \mathcal{L}) \to (A, B : \star) \to (\ell \preceq B) \to \ell \preceq (A \to B))
\end{aligned}$$

These abstract types and functions, along with an implementation, constitute a shallow embedding of DCC into CC. We can start looking at the implementation of this embedding, which will form our module $dcc : \texttt{DCC}$. The implementation of T is the same as for S in Section 4, namely $T\ \ell\ A = A$, from which the definitions of `return`, `map`, and `join` follow trivially. Next we need

an implementation of $(\preceq)$. Note that the type $\ell \preceq A$ needs to correspond to a proof that ℓ and A are related as constructed by the rules above ($\texttt{protect}_T$, $\texttt{protect}_{T'}$, ...). However, because we choose $T\ \ell\ A$ to be implemented as A, we do not need any specific information from the witnesses $\ell \preceq A$. Effectively, in the implementation, all security levels are erased — non-interference is ensured purely statically. For this reason we choose $\ell \preceq A = \top$, and given this implementation the implementations of the $\texttt{protect}_x$ functions are all trivial. As a result, we need not move from CC to the Calculus of Inductive Constructions [Bertot and Castéran 2013] even though the original definition of $\preceq$ is as an inductive relation.

Having given the interface and the implementation of the *dcc* module, we turn to its parametricity interpretation.

The key ingredient is the relation associated with $(\preceq)$.

$$R_\preceq : (\ell : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to \ell \preceq A_0 \to \ell \preceq A_1 \to \star$$
$$R_\preceq\ \ell\ A_0\ A_1\ R_A\ p_0\ p_1 = (\ell \not\sqsubseteq \hat{\ell}) \to (a_0 : A_0) \to (a_1 : A_1) \to R_A\ a_0\ a_1$$

Contrary what one might expect, this relation does not restrict its arguments p_0 and p_1. Indeed, those contain no information — so there is nothing to restrict. Instead, we choose $R_\preceq$ to restrict the *relation* R_A. In essence, what we did not do in $\preceq$ itself, we do here. The restriction is the following: if $(\ell \not\sqsubseteq \hat{\ell})$, then R_A must be the full relation. Intuitively, this means that if the types A_0 and A_1 are secret, then R_A must consider their values indistinguishable. The cognoscenti will note that this definition is very close to **Proposition 3.2** in Abadi et al.'s original text on DCC [Abadi et al. 1999]. We will see shortly how this definition happens to fit our needs.

Can this condition be ensured by the functions that produce $\preceq$, namely $\texttt{protect}_X$? We focus on the definition of the two most interesting cases, $R_{\texttt{protect}_T}$ and $R_{\texttt{protect}_{T'}}$. For $R_{\texttt{protect}_T}$ we are looking for an inhabitant of the following type:

$$R_{\texttt{protect}_T} : (\ell, \ell' : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to (p : \ell \sqsubseteq \ell') \to$$
$$R_\preceq\ \ell\ (T\ \ell'\ A_0)\ (T\ \ell'\ A_1)\ (R_T\ \ell'\ A_0\ A_1\ R_A)\ (R_{\texttt{protect}_T}\ \ell\ \ell'\ A_0\ p)\ (R_{\texttt{protect}_S}\ \ell\ \ell'\ A_1\ p)$$

Recall that $\texttt{protect}_T\ \ell\ \ell'\ A_i\ p = <>$ and $R_T\ \ell'\ A_0\ A_1\ R_A\ t_0\ t_1 = (\ell \sqsubseteq \hat{\ell}) \to R_A\ t_0\ t_1$. Hence the above type signature is equivalent to the simpler:

$$R_{\texttt{protect}_T} : (\ell, \ell' : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to$$
$$(\ell \sqsubseteq \ell') \to (\ell \not\sqsubseteq \hat{\ell}) \to (t_0 : T\ \ell'\ A_0) \to (t_1 : S\ \ell'\ A_1) \to$$
$$(\ell' \sqsubseteq \hat{\ell}) \to R_A\ t_0\ t_1$$

