

Dynamic IFC Theorems for Free!

Maximilian Alghed
Chalmers
Göteborg, Sweden
alghed@chalmers.se

Jean-Philippe Bernardy
University of Gothenburg
Göteborg, Sweden
jean-philippe.bernardy@gu.se

Cătălin Hrițcu
MPI-SP
Bochum, Germany
catalin.hritcu@gmail.com

Abstract—We show that noninterference and transparency, the key soundness theorems for dynamic IFC libraries, can be obtained “for free”, as direct consequences of the more general parametricity theorem of type abstraction. This allows us to give very short soundness proofs for dynamic IFC libraries such as faceted values and LIO. Our proofs stay short even when fully mechanized in the Agda proof assistant. Moreover, our proofs cover the actual Agda implementations of the libraries, not just an abstract model.

1 Introduction

The goal of information flow control (IFC) research is to develop language-based techniques to ensure that security policies relating to confidentiality and integrity of data are followed, by construction. This paper is about a recent incarnation of this idea: *IFC as a library*. This appealing approach, pioneered by Li and Zdancewic [1] and championed by Russo et al. [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] among others [13, 14, 15], promises to ease the integration of IFC techniques into existing software development pipelines, by replacing the specialized languages, compilers, and run-time systems traditionally needed for IFC applications, with libraries providing similar guarantees. Practically speaking, programming with an IFC library is similar to programming with a specialized IFC language, with one exception: Rather than being stand-alone, the library integrates with and uses the features of its host language to provide an interface that guarantees that all client programs are secure.

These libraries enforce IFC using two key language features:

Controlling side effects: most IFC libraries are implemented in safe [16] Haskell [17], a language that allows the library author to enforce type-based control over side effects (with the exception of non-termination).

Abstraction: all IFC libraries rely on type abstraction to provide data confidentiality and integrity.

But embedding IFC as a library risks leaving a gap in the soundness proofs, which usually do not cover how the library interacts with the host language. Indeed, the typical soundness proofs work by constructing a *model* of the library as a more-or-less standard IFC calculus, for which the authors prove some variant of the noninterference security property [18]. So although these noninterference proofs are sometimes mechanically verified [7, 10, 11, 13, 14], they give no guarantees about the realization of the calculus *as a library*.

The formal connection between the calculus and the library, which relies on host language features such as type abstraction,

was so far almost never investigated. This can have important security consequences. For instance, Stefan et al. [19] claimed that their concurrent LIO Haskell library for dynamic IFC protects against so-called “internal timing attacks” (attacks where the program itself can observe and exploit secret-dependent timing information). However, Buiras and Russo [20] later demonstrated that this was not the case: Haskell’s lazy evaluation permits reliably bypassing the mechanisms of Stefan et al., enabling reliable internal timing attacks [20, 21]. And while we do not yet deal with concurrency and timing attacks, we are the first to provide formal guarantees covering *the actual code of dynamic IFC libraries*, such as LIO. The key to scaling security proofs to concrete implementations of dynamic IFC libraries is employing better proof techniques.

In this paper we provide simpler proofs for the implementations of two different kinds of dynamic IFC libraries. On the one hand, we study the original (sequential) LIO library [3], in which individual values can be labeled with metadata specifying their confidentiality and integrity levels and computations carry a “current label” that soundly over-approximates the level of already inspected labeled values. On the other hand, we study a library based on *faceted values* [7, 22], which are decision trees that can evaluate to different values based on the privilege level of the observer. These two styles reflect the most common ways to enforce dynamic IFC as a library.

To prove noninterference in a simple way, we give semantics to the libraries in terms of *logical relations*. Every type T induces a relation $\llbracket T \rrbracket$, such that every well-typed program $p : T$ is related to itself with respect to the binary relation induced by T . This connection between terms, types, and logical relations is called the *fundamental lemma* of logical relations, or the *abstraction theorem* [23], or *parametricity*. In his seminal work, Reynolds [23] uses this technique to show that users of an abstract type can never observe the details of its implementation. In this paper we apply this idea to dynamic IFC, showing that noninterference for dynamic IFC libraries is a direct consequence of the same parametricity theorem.¹

In practice, we implement the dynamic IFC libraries in a language with dependent types. This allows us to program our libraries and their clients in the same formalism we use to reason about the security of such programs. We also use

¹Theorems directly obtained from parametricity are often called *free theorems* [24], thus the title of this paper.

the fact that (dependent) types are turned mechanically to logical relations and programs are turned mechanically to proofs of satisfaction of such relations, and everything remains neatly expressible within the framework of a language with dependent types [25]. This way, we do not need to prove any fundamental lemma for our logical relations, and additionally the junction between implementation and theory is watertight.

This proof technique was recently applied by Algehed and Bernardy [26] to show the noninterference of *static* IFC libraries. The work in this paper differs in two crucial ways:

- 1) We show that the same technique can be applied to prove noninterference for *dynamic* IFC libraries. While this might seem counterintuitive at first, since parametricity is a property of type abstraction, which is a static enforcement mechanism, our work shows formally that dynamic IFC libraries ultimately achieve their security also from type abstraction.
- 2) We show that the same proof technique can be used to prove more than just noninterference. Specifically, we use the parametricity theorem to also prove that our faceted values library is *transparent* [7, 27], ensuring that programs that are already secure do not have their semantics altered by our library.

More importantly, we inherit the simplicity of the parametricity-based proof technique. In particular, our *completely mechanized proof of noninterference for an actual implementation* in Agda [28] of the state-of-the-art LIO library is *an order of magnitude shorter* than the previous partial Coq proof for a model of the same library [10]. Moreover, the simplicity of our proof technique allows us to cover *more* of our implementation of the LIO library than the previous Coq proof. Specifically, our mechanized LIO proof also covers the non-trivial mechanisms that enable soundly recovering from IFC and user-thrown exceptions [10, 29].

Concretely, we make the following **contributions**:

- We give Agda implementations of two core dynamic IFC libraries: LIO [3, 10] (Section 4.2) and a faceted values library inspired by the Faceted [22] and Multef [7] Haskell libraries (Section 2).
- We use the parametricity theorem (Section 3) for these dynamic IFC libraries to provide simple noninterference proofs for any client of each library (Section 4).
- We use the same proof technique to also show that our faceted values library satisfies transparency [7, 27], which ensures that this library does not alter the semantics of already secure programs. (Section 5).
- Our core libraries can be soundly extended to side effects without additional proof obligations; we illustrate this by adding mutable state to our core faceted values library (Section 6). The key idea is simple [3, 6]: the noninterference of any client of our core libraries, also holds for any client *library*. In particular, a client library implementing side effects using only the primitives exposed by the core library automatically enjoys noninterference.
- Finally, all our proofs have been fully mechanized in the

```
module MultefImplementation where
data Fac : Set → Set where
  return : {A : Set} → A → Fac A
  facet   : {A : Set} → Label → Fac A → Fac A
           → Fac A

bind : {A B : Set} → Fac A → (A → Fac B) → Fac B
bind (return a) c = c a
bind (facet ℓ f0 f1) c = facet ℓ (bind f0 c) (bind f1 c)
```

Fig. 1. Faceted values part of Multef in Agda

Agda proof assistant, and are available as supplementary material for this paper.² As a consequence of the use of parametricity for dependent types, the implementation of our libraries and the proofs are surprisingly short: our whole formal development for the two different libraries is less than one thousand lines of Agda.