The implementation follows naturally from the transitivity of $(\sqsubseteq)$, which gives us the contradictory fact that $\ell \sqsubseteq \hat{\ell}$ and $\ell \not\sqsubseteq \hat{\ell}$ hold at the same time. Thus the conclusion $(R_A\ t_0\ t_1)$ holds vacuously. To find the type of $R_{\texttt{protect}_{T'}}$, we expand the $[\![\_]\!]$ brackets in the type and simplify it by removing $\top$ arguments. We get that the type of $R_{\texttt{protect}_{T'}}$ is equivalent to the following:

$$(\ell, \ell' : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to$$
$$((\ell \not\sqsubseteq \hat{\ell}) \to (a_0 : A_0) \to (a_1 : A_1) \to R_A\ a_0\ a_1) \to$$
$$(\ell \not\sqsubseteq \hat{\ell}) \to (t_0 : T\ \ell\ A_0) \to (t_1 : T\ \ell\ A_1) \to$$
$$(\ell' \sqsubseteq \hat{\ell}) \to R_A\ t_0\ t_1$$

We can see that by ignoring the last argument and by the fact that $T\ \ell\ A = A$ the type can be simplified to the following:

$$\begin{aligned}&(\ell, \ell' : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to \\ &((\ell \not\sqsubseteq \hat{\ell}) \to (a_0 : A_0) \to (a_1 : A_1) \to R_A\ a_0\ a_1) \to \\ &(\ell \not\sqsubseteq \hat{\ell}) \to (a_0 : A_0) \to (a_1 : A_1) \to R_A\ a_0\ a_1\end{aligned}$$

In turn, this means that $R_{\mathtt{protect}_{T'}}$ is also trivially implementable. The other cases, $R_{\mathtt{protect}_\times}$ and $R_{\mathtt{protect}_\to}$ follow structurally, using the $\ell \preceq \ldots$ arguments in their respective definitions.

We are now ready to set the keystone of the construction, namely the full definition $R_{\mathtt{join}}$. To inhabit the type of $R_{\mathtt{join}}$:

$$\begin{aligned}R_{\mathtt{join}} : &(\ell : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : \star) \to \\ &(p_0 : \ell \preceq A_0) \to (p_1 : \ell \preceq A_1) \to R_\preceq\ \ell\ A_0\ A_1\ R_A\ p_0\ p_1 \to \\ &(t_0 : T\ \ell\ A_0) \to (t_1 : T\ \ell\ A_1) \to R_T\ \ell\ A_0\ A_1\ R_A\ t_0\ t_1 \to \\ &R_A\ (\mathtt{join}\ \ell\ A_0\ p_0\ s_0)\ (\mathtt{join}\ \ell\ A_1\ p_1\ s_1)\end{aligned}$$

we must produce a proof that $R_A\ t_0\ t_1$ holds — ostensibly from thin air. Since if $\ell \sqsubseteq \hat{\ell}$ holds then we can use the proof of $R_T\ \ell\ A_0\ A_1\ R_A\ t_0\ t_1$ to produce a proof that $R_A\ t_0\ t_1$ holds (recall $R_T\ \ell\ A_0\ A_1\ R_A\ t_0\ t_1 = (\ell \sqsubseteq \hat{\ell}) \to R_A\ t_0\ t_1$). If $\ell \sqsubseteq \hat{\ell}$ does not hold, then we had carefully prepared the definition of $R_\preceq$ to give us exactly what we wanted.

Hence, to be able to conclude, we are only missing the decidability of the $(\sqsubseteq)$ relation, namely a function $(\widetilde{\sqsubseteq})$ which produces proofs of $(\sqsubseteq)$ or its negation[2]:

$$(\widetilde{\sqsubseteq}) : (\ell, \ell' : \mathcal{L}) \to (\ell \sqsubseteq \ell') + (\ell \not\sqsubseteq \ell')$$

We promptly add this function to the list of requirements, allowing us to define $R_{\mathtt{join}}$. The definition proceeds by case-split on $\ell \sqsubseteq \hat{\ell}$:

$$\begin{aligned}&R_{\mathtt{join}}\ \ell\ A_0\ A_1\ R_A\ p_0\ p_1\ R_\preceq\ t_0\ t_1\ R_T = \\ &\quad \mathtt{case}\ (\ell \mathbin{\widetilde{\sqsubseteq}} \hat{\ell})\ \mathtt{of} \\ &\qquad \mathtt{inl}\ p \to R_T\ p \\ &\qquad \mathtt{inr}\ p \to R_\preceq\ p\ t_0\ t_1\end{aligned}$$