While we use Agda as our language of choice in this paper, we expect that our results are easily to generalize to other languages with strong abstraction mechanisms and dependent types, like Coq and F*, provided the requisite parametricity theory can be established, which is already the case for Coq [25].

2 Dynamic IFC as a Library

Before turning our attention to embedding dynamic IFC as a library, we give a quick primer on our host language, Agda: a total functional programming language with dependent types [28].

One characteristic of Agda is that terms, types, and propositions are unified. Thus, a single dependent arrow type-former (\rightarrow) can be used to define function types and quantifications. For example the type $(x : \text{Bool}) \rightarrow \text{Bool}$ is a function type whose domain and co-domain are Booleans. The type $(x : \text{Bool}) \rightarrow x \equiv \text{true} \vee x \equiv \text{false}$ is an example of a quantified proposition over booleans. The type $(A : \text{Set}) \rightarrow \text{List } A \rightarrow \text{List } A$ is that of a polymorphic list-transformation. This last example illustrates quantification over types, which is seen as a dependent function whose domain is the type of types, written Set in Agda. (To avoid logical inconsistencies sets of sets are organized in a tower, such that $\text{Set}_i : \text{Set}_{i+1}$, and Set is a shorthand for Set_0 .)

As a convenience feature, Agda offers *implicit arguments*. If a function domain is written with braces, for example $f : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \text{List } A$, then when calling the function f one will omit the corresponding argument, for example $f \text{ someList} : \text{List } A$. In this situation, Agda will try to infer the omitted argument from the context. For example if $\text{someList} : \text{List } \text{Bool}$ then $A = \text{Bool}$. If doing so is impossible or ambiguous, Agda will report an error.

While Agda has many more features, we will only use a small time-tested subset. We also make use of records, which

²Supplementary material available at <https://github.com/MaximilianAlgehed/DynamicIFCTheoremsForFree>

will be discussed below, and some limited use of data types, that work like in other functional languages.

Our first library is based on *faceted values* [22, 27] and is displayed in Figure 1. This is a straightforward port of the faceted values part of the Multef Haskell library [7] to Agda. Faceted values are binary decision trees that can evaluate to different values based on the privilege level of the observer. Formally, a faceted value $f : \text{Fac } A$ can either be a regular value $v : A$ that does not depend on who is observing it, in which case $f = \text{return } v$, or it can depend on who is observing it, in which case it is a tree node containing a label $\ell : \text{Label}$ (we assume a base-type `Label` of security labels explained below) and children f_0 and f_1 of type $\text{Fac } A$ (i.e., $f = \text{facet } \ell \ f_0 \ f_1$). If an observer has access to level ℓ they will see f_0 , otherwise f_1 . For example, the faceted value `facet Alice (return 0) (return 1) : Fac Int` looks like 0 to anyone who is able to observe values with Alice’s confidentiality level, and looks like 1 to everyone else.

Combining two faceted values also yields a faceted value. For example, if we add $f_0 = \text{facet Alice (return 0) (return 1)}$ and $f_1 = \text{facet Bob (return 2) (return 3)}$, we get the *recursive* faceted value:

```
f0+f1 = facet Alice
        (facet Bob (return (0 + 2)) (return (0 + 3)))
        (facet Bob (return (1 + 2)) (return (1 + 3)))
```

By inspecting this value, we see that if the observer can observe *both* Alice’s and Bob’s data, they observe the sum $f_0 + f_1$ as 0 + 2, whereas an observer who can see only Bob’s data, *not* Alice’s, sees the sum as 1 + 2.

To formally define this addition operation on faceted values we follow the literature [7, 22] and show that faceted values form a *monad* [30, 31], which provides a general computational tool we can use to easily define operations like addition. A monad is a triple of a type former, $M : \text{Set} \rightarrow \text{Set}$ (here $M = \text{Fac}$), and two operations, $\text{return} : \{A : \text{Set}\} \rightarrow A \rightarrow M A$ that takes a pure value and embeds it into a monadic value, and $\text{bind} : \{A B : \text{Set}\} \rightarrow M A \rightarrow (A \rightarrow M B) \rightarrow M B$ that takes a “computation” $m : M a$ and a continuation $c : A \rightarrow M B$ and produces a computation $\text{bind } m \ c : M B$.³ Using the monad operations, we can define addition and similar operations for Fac easily in the following manner:

```
(+) : Fac Int → Fac Int → Fac Int
fx + fy = bind fx \ x →
          bind fy \ y →
          return (x + y)
```

The syntactic form $\lambda x \rightarrow \dots$ in the code above is simply Agda notation for lambda abstraction. Agda provides us with a syntactic convenience for monadic computations in the form of so-called *do*-notation. The code above can be written as:

```
(+) : Fac Int → Fac Int → Fac Int
fx + fy = do
```

³A monad requires `bind` and `return` to satisfy certain *algebraic laws*. However, these are orthogonal to our development and thus omitted.

```
desugar      =
do x ← m     | bind m \ x →
  c          | desugar (do c)

desugar      =
do m         | bind m \ _ →
  c          | desugar (do c)

desugar      =
do let x = e | let x = e in
  m         | desugar (do m)
```

Fig. 2. Rules for desugaring Agda’s *do*-notation

```
x ← fx
y ← fy
return (x + y)
```

Here, the syntax $x \leftarrow e$ corresponds to the use of `bind`, following the desugaring rules listed in Figure 2. We will use this *do*-notation in the rest of the paper.

As illustrated by the addition example above, the `bind` operation from Figure 1 works by traversing faceted values in its operands and constructing a recursive faceted value, taking into account all possible observers.

Following the IFC literature [32], whether a level can be observed from another level is given by a partial order $\sqsubseteq : \text{Label} \rightarrow \text{Label} \rightarrow \text{Bool}$. This order is also required to form a join semi-lattice, and thus it has a least element $\ell_\perp : \text{Label}$ and the least-upper-bound always exists and is given by the function $\sqcup : \text{Label} \rightarrow \text{Label} \rightarrow \text{Label}$. Using this lattice structure, we can define `project 1 f`, intuitively what a value of type $f : \text{Fac } a$ will “look like” to an observer at level 1. We define `project 1 f` by recursion on f :

```
project : {A : Set} → Label → Fac A → A
project ℓ (return a)      = a
project ℓ (facet ℓ' f0 f1) =
  if ℓ' ⊆ ℓ then
    project ℓ f0
  else
    project ℓ f1
```

In turn, from the definition of `project`, we can define what it means for a program, for example a function $p : \text{Fac Int} \rightarrow \text{Fac Int}$, to be *secure*. The program p is *non-interfering* if given any label $\ell : \text{Label}$ and two faceted values $f_0, f_1 : \text{Fac Int}$ such that $\text{project } \ell \ f_0 \equiv \text{project } \ell \ f_1$ we have that $\text{project } \ell \ (p \ f_0) \equiv \text{project } \ell \ (p \ f_1)$. In other words, p does not reveal information from a different security level to an observer at level ℓ .