In the interest of completeness we also include the definition of $R_{\mathtt{map}}$. Because of the higher-order type of `map`, the type for $R_{\mathtt{map}}$ is somewhat involved:

$$\begin{aligned}R_{\mathtt{map}} : &(\ell : \mathcal{L}) \to (A_0, A_1 : \star) \to (R_A : A_0 \to A_1 \to \star) \to \\ &(B_0, B_1 : \star) \to (R_B : B_0 \to B_1 \to \star) \to \\ &(t_0 : T\ \ell\ A_0) \to (t_1 : T\ \ell\ A_1) \to R_T\ \ell\ A_0\ A_1\ t_0\ t_1 \to \\ &(f_0 : A_0 \to B_0) \to (f_1 : A_1 \to B_1) \to \\ &(R_f : (a_0 : A_0) \to (a_1 : A_1) \to R_A\ a_0\ a_1 \to R_B\ (f_0\ a_0)\ (f_1\ a_1)) \to \\ &R_T\ \ell\ B_0\ B_1\ R_B\ (\mathtt{map}\ \ell\ A_0\ B_0\ t_0\ f_0)\ (\mathtt{map}\ \ell\ A_1\ B_1\ t_1\ f_1)\end{aligned}$$

[2]Note that the type of $(\widetilde{\sqsubseteq})$ requires that we have co-products in the language. It is however well known that these can be encoded using a standard church-encoding so we do not need to change anything in the type-system or the definition of $[\![\_]\!]$. Regardless, the proofs that we write make use of Haskell-style `case ... of ...` syntax for ease of reading when deconstructing values of type $A + B$.

What this type says is that given that the functions f_0 and f_1 return related results given related arguments, relatedness at R_T is preserved. Defining R_{map} is straightforward:

$$R_{\mathsf{map}}\ \ell\ A_0\ A_1\ R_A\ B_0\ B_1\ R_B\ t_0\ t_1\ R_t\ f_0\ f_1\ R_f = \lambda p : (\ell \sqsubseteq \hat{\ell}).\ R_f\ t_0\ t_1\ (R_t\ p)$$

With all the relations in place we are ready once again for the noninterference theorem.

Theorem 3 (Noninterference). *Given $A : \star$ and $\ell : \mathcal{L}$ such that $\ell \not\sqsubseteq \hat{\ell}$ for any $f : (dcc : \mathtt{DCC}) \to (T\ dcc)\ \ell\ A \to (T\ dcc)\ \hat{\ell}\ \mathtt{Bool}$ and any two $a_0, a_1 : A$ we have $f\ dcc\ a_0 \equiv f\ dcc\ a_1$ where dcc is the module defined in this section.*

Proof. By Lemma 1 (Parametricity) we have:

$$[\![f]\!]\ dcc\ dcc\ R_{dcc} : (a_0, a_1 : T\ \ell\ A) \to R_T\ \ell\ A\ A\ [\![A]\!]\ a_0\ a_1 \to R_T\ \hat{\ell}\ \mathtt{Bool}\ \mathtt{Bool}\ [\![\mathtt{Bool}]\!]\ (f\ sec\ a_0)\ (f\ sec\ a_1)$$

Let p be the witness of $\ell \not\sqsubseteq \hat{\ell}$. By the definition of T and R_T we have, for any $a_0, a_1 : A$:

$$[\![f]\!]\ dcc\ dcc\ R_{dcc}\ a_0\ a_1\ (\lambda q.\ \mathtt{ex\text{-}falso}\ (p\ q)\ ([\![A]\!]\ a_0\ a_1))\ (\mathtt{refl\text{-}flow}\ \hat{\ell})\ : f\ sec\ a_0 \equiv f\ sec\ a_1$$

□

Having tackled Noninterference, the next question is how this proof of Noninterference compares to the original by Abadi et al. [1999]. The denotational semantics of DCC is as objects and relations in a category of complete partial orders, where types are represented as sets and relations on sets. In that semantics, Abadi et al. obtain a sort of Noninterference theorem in the following form:

> If t is a type protected at level ℓ and $\ell \not\sqsubseteq \hat{\ell}$ then the relation associated with $|t|$ is the total relation.

Where $|t|$ denotes the set underlying the translation of the type t. In the formulation presented here, the above propositions follows immediately from the definition of $R_{\leq}$.

In this section we have given an encoding of DCC in the Calculus of Constructions and used it to prove noninterference in a simple fashion. Expressing proofs by Parametricity inside the language being studied enabled clear definitions and a proof devoid of any inductive argument.

6 DEEP EMBEDDING OF DEPENDENCY CORE CALCULUS

All our results so far concern a shallow embedding of DCC into CC. While many will be satisfied with a shallow embedding, others (including some anonymous reviewers of a previous version of this paper) may rightly wonder if these results carry over to the usual presentation of DCC, which uses an inductively-defined family of well-typed terms (also sometimes called a deep embedding).

In this section we address this question by the affirmative. We show that our shallow embedding is faithful to the deep embedding. Consequently all results about the evaluation of shallow-embedded terms carry over to the evaluation of deep-embedded terms. As a corollary, non-interference carries over. The situation is summarized schematically in Fig. 1.

Since DCC is mostly standard, with the exception of the typing rules for `return` and `bind` presented in the previous section, we focus directly on the encoding of the traditional DCC (of Abadi et al. [1999]) in our shallow embedding of label-polymorphic DCC in CC. The encoding is structured as a usual denotational semantics. For every DCC type t, we give a CC type $|t|$, assuming that we have a $dcc : \mathtt{DCC}$ implementation of the `DCC` module in our context. The translation of types is straightforward:

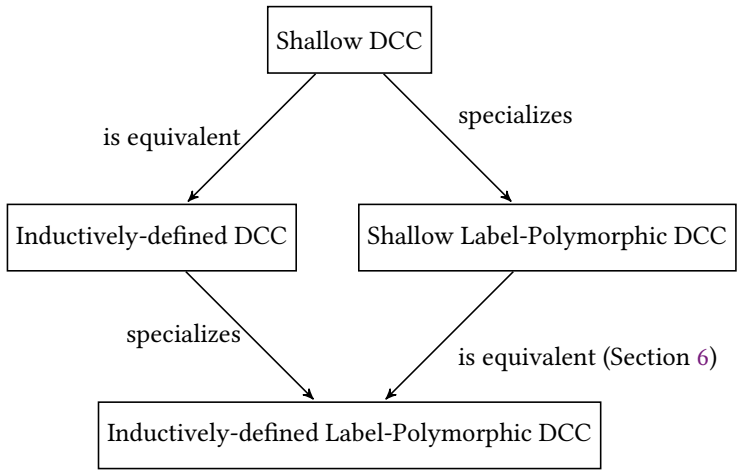

Fig. 1. Overview of calculi of interest. The bulk of the paper is concerned with non-interference for the Shallow Label-Polymorphic DCC. We additionally prove the equivalence of that result with non-interference for the inductively-defined Label-Polymorphic DCC in 6. We automatically get the corresponding results for DCC, because the Label-Polymorphic DCC is a conservative extension of DCC.

$$
\begin{aligned}
|unit| &= \top \\
|t_0 \times t_1| &= |t_0| \times |t_1| \\
|t_0 + t_1| &= |t_0| + |t_1| \\
|t_0 \longrightarrow t_1| &= |t_0| \longrightarrow |t_1| \\
|T\ \ell\ t| &= (T\ dcc)\ \ell\ |t|
\end{aligned}
$$

Where $T\ dcc : \ell \longrightarrow \star \longrightarrow \star$ denotes the selector which picks out the first item (the implementation of the T type) from the $dcc : \texttt{DCC}$ product. As before we assume products and co-products encoded in the usual fashion and a lattice $\mathcal{L}$ common to DCC and our encoding. The interpretation extends to typing contexts: $|x : t, \Gamma| = x : |t|, |\Gamma|$. Note that while CC has a notion of telescopes, or dependent contexts, this generality is only be exploited to a very narrow extent, because all source contexts are simply typed.

For every well-typed DCC term e we give an interpretation in CC, $[\![e]\!]$. We do not write the translation in full here (supplementary material can be consulted for any detail), but rather give a pedagogical account which ignores some of the tedium of working in pure CC. We also take the liberty to present this in terms of the `bind` primitive rather than `map` and `join` primitives of Section 5 to keep the exposition clean of the clutter introduced by the different presentations.