To see an example of this property in action, consider the `Temp client` module in Figure 3. When we give `isCold` the arguments f_0 and f_1 defined as:

```
f0 f1 : Fac Int
f0 = facet Alice (return 10) (return 0)
f1 = facet Alice (return 30) (return 0)
```

we get:

```
isCold f0 = facet Alice (return Cold) (return Cold)
isCold f1 = facet Alice (return Hot) (return Cold)
```

```

module Temp where

data HotOrCold : Set where
  Hot  : HotOrCold
  Cold : HotOrCold

isCold : Fac Int → Fac HotOrCold
isCold fint = do
  x <- fint
  if x > 25 then
    return Hot
  else
    return Cold

```

Fig. 3. A client of the Multef library

```

isAlicePos : Fac Int → Fac Bool
isAlicePos (facet Alice (return n) f) = return (n > 0)
isAlicePos f = return False

```

Fig. 4. An illbehaved client

If the observer level ℓ is Bob, who cannot see Alice's data (i.e., $\text{Bob} \not\sqsubseteq \text{Alice}$) we have that $\text{project } f_0 \text{ Bob} \equiv \text{project } f_1 \text{ Bob}$ and also $\text{project (isCold } f_0) \text{ Bob} \equiv \text{project (isCold } f_1) \text{ Bob}$.

The goal of library-based IFC is to ensure that all client code behaves securely, as `isCold` does. However, suppose that we could write the code in Figure 4. Function `isAlicePos` uses pattern matching to check whether its faceted input is precisely of the form `facet Alice (return n) f`, for some n and f , and if so returns a raw (un)faceted value `return (n > 0)`. Otherwise `isAlicePos` returns `return False`. The function `isAlicePos` clearly breaks noninterference. What goes wrong here is that even though we carefully ensure that the functions `facet`, `return`, and `bind` respect noninterference, the `isAlicePos` client function could just side-step our interface and break up faceted values to look directly at Alice's secrets.

To ensure security, therefore, we must use the abstraction mechanisms of the underlying programming language [33]. In the case of Agda, this means two things. First, we must define the *interface* of the IFC library, as a record type. In the case of core Multef, the interface can be found in Figure 5.

Second, any client can depend *only* on this interface. For Agda, this is ensured by parameterizing the client modules by the IFC library interface, for example, the following will make the `BadClient` module parametric:

```

record MultefInterface where
  field
    Fac : Set → Set
    return : {A : Set} → A → Fac A
    facet : {A : Set} → Label → Fac A → Fac A
           → Fac A
    bind : {A B : Set} → Fac A → (A → Fac B)
           → Fac B

```

Fig. 5. The abstract interface to the Multef library

```

module BadClient (imp : MultefInterface) where
open MultefInterface imp

isAlicePos : Fac Int → Fac Bool
...

```

Then Agda will not allow us compile this module because of the `isAlicePos` function: we get an error telling us that `Fac` is an abstract type, and does not expose a constructor facet on which we can pattern-match (in the interface, `facet` is just a *function* and not a datatype constructor). Finally, we must make sure that our secure implementation, which we will call `sec`, indeed implements the interface. In Agda syntax:

```

sec : MultefInterface
sec = record {MultefImplementation}

```

In Section 4, we will use this abstraction barrier to provide a *relational interpretation* for the Multef interface (in addition to that of the LIO library [3]) and a proof that the implementation satisfies this relational interpretation. By the abstraction theorem, all client modules will be proved to be noninterfering.

3 Parametricity and Data Abstraction

Before we dive into proving noninterference for our libraries, we give a quick primer on constructive reasoning in Agda. Using the propositions-as-types (also known as the Curry-Howard) correspondence [34, 35], we can interpret Agda types as propositions and programs as proofs in constructive logic:

The type of types is `Set` and inhabitants of this type can also be seen as propositions. Type $\top : \text{Set}$ is inhabited by a trivial `tt : \top` inhabitant, and thus represents Truth as a proposition:

```

data  $\top$  : Set where tt :  $\top$ 

```

Conversely $\perp : \text{Set}$ is not inhabited by any term and thus represents Falsity. Conjunction $_ \wedge _ : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$ and disjunction $_ \vee _ : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$ are implemented as product and sum types, respectively. Finally, as also explained in the previous section, quantification and implication correspond to the dependent function type $(x : A) \rightarrow B$.

Since types are propositions, their inhabitants are proofs. Concretely, if $P : \text{Set}$ is a proposition (type) then $t : P$ is a proof (inhabitant) of the proposition (type). For example, the canonical proof of proposition $A \wedge B \rightarrow B \wedge A$ is the following function:

```

swap : A  $\wedge$  B → B  $\wedge$  A
swap (a , b) = (b , a)

```

With this background in place, we now introduce parametricity and data abstraction for dependently typed languages. This proof technique is based on logical relations [36, 37], which are an elegant tool to prove properties about programming languages, and in particular IFC. The key idea is that one interprets every type as a relation. For every type A , one builds a relation $\llbracket A \rrbracket$ — thus $\llbracket A \rrbracket a_0 a_1$ is a proposition given two values $a_0 a_1 : A$ and so $\llbracket A \rrbracket : A \rightarrow A \rightarrow \text{Set}$. One then proves the fundamental lemma of logical relations, also known as abstraction theorem, or parametricity theorem:

Proposition 1 (Parametricity). *If $\tau : A$ then $\llbracket A \rrbracket \tau \tau$. That is, every program τ of type A satisfies the relational interpretation of its type $\llbracket A \rrbracket$.*

One often uses a custom logical relation [38], but there is a general, most fundamental way to interpret dependent types as relations, given by Bernardy et al. [25], which we adapt below for the syntax of Agda types.⁴

Definition 1. (Relational interpretation of types) This meta-level definition works by induction on the structure of types.

$$\begin{aligned}
\llbracket \text{Set}_i \rrbracket A_0 A_1 &= A_0 \rightarrow A_1 \rightarrow \text{Set}_i \\
\llbracket (x:A) \rightarrow B \rrbracket f_0 f_1 &= (x_0 : A_0) \\
&\quad \rightarrow (x_1 : A_1) \\
&\quad \rightarrow (x_r : \llbracket A \rrbracket x_0 x_1) \\
&\quad \rightarrow \llbracket B \rrbracket (f_0 x_0) (f_1 x_1) \\
\llbracket \text{record field } f^i : A_i \rrbracket r_0 r_1 &= \text{record field} \\
&\quad f_r^i : \llbracket A_i \rrbracket (f^i r_0) (f^i r_1) \\
\llbracket B \rrbracket b_0 b_1 &= b_0 \equiv b_1
\end{aligned}$$

As mentioned above, the type of types (Set) is interpreted as a function from two types to Set (i.e., a relation). A function type is interpreted as a relation requiring that inhabitants map related arguments to related results. A record type is interpreted as a relation relating two instances of the record by relating all their fields. Finally, base-types (B), like booleans, are interpreted as propositional equality.

To prove the fundamental lemma, one proceeds by giving a relational interpretation $\llbracket t \rrbracket$ for every term t : For our development, this interpretation is as follows.