Definition 4 (Interpretation).

$$
\begin{aligned}
[\![\lambda x.\ e[x]]\!]\ \gamma &= \lambda x.\ [\![e]\!](x, \gamma) \\
[\![e\ e_1]\!]\ \gamma &= [\![e]\!]\ \gamma\ ([\![e_1]\!]\ \gamma) \\
[\![(e_1,\ e_2)]\!]\ \gamma &= ([\![e_1]\!]\ \gamma, [\![e_2]\!]\ \gamma) \\
[\![\pi_i\ e]\!]\ \gamma &= \pi_i\ ([\![e]\!]\ \gamma) \\
[\![\iota_i\ e]\!]\ \gamma &= \iota_i\ ([\![e]\!]\ \gamma) \\
[\![\mathsf{case}\ e\ e_1\ e_2]\!]\ \gamma &= \mathsf{case}\ ([\![e]\!]\ \gamma)\ ([\![e_1]\!]\ \gamma)\ ([\![e_2]\!]\ \gamma) \\
[\![\mathsf{return}_\ell\ e]\!]\ \gamma &= \mathsf{return}\ \ell\ ([\![e]\!]\ \gamma) \\
[\![\mathsf{bind}\ p\ e\ e_1]\!]\ \gamma &= \mathsf{bind}\ (\textit{protect-type}\ p)\ ([\![e]\!]\ \gamma)\ (\lambda x.\ [\![e_1]\!]\ (x, \gamma))
\end{aligned}
$$

Most of the constructions are mapped simply to their CC counterparts [3]. The main idea is that the free variables (denoted by name on the left hand side) correspond to positions in the environment γ. Note however that on the left-hand-side, `bind` refers to the DCC bind, while on the right-hand-side it points to the function in the the `dcc : DCC` module, defined in the previous section. For our purposes, the most interesting case in the definition above is that of `bind`. Recall the typing rule for `bind` from before:

$$\frac{\Gamma \vdash e : T\ \ell\ A \quad \Gamma, x : A \vdash e' : B \quad \ell \preceq B}{\Gamma \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : B}$$

An auxiliary translation *protect-type* is needed to convert the precondition $\ell \preceq B$ into a proof of $\ell \preceq_{\mathsf{dcc}} |B|$. We do not give the implementation of *protect-type* here, noting that it is a simple case-by-case translation of the rules for the $\preceq$ relation into the $\mathtt{protect}_x$ functions from the `dcc : DCC` module in the previous section. In the translation above we refer to this proof as p on the left-hand-side and the translation as *protect-type* p on the right. Constructing this proof is done as one would expect by induction on the derivation of $\ell \preceq B$. Note also that the bound variable x gives rise to an abstraction in CC in $(\lambda x.\ [\![e_1]\!]\ \gamma)$

Having established a translation from DCC to CC we can turn to the issue of correctness. The first important theorem about the translation is that the translation is type-preserving.

Theorem 5 (Type-preservation). *If* $\Gamma \vdash e : t$, *then* $\mathsf{dcc} : \mathtt{DCC} \vdash [\![e]\!] : |G| \to |t|$.

Proof. By induction on the derivation $\Gamma \vdash e : t$. □

We can then turn our attention to the behaviour of evaluation.

The we adopt the usual small-step evaluation relation for DCC, written ($\longrightarrow$). Its reflexive and transitive closure is written ($\longrightarrow^*$). This relation is strongly normalizing. On the CC side, we rely on the standard evaluation relation. But, because the proof is formalised *within* CC itself, all β-equivalent CC terms are definitionally equal (by the relation ($\equiv$)). We can then show that the translation preserves equality on terms. For induction to work, we must however generalise to open terms, as follows:

Theorem 6. *Assume* $\Gamma \vdash e_1 : t$ *and* $\Gamma \vdash e_2 : t$. *If* $e_1 \longrightarrow e_2$ *then for every environment* $\gamma : |\Gamma|$ *we have* $[\![e_1]\!](\gamma) \equiv [\![e_2]\!](\gamma)$.

The proof is mostly standard: one proves the usual substitution lemmas; then the proof can proceed by case analysis for the reduction relation. One particular aspect deserves mention however. One might assume that we require the specific instance of our `DCC` module, defined above, for the proof to go through. However this is not the case: the proof is almost entirely independent of the

[3] In the original paper, $\mathtt{return}_\ell$ in DCC is written η_ℓ. We use `return` here in the interest of consistency and ease of reading

implementation of the DCC module. There just is one exception; we need to be able to prove the following proposition:

$$\texttt{bind dcc}\ \ell\ A\ B\ p\ (\texttt{return dcc}\ \ell\ a)\ f \equiv f\ a$$

This condition turns out to be nothing more than the the right-unit monadic law for bind and return found in the literature [Moggi 1991], generalised to take into account security levels. It is necessary to discharge the case for reduction of a term on the form $\texttt{bind}\ x = \texttt{return}_\ell\ v\ \texttt{in}\ e'$. In this light this precondition should come as no surprise given both the semantics of bind in DCC and our idea of T as a family of monads.

Corollary 7. *For any closed DCC terms a and b such that $\vdash a : T_\ell(Bool)$ and $\vdash b : T_\ell(Bool)$, if $[\![a]\!] \equiv [\![b]\!]$, then $a \equiv_\beta b$.*

Proof. By case analysis. By strong normalisation of DCC, we either have $a \longrightarrow^* \texttt{return}_\ell\ \texttt{true}$ or $a \longrightarrow^* \texttt{return}_\ell\ \texttt{false}$, and likewise for b. Assume $a \longrightarrow^* \texttt{return}_\ell\ \texttt{false}$ without loss of generality. If $b \longrightarrow^* \texttt{return}_\ell\ \texttt{false}$ we have our result, by confluence. Assume on the contrary $b \longrightarrow^* \texttt{return}_\ell\ \texttt{true}$. Then by the above theorem $[\![a]\!] \equiv \texttt{return}_\ell\ \texttt{false}$ and $[\![b]\!] \equiv \texttt{return}_\ell \texttt{true}$, which is contradictory with the assumption that $[\![a]\!] \equiv [\![b]\!]$. □

Theorem 8 (Noninterference, for original DCC). *Given A a DCC type, and $\ell : \mathcal{L}$ such that $\ell \not\sqsubseteq \hat{\ell}$ for any DCC term e, such that $\vdash e : T_\ell(A) \longrightarrow T_{\hat{\ell}}(\texttt{Bool})$, and any two other closed terms $a_0, a_1 : A$, we have $e(\texttt{return}_\ell\ a_0) \equiv_\beta e(\texttt{return}_\ell\ a_1)$.*

Proof. By applying Theorem 3 (Noninterference for the shallow embedding of DCC) on $[\![e]\!]$, we get that, for any CC values a_0' and a_1', $[\![e]\!]\ a_0' \equiv [\![e]\!]\ a_1'$. By letting $a_0' = [\![a_0]\!]()$ and $a_1' = [\![a_1]\!]()$, we have $[\![e]\!]([\![a_0]\!]()) \equiv [\![e]\!]([\![a_1]\!]())$. By the above corollary, we get $e[\texttt{return}_\ell\ a_0] \equiv_\beta e[\texttt{return}_\ell\ a_1]$. The desired result is obtained by one step of beta-expansion on both sides. □

7 IMPLEMENTATION IN HASKELL

Because our development is completely formalised in Agda, and Agda can be used as a programming language, one could simply make use our *dcc* module as a security library for Agda.

However, Agda is not very commonly used as a programming language. Therefore in this section we show how our theoretical development can be adapted as a Haskell library.

```
module DCC (T, eta, mu, Protected, protect_T, protect_T', protect_x, protect_fun) where

data T (f :: lattice -> lattice -> *) (l :: lattice) (a :: *) = T a

eta :: a -> T f l a
eta = T

mapT  :: T f l a -> (a -> b) -> T f l b
mapT (T a) f = T (f a)

joinT :: Protected f l a -> T f l a -> a
joinT Proof (T a) = a
```

We make use of the Haskell module structure to hide the `T` constructor, making pattern matching on values of type `T f l a` impossible. The `f :: lattice -> lattice -> *` argument in the definition of `T` is an addition to the theoretical development. It serves as a way to make the code parametric in the choice of the $\sqsubseteq$ relation, a value of type `f l l'` is a proof that $l \sqsubseteq l'$. The `Protected f l a` argument to `joinT` mirrors the $\ell \preceq A$ argument from the theory. In the theoretical development we

defined `Protected f l a` as `⊤`, or `()` in Haskell. We can do the same here, selecting the following implementation for `Protected`:

```
data Protected :: (l -> l -> *) -> l -> * -> * where
  Proof :: Protected f l a
```