Definition 2. Relational interpretation of terms

$$\begin{aligned}
\llbracket t \ u \rrbracket &= \llbracket t \rrbracket u_0 u_1 \llbracket u \rrbracket \\
\llbracket \lambda x \rightarrow t \rrbracket &= \lambda x_0 \rightarrow \lambda x_1 \rightarrow \lambda x_r \rightarrow \llbracket t \rrbracket \\
\llbracket \text{record } \{f^i = t_i\} \rrbracket &= \text{record } \{f_r^i = \llbracket t_i \rrbracket\} \\
\llbracket f^i \ t \rrbracket &= f_r^i \llbracket t \rrbracket \\
\llbracket x \rrbracket &= x_r \\
\llbracket b \rrbracket &= \text{refl} \\
\llbracket A \rrbracket &= \lambda a_0 \rightarrow \lambda a_1 \rightarrow \llbracket A \rrbracket a_0 a_1
\end{aligned}$$

This interpretation mimics the behavior of the relational interpretation of types. In particular, if the term is a base type constant (the penultimate case) then the proof of relatedness is simply reflexivity of equality, and if the term is a type (the last case), we construct an explicit relation and fall back to the interpretation for types. Thus, the interpretation of types as relation and the interpretation of terms as proofs can be unified, hence the use of a single notation $\llbracket \cdot \rrbracket$ for both purposes. We refer the reader to Bernardy et al. [25] for details.

This relational interpretation of terms and types provides an once and for all proof of the parametricity theorem:

⁴Their theory is for pure type systems with inductive families, covering all the features of Agda that we use in this paper.

Theorem 1. (Parametricity [25])

If $\tau : A$ then $\llbracket \tau \rrbracket : \llbracket A \rrbracket \tau \tau$.

As an illustration, we show how to use Theorem 1 to prove properties about an abstract module and its clients. Consider the following (restricted) interface for Booleans:

```

record Booleans where
  field
    Bool    : Set
    true    : Bool
    false   : Bool
    ^       : Bool → Bool → Bool

```

The above declares an abstract type, `Bool`, two constants `true` and `false` of type `Bool`, and a binary operation `^` over `Bool`. We instantiate this interface in the standard way:

```

module Impl where
  data Bool : Set where
    true  : Bool
    false : Bool
    ^     : Bool → Bool → Bool
    ^ false _ = false
    ^ _ b    = b

```

```

booleans : Booleans
booleans = record {Impl}

```

What may be surprising about this `Booleans` interface is that when we use it, we always end up writing monotonic functions. More precisely, if we write a function $o : (\text{imp} : \text{Booleans}) \rightarrow \text{Bool} \text{imp} \rightarrow \text{Bool} \text{imp}$, it is possible to prove, using parametricity, that if $b_0, b_1 : \text{Impl.Bool}$ are such that b_0 implies b_1 then $o \text{ booleans } b_0$ implies $o \text{ booleans } b_1$. Intuitively, this is because the `Booleans` interface gives the o function no way to do negation. This means that all $o \text{ imp}$ can do to its $b : \text{Bool imp}$ argument is to either discard it and return some other boolean or take its conjunction with some other boolean. These other booleans are either constants, or obtained by calling functions, which are themselves parametric in `imp`. However, “by induction”, these functions are also monotonic and so o is monotonic.

Making this “by induction” phrase precise demands an argument based on logical relations. In Agda we can let Theorem 1 do the ground work, making the proof feel nearly automatic. To see how this works formally, we need to understand two things. Firstly, the *standard* relational interpretation of the `Booleans` interface ($\llbracket \text{Booleans} \rrbracket$, obtained mechanically by using the meta-level function from Definition 1), tells us how to relate two implementations of `Booleans`:⁵

```

record  $\llbracket \text{Booleans} \rrbracket$  (m0 m1 : Booleans) : Set1 where
  field
    Boolr : Bool m0 → Bool m1 → Set
    truer : Boolr (true m0) (true m1)
    falser : Boolr (false m0) (false m1)
    ^r : ∀ a0 a1 → Boolr a0 a1 →
          ∀ b0 b1 → Boolr b0 b1 →
          Boolr (^ m0 a0 b0) (^ m1 a1 b1)

```

⁵Accessing a record field is done by treating the field name as a function with the analyzed record as an additional first argument.

This relation contains a *custom* logical relation (Bool_r), such that each method in the interface respects this relation (and true_r , false_r , \wedge_r are proofs witnessing this).

Secondly, because o is parameterized by $\text{imp} : \text{Booleans}$, we have that $\llbracket \text{o} \rrbracket$ is parameterized over $\llbracket \text{Booleans} \rrbracket$:

```
 $\llbracket \text{o} \rrbracket$  : (imp0 imp1 : Booleans)
  → (impr :  $\llbracket \text{Booleans} \rrbracket$ ) imp0 imp1)
  → (b0 : Bool imp0) (b1 : Bool imp1)
  → (br : Boolr impr b0 b1)
  → Boolr impr (o imp0 b0) (o imp1 b1)
```

Now all it takes to prove our theorem is to realize that:

- 1) We care about two `Bool` booleans, so we have to take $\text{imp}_0 = \text{imp}_1 = \text{booleans}$, and
- 2) $\text{imp}_r : \llbracket \text{Booleans} \rrbracket$ `booleans booleans` is an argument to $\llbracket \text{o} \rrbracket$ that we get to pick, and
- 3) the final thing we want to prove is that if $b_0 \Rightarrow b_1$, then $\text{o} \dots b_0 \Rightarrow \text{o} \dots b_1$, so we want that $\text{Bool}_r \text{ b}_0 \text{ b}_1 = b_0 \Rightarrow b_1$.

With insight (2) and (3) and a definition of \Rightarrow as a relation on `Bool`:

```
 $\_ \Rightarrow \_$  : Bool → Bool → Set
true  ⇒ false = ⊥
_      ⇒ _    = ⊤
```

we can define the fields of booleans_r as follows:

```
booleansr :  $\llbracket \text{Booleans} \rrbracket$  booleans booleans
booleansr = record {
  Boolr    =  $\_ \Rightarrow \_$ 
; truer    = tt
; falser   = tt
;  $\wedge_r$  true true ar b0 b1 br = br
;  $\wedge_r$  false a1 ar b0 b1 br = tt
}
```

The proofs that `true` and `false` satisfy the relation are trivial. For \wedge , we proceed by a simple case analysis. This gives us all the pieces we need to construct our monotonicity proof:

```
 $\llbracket \text{o} \rrbracket$  booleans booleans booleansr :
  (b0 : Bool booleans) → (b1 : Bool booleans)
  → (br : b0 ⇒ b1) → o booleans b0 ⇒ o booleans b1
```

Now, suppose we add negation to the interface of `Booleans`, with the usual implementation in `booleans`:

```
record Booleans : Set1 where
  field
  ...
  neg : Bool → Bool
```

```
module Impl where
  ...
  neg : Bool → Bool
  neg true = false
  neg false = true
```

```
booleans : Booleans
booleans = record {Impl}
```

If we try to prove the same monotonicity theorem, which now shouldn't hold, we run into issues when we try to provide neg_r in the new booleans_r . Specifically, trying to fulfill the proof obligation by case analysis leaves us with an impossible goal:

```
negr : (a0 a1 : Bool) → a0 ⇒ a1
      → neg booleans a0 ⇒ neg booleans a1
negr true true tt = tt
negr false true tt = ? -- Goal is true ⇒ false
negr false false tt = tt
```

As demonstrated by this example, parametricity for dependent types is not just an adequate tool for reasoning about metatheoretic properties of libraries, it's also compositional. To prove that a library guarantees some property, one simply defines the necessary relations on one's types and proves that each operation preserves these relations. In Section 4 we use this technique to show noninterference for two security libraries. The proofs work like the simple proof above: we define the relations necessary to prove noninterference and show that each operation in the library respects the relations.