Alongside which we add functions for constructing the proofs:

```
protectT :: NFData (f l l') => f l l' -> Protected f l (T f l' a)
protectT f = f `deepseq` Proof

protectT' :: Protected f l a -> Protected f l (T f l' a)
protectT' Proof = Proof

protectx :: Protected f l a -> Protected f l b -> Protected f l (a, b)
protectx Proof Proof = Proof

protectfun :: Protected f l b -> Protected f l (a -> b)
protectfun Proof = Proof
```

These functions are precisely the ones found in the definition of the *dcc* : DCC module in Section 5. The `NFData (f l l')` constraint in the definition of `protectT` and the use of `deepseq`, which simply fully evaluates its first argument before returning the second, are there to make sure that the `f l l'` proof is fully evaluated before the function returns. This alongside the explicit pattern-matches on `Proof` constructors ensures this code is strict in all its arguments. This is important to ensure similar soundness for the Haskell code as the theory. Were this not the case we would be able to write code that forges evidence of the proof-carrying objects. Code which does forge evidence will diverge at runtime, for this reason along with Haskell's usual partiality we are only able to provide termination insensitive noninterference. Information can still be leaked via the termination side-channel. The code requires some extensions to enable Haskell to act more like a dependently typed language. The extensions are `GADTs`, `PolyKinds`, `DataKinds`, and `KindSignatures`.

To see how to use this code with a specific lattice we give the necessary definitions for the canonical two point lattice with elements H and L where $H \not\sqsubseteq L$ is the only disallowed flow. We represent the lattice as a datatype `data TwoPoint = H | L`. The ($\sqsubseteq$) relation is represented as a simple proof carrying GADT:

```
data TPFlow :: TwoPoint -> TwoPoint -> * where
  Up   :: TPFlow L H
  Same :: TPFlow l l
```

We also need to give an instance of `NFData TPFlow`:

```
instance NFData (TPFlow l l') where
  rnf Up   = ()
  rnf Same = ()
```

Finally we can specialise the `T` family to the two-point lattice `type T_TwoPoint = T TPFlow`.

8 RELATED WORK

The connection between parametricity and noninterference has a rich history. Abadi et al. [1999] introduced the *Dependency Core Calculus* (DCC), a unifying framework for dependency tracking in program calculi. Tse and Zdancewic [2004] attempted to give a translation of DCC into System F. However, a counterexample to Tse and Zdancewic's key lemma was found by Shikuma and Igarashi [2008]. Bowman and Ahmed [2015] attempted to fix the issue by giving a full-abstraction result for DCC and System F_ω. While this result is very impressive, the proof technique used, involving open logical relations and back-translation of F_ω terms into DCC, hides the simplicity

of the underlying connection between noninterference and parametricity. Bowman and Ahmed's choice of translation scheme for T, a CPS conversion, further complicates the reading of their result. In this paper we have shown how to give a more straightforward account of noninterference as a consequence of parametricity by lifting the DCC primitives into a PTS in a way very reminiscent of how one would naively implement DCC in a functional language. None of the previous proofs of noninterference from parametricity have, to the best of our knowledge, been mechanised in a proof assistant, a clear contribution of our work. The core proof is surprisingly small, coming in at under 200 lines of Agda, evidence that our technique does indeed yield simple proofs.

As opposed to Bowman and Ahmed we prove noninterference for a polyvariant, i.e. *label-polymorphic*, version of DCC, embedded in the Calculus of Constructions. As is, our result does not subsume theirs however. One key contribution made by Bowman and Ahmed is the translation from DCC to System F_ω and *back again*.

Aguirre et al. [2017] give a translation of DCC into a higher-order relational calculus. Their proof uses a logical relations argument with a logical relation similar to ours. Instead of giving a direct definition of $R_\preceq$ to show that values of protected types are always equivalent this result is given as a lemma in their system, similarly to Abadi et al..

Algehed [2018] recently proposed an alternative presentation of DCC which he calls SDCC (for Simplified DCC). The calculus does away with the $\preceq$ relation in favour of a number of arguably simpler primitives, most notably $\mathsf{up} : (A : \star) \to (\ell, \ell' : \mathcal{L}) \to (\ell \sqsubseteq \ell') \to T\ \ell\ A \to T\ \ell'\ A$ and $\mathtt{com} : (A : \star) \to (\ell, \ell' : \mathcal{L}) \to T\ \ell\ (T\ \ell'\ A) \to T\ \ell'\ (T\ \ell\ A)$. Similarly, Algehed and Russo [2017] point out that $\ell \preceq A$ corresponds to a function $T\ \ell\ A \to A$. While these presentations can both easily fit into the parametricity framework (the reader is encouraged to attempt this as an exercise), we have chosen to stick with the traditional presentation of DCC because we believe that that our definition of $R_\preceq$ can serve as an instructive example of how to apply the parametricity theory.