4 Two Proofs of Noninterference

In this section we use the parametricity technique outlined above to prove noninterference for two dynamic IFC libraries. The first proof (Section 4.1) is for the faceted values part of the `Multef Haskell` library, which we have already introduced in Section 2. The second proof is for the significantly more complex `LIO` library (Section 4.2).

4.1 Noninterference for Faceted Values

Recall the faceted values interface in Figure 5 from Section 2. It exports a type `Fac` for faceted values and operations `facet`, `return`, and `bind` for manipulating them. The goal in this section is to show that any client library of this abstract interface obeys noninterference.

Intuitively, this means that any client function of `MultefImplementation` needs to take ℓ -equivalent inputs to ℓ -equivalent outputs. In order to make this meaningful we recall the `sec` instantiation of `MultefInterface` and the definition of `project` from Section 2 and provide the following definition of ℓ -equivalence: If A is a base-type and $f_0, f_1 : \text{Fac } A$ we say that f_0 and f_1 are ℓ -equivalent, written $f_0 \sim \langle \ell \rangle f_1$, when:

$$f_0 \sim \langle \ell \rangle f_1 = \text{project } f_0 \ell \equiv \text{project } f_1 \ell$$

Where \equiv is propositional equality.

In order to prove noninterference, we need to prove that, for a given base-type A (say `Bool`), $\text{o} : \text{Fac } A \rightarrow \text{Fac } A$ and if $f_0 \sim \langle \ell \rangle f_1$ then $\text{o } f_0 \sim \langle \ell \rangle \text{o } f_1$. However, this only holds if o is a function in a *client* of the `Multef` library (more accurately, its abstract interface). In other words, noninterference only needs to hold for a function $\text{o} : (\text{m} : \text{MultefInterface}) \rightarrow \text{Fac } \text{m } A \rightarrow \text{Fac } \text{m } A$.

Recall from the `Booleans` example in Section 3 that we can reason about o by providing a suitable $\text{sec}_r : \llbracket \text{Multef} \rrbracket \text{ sec sec}$, a *proof* that `sec` (from Section 2):

```
sec : MultefInterface
sec = record {MultefImplementation}
```

satisfies the relational interpretation of its type. Formally, parametricity requires us to construct a $f_r : \llbracket A \rrbracket f f$ for each function $f : A$ in the `MultefInterface` record type.

```

Facr : (A0 A1 : Set)
  → (Ar : [[Set]] A0 A1)
  → [[Set]] (Fac sec A0) (Fac sec A1)

facetr : (A0 A1 : Set) → (Ar : [[Set]] A0 A1)
  → (ℓ0 ℓ1 : Label) → (ℓ : [[Label]] ℓ0 ℓ1)
  → (f00 : Fac sec A0) → (f01 : Fac sec A1)
  → (f0r : Facr A0 A1 Ar f00 f01)
  → (f10 : Fac sec A0) → (f11 : Fac sec A1)
  → (f1r : Facr A0 A1 Ar f10 f11)
  → Facr A0 A1 Ar (facet sec ℓ0 f00 f10)
    (facet sec ℓ1 f01 f11)

returnr : (A0 A1 : Set) → (Ar : [[Set]] A0 A1)
  → (a0 : A0) → (a1 : A1) → (ar : Ar a0 a1)
  → Facr A0 A1 Ar (return sec a0) (return sec a1)

bindr : (A0 A1 : Set) → (Ar : [[Set]] A0 A1)
  → (B0 B1 : Set) → (Br : [[Set]] B0 B1)
  → (f0 : Fac sec A0) → (f1 : Fac sec A1)
  → (fr : Facr A0 A1 Ar f0 f1)
  → (c0 : A0 → Fac sec B0) → (c1 : A1 → Fac sec B1)
  → (cr : (a0 : A0) → (a1 : A1) → (ar : Ar a0 a1)
    → Facr B0 B1 Br (c0 a0) (c1 a1))
  → Facr B0 B1 Br (bind sec f0 c0) (bind sec f1 c1)

```

Fig. 6. Fields in `[[MultefInterface]] sec sec`

Concretely, this means that to construct `secr`, we need to construct Agda terms inhabiting the four types in Figure 6.

Picking our implementation of `secr : [[MultefInterface]] sec sec` that we will use in our noninterference proof is straightforward given our formulation of the $f_0 \sim \langle \ell \rangle f_1$ relation above. Given ℓ^* as the attacker-level that we are concerned about, we pick:

`Facr A0 A1 Ar f0 f1 = Ar (project f0 ℓ*) (project f1 ℓ*)`

Note that if A is a *base type*, then $[[A]] = _ \equiv _$ and so `Facr A A [[A]]` corresponds to $\sim \langle \ell^* \rangle$. The definitions of `facetr`, `returnr`, and `bindr` are easy to fill out, and can be looked up in the Agda mechanization.⁶

Theorem 2 (Noninterference for Faceted Execution). *Given:*

`o : (m : MultefInterface) → Fac m Bool → Fac m Bool`

We know that for all $f_0, f_1 : \text{Fac sec Bool}$ such that there exists a term:

`assume : f0 $\sim \langle \ell^* \rangle$ f1`

We show that:

`o sec f0 $\sim \langle \ell^* \rangle$ o sec f1`

Proof. From Theorem 1, we know that the following holds:

```

[[o]] : (m0 m1 : MultefInterface)
  → (mr : [[MultefInterface]] m0 m1)
  → (f0 : Fac m0 Bool) → (f1 : Fac m1 Bool)
  → (fr : Facr mr Bool Bool [[Bool]] f0 f1)
  → Facr mr Bool Bool [[Bool]] (o m0 f0) (o m1 f1)

```

We have chosen `secr : [[MultefInterface]] sec sec` so that:

`Facr secr Bool Bool [[Bool]] f0 f1 = f0 $\sim \langle \ell^* \rangle$ f1`

This means that:

`[[o]] sec sec f0 f1 assume : o sec f0 $\sim \langle \ell^* \rangle$ o sec f1`

And is thus a valid Agda proof term for our theorem. \square

⁶It may be helpful to recall that `Label` is a base-type, so the `[[Label]] ℓ0 ℓ1` argument to `facetr` is equivalent to $\ell_0 \equiv \ell_1$.

```

record LIOInterface : Set1 where
  field
    -- Labeled Values
    Labeled : Set → Set
    label    : {A : Set} → Label → A → Labeled A
    labelOf  : {A : Set} → Labeled A → Label
    -- LIO Computations
    LIO      : Set → Set
    return   : {A : Set} → A → LIO A
    bind     : {A B : Set} → LIO A
      → (A → LIO B) → LIO B
    unlabel  : {A : Set} → Labeled A → LIO A
    toLabeled : {A : Set} → Label
      → LIO A → LIO (Labeled A)
    -- Exceptions
    throw    : {A : Set} → UserError → LIO A
    catch    : {A : Set} → LIO A
      → (E → LIO A) → LIO A

sec : LIOInterface
sec = { Labeled A = (A  $\uplus$  E)  $\times$  Label
      ; LIO A = (ℓc : Label) →  $\Sigma$  ((A  $\uplus$  E)  $\times$  Label)
        (λ r → ℓc  $\sqsubseteq$   $\pi_2$  r)
      ; ... }

```

Fig. 7. Our Agda port of the core LIO library

4.2 Noninterference for Core LIO

Next we turn our attention to LIO. We first explain our Agda port of LIO and then how we use parametricity to prove noninterference for it. Our port covers both the original LIO library [3] and a more recent extension to allow recovering from user-defined and IFC exceptions [10], but leaves out state (in Section 6 we give a simple way state could be added for free on top of this core library) and clearance (a concern not directly related to IFC that could also be easily added). The interface of the core library can be seen in Figure 7, but most of the implementation has been elided for space. Instead on focusing on the code, we give the intuition behind this library.

The interface first defines `Labeled A`, the abstract type of labeled values of type A . If v is a value of type A and ℓ is an IFC label, then we can use the `label ℓ v` operation to classify value v at level ℓ , which results in a labeled value of type `Labeled A`. Once we have a `Labeled` value we can use `labelOf`

to obtain its label, which witnesses the fact that, in LIO, the label on data is public information. Unlike the label, the value of a $lv : \text{Labeled } A$ is not public, but protected precisely by $\text{labelOf } lv$. This means that it would not be secure to simply extract the value of lv , so the LIO interface provides no such operation. Instead, to work with labeled values in a way that ensures IFC we need to turn to *labeled LIO computations*.

An LIO computation keeps track of a *current label*, which is the upper bound of all labeled values already inspected by the computation. LIO threads through the current label, and in our case we also produce an explicit proof that the current label can only increase in the IFC lattice, or stay the same. This (monotonic) state passing makes LIO a monad, with `return` and `bind` operations having analogous type signatures to those for faceted execution. In addition, the LIO monad provides an `unlabel` operation that soundly returns the value inside a labeled value lv by increasing the current label by $\text{labelOf } lv$:

```
unlabel : {A : Set} → Labeled A → LIO A
```

This allows LIO computations to process labeled data, while using the current label to track both explicit and implicit information flows (i.e., flows through the control flow of the program [32]). Once we are done computing based on labeled values we can label the result and restore the current label to what it was at the beginning of the current computation. To prevent leaking information via the label of the final result, this label has to be chosen in advance before inspecting any labeled data. This functionality is implemented by the following operation:

```
toLabeled : {A : Set} → Label
           → LIO A → LIO (Labeled A)
```

The expression `toLabeled ℓ lio` runs the `lio` computation and if at the end the current label is below ℓ the result is labeled ℓ and the current label restored. On the other hand, if at the end the current label is not below ℓ we have to signal an IFC error. The original LIO [3] treated such errors as fatal and stopped execution, however a more recent extension of LIO [10] makes IFC errors recoverable. In the case of a wrongly annotated `toLabeled` though, throwing an exception would not be sound: we can restore the current label at the end only if that is a control-flow join point. To preserve this property, LIO returns instead a *delayed exception* [29], which is another kind of labeled value, labeled with the originally chosen level ℓ . When unlabeling it the delayed exception is re-thrown, which is sound because, as explained above, unlabeling a value raises the current label.

In addition to re-throwing delayed exceptions on `unlabel`, LIO provides standard primitives to throw user exceptions and to catch arbitrary ones. In order to achieve soundness, though, LIO also has to delay any such exceptions at the end of `toLabeled`. To make debugging easier, all exceptions carry information such as the current label at the time the exception was originally thrown together with a stack trace. In addition, IFC exceptions record additional information about the involved labels, when this information can be securely revealed (e.g., when the label check fails for `toLabeled ℓ it`

is secure to reveal the label ℓ , but not the current label that is not below ℓ).

With this intuition in place, we can look at the actual definitions of the `Labeled` and `LIO` types. The definition of `Labeled` is straightforward: a labeled value is a pair of a `Label` and either a result of type A , or a delayed exception of type E , $\text{Labeled } A = (A \uplus E) \times \text{Label}$, where \uplus denotes the tagged sum, or “disjunctive union”:

```
data _ $\uplus$ _ : Set → Set → Set where
inj1 : {A B : Set} → A → A  $\uplus$  B
inj2 : {A B : Set} → B → A  $\uplus$  B
```

The definition of `LIO` meanwhile, is more involved. A term of type `LIO A` is a function that takes a current label $\ell c : \text{Label}$ and produces a result that we call a *configuration*. A configuration is an output label together with either a result of type A or a delayed exception of type E (that’s either a user exception or an IFC exception whose details are omitted here). However, as noted above we make the LIO computation also produce a proof that the output label is at least as restrictive as the input label ℓc . This addition is opaque to the programmer, who programs against the abstract interface of LIO, but is useful for simplifying our proofs, since it prevents the logical relation below from being cluttered with this monotonicity proof. Consequently, the definition of LIO is as follows:

```
-- Configuration
Cfg A = (A  $\uplus$  E)  $\times$  Label
-- LIO computation
LIO A = ( $\ell c : \text{Label}$ ) -- Input label
       →  $\Sigma$  (Cfg A) -- Result
       ( $\lambda r \rightarrow \ell c \sqsubseteq \pi_2 r$ ) -- Useful proof
```

This definition uses a generalized sum type, or Σ -type, to connect the proof of monotonic labels to the computation. The Σ type-former is given by the following record type:

```
record  $\Sigma$  (A : Set) (B : A → Set) : Set where
field  $\pi_1$  : A
       $\pi_2$  : B  $\pi_1$ 
```

and we additionally have the notation (a, b) for record $\{\pi_1=a, \pi_2=b\}$. Finally, to avoid confusion we note that the `_ , _`-syntax is shared between Σ -types and simple product types $(A \times B)$, indeed the latter is an instantiation of the former: $A \times B = \Sigma A (\lambda _ \rightarrow B)$.

With this background in place, we turn to stating noninterference for LIO. We do this for any client function that is parametric in the LIO interface, takes a labeled boolean input, and performs a LIO computation returning a boolean as result:

```
o : (m : LIOInterface) → Labeled m Bool → LIO m Bool
```

To state noninterference for `o sec` we need to define ℓ^* -equivalence for labeled values and LIO computations.

We say that two labeled values $lv_0, lv_1 : \text{Labeled Bool}$ are indistinguishable at an observer label ℓ^* , written $lv_0 \sim(\ell^*) lv_1$, if and only if:

- 1) $\text{labelOf } lv_0 \equiv \text{labelOf } lv_1$ and
- 2) if $\text{labelOf } lv_0 \sqsubseteq \ell^*$ then $lv_0 \equiv lv_1$

Point 1 says that the labels of lv_0 and lv_1 may never diverge from each other, since they are public information. Point 2 says

that if lv_0 (and therefore also lv_1) is a public level, then lv_0 and lv_1 have to be equal.