We have implemented our DCC module in Haskell as a security library. Typical Noninterference proofs for security libraries use a very different technique from the ones presented in this paper. Vassena and others have made extensive use of a technique called erasure to conduct their proofs for libraries like MAC [Russo 2015], LIO [Stefan et al. 2011], and Sec [Russo et al. 2008]. Erasure establishes a simulation between a semantics with all secrets replaced by a dummy construct. While the technicalities of this technique and ours differ, both are based on the same core intuition. One advantage of the erasure technique is its applicability to non-terminating or partial programs in proofs of Termination Sensitive Noninterference and Progress Sensitive Noninterference [Vassena and Russo 2016]. We are unsure how well parametricity extends to this setting. However Wadler [1989] remarks that weaker versions of Parametricity still hold in a variant of System F with non-termination. This leads us to believe that at least some of our results carry over to this setting. We leave further investigation of the topic as future work.

9 CONCLUSION AND FUTURE WORK

In this paper we have provided the first mechnized proof of noninterference based on parametricity. This is made possible by the simplicity and expressivity of parametricity for dependent types. We find that this proof structure clearly separates the essential parts (how to interpret security types) from the mechanical inductive work (handled for free by parametricity). We find our logical interpretation to be sufficiently elegant to serve as a (formal) introductory semantics of information-flow security. We would go so far as recommending this proof structure for all non-interference proofs where the technique applies.

This approach is best-suited for shallow embedding of security languages. In fact we have provided the first such security library with a proof of noninterference based entirely on parametricity.

Such a proof is easily amenable to mechanisation: the noninterference proof for our DCC library is less than 200 lines of Agda.

Yet, if one insists on a classic presentation of a security language as an inductively defined family of well-typed expressions (also known as a deep-embedding), we can transport our non-interference proofs to such a setting. This can be done using an interpretation from a deep embedding to the shallow one.

One potential shortcoming of the technique presented in this paper is with languages containing arbitrary side-effects and non-termination. How parametricity for dependent types extends to effectful calculi has yet to be thoroughly explored. A possible avenue is to encode such effects, using a standard monadic setting. However, Abadi et al. [Abadi et al. 1999] and Algehed and Russo [Algehed and Russo 2017] show how to encode a large number of what are normally considered impure side-effects in DCC. These encodings are done using only pure functions, and so are entirely applicable in our setting.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful and insightful comments. This work was supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 147–160.

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 21.

Maximilian Algehed. 2018. A Perspective on the Dependency Core Calculus. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security (PLAS '18)*. ACM, New York, NY, USA, 24–28. https://doi.org/10.1145/3264820.3264823

Maximilian Algehed and Alejandro Russo. 2017. Encoding DCC in Haskell. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. ACM, 77–89.

Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda calculus with types*. Cambridge University Press.

Jean-philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (2012), 107–152.

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

William J Bowman and Amal Ahmed. 2015. Noninterference for free. *ACM SIGPLAN Notices* 50, 9 (2015), 101–113.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 289–301.

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523.

Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and computation* 76, 2-3 (1988), 95–120.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

John C Reynolds. 1983. Types, abstraction and parametric polymorphism. (1983).

Alejandro Russo. 2015. Functional pearl: Two can keep a secret, if one of them uses Haskell. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 280–288.

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A library for light-weight information-flow security in Haskell. In *ACM Sigplan Notices*, Vol. 44. ACM, 13–24.

Naokata Shikuma and Atsushi Igarashi. 2008. Proving Noninterference by a Fully Complete Translation to the Simply Typed lambda-calculus. *CoRR* abs/0808.3307 (2008). arXiv:0808.3307 http://arxiv.org/abs/0808.3307

Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *ACM Sigplan Notices*, Vol. 46. ACM, 95–106.

Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 115–125.

Marco Vassena and Alejandro Russo. 2016. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 15–28.

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. MAC A verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming* 95 (2018), 148 – 180. https://doi.org/10.1016/j.jlamp.2017.12.003

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. ACM, 347–359.