Next we turn our attention to the relation for Bool-returning LIO computations. Firstly, recall that c_0 and c_1 are state-passing computations where the state is the current label: $\text{Label} \rightarrow \Sigma ((\text{Bool} + \text{E}) \times \text{Label}) \dots$. Because the current label is not necessarily public, but instead it protects itself, the standard way to define ℓ^* -equivalence for the current label is the following:

$$lc_0 \sim \langle \ell^* \rangle lc_1 = (lc_0 \sqsubseteq \ell^* \vee lc_1 \sqsubseteq \ell^*) \rightarrow lc_0 \equiv lc_1$$

Two current labels are ℓ^* -equivalent if whenever one of them is observable at level ℓ^* the other label is the same. We extend this to final configurations of type $(\text{Bool} + \text{E}) \times \text{Label}$:

$$(r_0, lc_0) \sim \langle \ell^* \rangle (r_1, lc_1) = (lc_0 \sim \langle \ell^* \rangle lc_1) \wedge (lc_0 \sqsubseteq \ell^* \wedge lc_1 \sqsubseteq \ell^* \rightarrow r_0 \equiv r_1)$$

For two configurations to be ℓ^* -equivalent, we require that the current labels are ℓ^* -equivalent and if they are public then the results of the computation should also be equal. With this in place, it is easy to define ℓ^* -equivalence for LIO computations:

$$c_0 \sim \langle \ell^* \rangle c_1 = (lc_0 \text{ } lc_1 : \text{Label}) \rightarrow lc_0 \sim \langle \ell^* \rangle lc_1 \rightarrow \pi_1 (c_0 \text{ } lc_0) \sim \langle \ell^* \rangle \pi_1 (c_1 \text{ } lc_1)$$

This requires that for any ℓ^* -equivalent initial current labels we obtain ℓ^* -equivalent final configurations (the π_1 projections are needed to ignore the proof part of the LIO type).

With these definitions in place we can now state our noninterference theorem:

Theorem 3 (Noninterference). *Given:*

$$o : (m : \text{LIOInterface}) \rightarrow \text{Labeled } m \text{ Bool} \rightarrow \text{LIO } m \text{ Bool}$$

For all $lv_0, lv_1 : \text{Labeled sec Bool}$ we assume that:

$$lv_0 \sim \langle \ell^* \rangle lv_1$$

We show that:

$$o \text{ sec } lv_0 \sim \langle \ell^* \rangle o \text{ sec } lv_1$$

The general strategy of the proof is the same as for Multef. In particular we use Theorem 1 to obtain:

$$\begin{aligned} [\![o]\!] &: (m_0 \ m_1 : \text{LIOInterface}) \\ &\rightarrow (m_r : [\![\text{LIOInterface}]\!] \ m_0 \ m_1) \\ &\rightarrow (l_0 : \text{Labeled } m_0 \text{ Bool}) \rightarrow (l_1 : \text{Labeled } m_1 \text{ Bool}) \\ &\rightarrow (l_r : \text{Labeled}_r \ m_r \ \text{Bool} \ \text{Bool} \ [\![\text{Bool}]\!] \ l_0 \ l_1) \\ &\rightarrow \text{LIO}_r \ m_r \ \text{Bool} \ \text{Bool} \ [\![\text{Bool}]\!] \ (o \ m_0 \ l_0) \ (o \ m_1 \ l_1) \end{aligned}$$

To use this result we first need to pick two relations $\text{Labeled}_r : [\![\text{Set} \rightarrow \text{Set}]\!] \ \text{Labeled} \ \text{Labeled}$ and $\text{LIO}_r : [\![\text{Set} \rightarrow \text{Set}]\!] \ \text{LIO} \ \text{LIO}$ and prove that relatedness at these relations is respected by the LIO operations. Moreover, to be helpful for proving noninterference these relations have to specialize (e.g., for Bool) to the ℓ^* -equivalence instances from in the noninterference statement above.

For labeled values, we use a straightforward generalization of the ℓ^* -equivalence definition above:

$$\begin{aligned} \text{Labeled}_r \ A_0 \ A_1 \ A_r \ lv_0 \ lv_1 &= \\ &(\text{labelOf } m_0 \ lv_0 \equiv \text{labelOf } m_1 \ lv_1) \wedge \\ &(\text{labelOf } m_0 \ lv_0 \sqsubseteq \ell^* \rightarrow (A_r \ [\![\sqcup]\!] \ E_r) \ (\pi_1 \ lv_0) \ (\pi_1 \ lv_1)) \end{aligned}$$

(Where the relation $A_r \ [\![\sqcup]\!] \ B_r$ relates two values if they are either inj_1 and related by A_r , or inj_2 and related by B_r .) In the second conjunct, we require that if the level of the two labeled values is public then either they are both errors related at E_r , or they both carry values of type A that are related at A_r . The relation $E_r : E \rightarrow E \rightarrow \text{Set}$ relates two delayed exceptions if and only if they are the same. We do a similar generalization from Bool to arbitrary types for defining LIO_r .

The main part of our Agda proof of Theorem 3 was showing that the LIO operations respect the Labeled_r and LIO_r relations. Some of the LIO operations have straightforward proofs (label, return, labelOf, unlabel, and throw), while the higher-order operations (bind, toLabeled, and catch), have slightly more interesting proofs that rely on the monotonicity of the current label. Fortunately, all these proofs are pleasantly short adding up to under 300 lines of Agda for the complete noninterference proof in our supplementary material.

The short and fully mechanized Agda proof we describe above can be contrasted with the previous partially mechanized proof for LIO [10]. This previous proof shows noninterference for an abstract calculus *without* exception handling and state, covering a strict subset of the features of the library implementation that we have verified here. Their proof technique, to show a simulation between evaluation of an LIO term with secrets and the same term with the secrets erased, is standard but cumbersome. Consequently, their proof amounts to over 3000 lines of Coq, even if it is not fully mechanized and it only covers a small calculus, not a concrete library implementation. Their proof could probably be finished and made shorter by using more tactic automation or a different proof strategy [29], yet it seems hard to match the conceptual simplicity and compactness of our parametricity-based proof.

5 Transparency

One of the primary justifications for faceted semantics is the so-called *transparency* theorem [27]. In short, transparency states that: for any program p that is noninterfering under a non-faceted, “standard” semantics, the behavior of p is preserved when p is run with faceted semantics. Intuitively, this means that there are no false alarms with faceted execution: if the program is noninterfering to begin with, facets don’t change anything. This is unlike systems like LIO, where false alarms are a problem that the programmer has to work around by adhering to proper programming style.

In our setting, this transparency property can be reformulated in terms of one of the key lemmas used to prove both noninterference and transparency for traditional faceted calculi: *faceted evaluation simulates standard evaluation* [27, 39].

To make sense of what this means in our context we need to explain the distinction between faceted and standard evaluation. Luckily, it is straightforward for us to define what we mean by different semantics for the same program: we simply give different implementations of the `MultefInterface`! In particular, the faceted semantics was already defined as the `MultefImplementation` module in Figure 1 from Section 2.

```

Maybe : Set → Set
Maybe A = A ⊔ ⊤

just : {A : Set} → A → Maybe A
just a = inj1 a

nothing : {A : Set} → Maybe A
nothing = inj2 tt

[[Maybe]] : (A0 A1 : Set) → (Ar : A0 → A1 → Set)
           → Maybe A0 → Maybe A1 → Set
[[Maybe]] _ _ Ar = Ar [[⊔]] [[⊤]]

```

Fig. 8. The Maybe type former and its relational interpretation.

```

module Std where
  Fac : Set → Set
  Fac a = Maybe a

  facet : {A : Set} → L → Fac A
         → Fac A → Fac A
  facet ℓ f0 f1 = nothing

  return : {A : Set} → A → Fac A
  return a = just a

  bind : {A B : Set} → Fac A
        → (A → Fac B) → Fac B
  bind f c = case f of \
    { inj1 a → c a
    ; inj2 tt → inj2 tt }

  std : MultefInterface
  std = record {Std}

```

Fig. 9. Standard Semantics of Multef

To define the standard semantics, we first introduce the Maybe (also known as “option”) type former, as a special case of the \uplus type, in Figure 8. With this in place, we define the standard semantics of Multef in Figure 9. Most of the definitions are unsurprising: $\text{Fac } A$ is $\text{Maybe } A$, while return and bind are standard for the Maybe monad. The only potentially surprising definition is $\text{facet } \ell f_0 f_1 = \text{nothing}$. To understand it, note that the standard phrasing of the transparency property in the literature makes reference to *facet-free* programs [7, 27, 39]. In our setting, we cannot easily talk about such “facet-free” programs, because all programs we study are clients of the MultefInterface , so instead we make do by talking about programs that do not return nothing under evaluation in the standard semantics.

Theorem 4 (Transparency). *Fix a label $\ell^* : \text{Label}$. Given any $b : \text{Bool}$, define the faceted value f^b as having value b for observers that can see data labeled ℓ^* , and value false otherwise as:*

$$f^b = \text{facet } \text{sec } \text{Bool } \ell^* (\text{return } \text{sec } b) \\ (\text{return } \text{sec } \text{false})$$

For any client function o , parametric in MultefInterface :

$$o : (m : \text{MultefInterface}) \rightarrow \text{Fac } m \text{ Bool} \rightarrow \text{Fac } m \text{ Bool},$$

which does not crash under the standard semantics (std) when given the non-faceted constant b as input:

$$o \text{ std } (\text{just } b) \neq \text{nothing},$$

then running $o \text{ std } (\text{just } b)$ yields the same result from the point of view of an observer at level ℓ^* as running o with the faceted semantics (sec) on input f^b :

$$o \text{ std } (\text{just } b) \equiv \text{just } \text{Bool } (o \text{ sec } f^b \ell^*)$$

To prove Theorem 4 we need to relate the execution of $o \text{ std}$ with the execution of $o \text{ sec}$. According to our setup, Theorem 1 gives us that if $o : (m : \text{MultefInterface}) \rightarrow T$ for some type T , then:

$$[[o]] : (m_0 m_1 : \text{MultefInterface}) \\ \rightarrow (m_r : [[\text{MultefInterface}]] m_0 m_1) \\ \rightarrow [[T]] (o m_0) (o m_1)$$

In particular, if we can provide some $\text{std-sec}_r : [[\text{MultefInterface}]] \text{std sec}$, then parametricity lets us relate $o \text{ std}$ and $o \text{ sec}$.

As we have seen with the previous proofs in this paper, the key to getting this to work is picking the correct instantiation of Fac_r in std-sec_r . In this case the choice is clear from the theorem that we are trying to prove. We want that, when the standard (non-faceted) result is not nothing , the projection at ℓ^* of the value resulting from the faceted execution is related to the standard value. In other words, we pick the following definition of Fac_r in std-sec_r :

$$\text{Fac}_r \text{ std-sec}_r = \lambda A_0 A_1 A_r f_0 f_1 \rightarrow f_0 \neq \text{nothing} \rightarrow \\ [[\text{Maybe}]] A_0 A_1 A_r f_0 (\text{just } (\text{project } f_1 \ell^*))$$

From this definition, filling out facet_r , return_r , and bind_r is straightforward. With the definition of std-sec_r in place, we can tackle the proof of Theorem 4.

Proof of Theorem 4. From:

$$o : (m : \text{MultefInterface}) \rightarrow \text{Fac } m \text{ Bool} \rightarrow \text{Fac } m \text{ Bool}$$

We obtain by parametricity:

$$[[o]] : (m_0 m_1 : \text{MultefInterface}) \\ \rightarrow (m_r : [[\text{MultefInterface}]] m_0 m_1) \\ \rightarrow (f_0 : \text{Fac } m_0 \text{ Bool}) \rightarrow (f_1 : \text{Fac } m_1 \text{ Bool}) \\ \rightarrow (f_r : \text{Fac}_r m_r \text{ Bool Bool } [[\text{Bool}]] f_0 f_1) \\ \rightarrow \text{Fac}_r m_r \text{ Bool Bool } [[\text{Bool}]] (o m_0 f_0) (o m_1 f_1)$$

We pick $m_0 = \text{std}$, $m_1 = \text{sec}$, $m_r = \text{std-sec}_r$, $f_0 = \text{just } b$, $f_1 = f^b$, and let f_r be some (omitted) proof that f_0 and f_1 are appropriately related⁷

$$f_r : \text{Fac}_r \text{ std-sec}_r \text{ Bool Bool } [[\text{Bool}]] f_0 f_1,$$

whose type becomes this after unfolding definitions:

$$f_r : \text{just } b \neq \text{nothing} \rightarrow \\ [[\text{Maybe}]] \text{Bool Bool } ___ (\text{just } b) \\ (\text{just } (\text{project } f^b \ell^*)).$$

This gives us the following instantiation for $[[o]]$:

$$[[o]] \text{ std sec std-sec}_r (\text{just } b) f^b f_r : \\ o \text{ std } (\text{just } b) \neq \text{nothing} \rightarrow$$

⁷The reader will find its definition in the Agda mechanization of this paper

```

module FacState (imp : MultefInterface) where
  open MultefInterface imp
  -- FSt Computations
  State : Set
  State = String → Bool

  FSt : Set → Set
  FSt = State → Fac (State × A)

  -- Monadic Fragment
  return : {A : Set} → A → FSt A
  return a = λ s → return (s , a)

  bind : {A B : Set} → FSt A
        → (A → FSt B) → FSt B
  bind f c = λ s → do
    (s' , a) ← f
    c a s'

  lift : {A : Set} → Fac A → FSt A
  lift f = λ s → do
    a ← f
    return (s , a)

  -- State Monad
  get : FSt State
  get = λ s → return (s , s)

  put : State → FSt ⊤
  put s' = λ s → return (s' , tt)

  -- Derived Operations
  read : String → FSt Bool
  read var = do
    s ← get
    return (s var)

  write : String → Bool → FSt ⊤
  write var val = do
    s ← get
    put (λ var' → if var==var' then
      val
    else
      s var')

```

Fig. 10. FacState implementation on top of Multef

$\llbracket \text{Maybe} \rrbracket \text{ Bool Bool } _ \equiv _ \text{ (o std (just b)) } \text{ (just (project (o sec f^b) \ell^*))}.$

This is sufficient to easily establish the theorem. \square

The key takeaway from this proof is the generality of parametricity as a proof technique. While previous proofs have focused on the connection between noninterference and parametricity [26, 40, 41], the proof above shows that parametricity can also be useful for proving other interesting meta-theoretical properties about security libraries.

6 Supporting Effects

The libraries that we have investigated so far only offer dynamic IFC functionality for programs without side effects. However, following Alghed and Russo [6], we can use

the old trick of *Monad Transformers* [42] to extend our current libraries with monadic side effects. In short, a monad transformer is something that can take a monad and add extra functionality to it, while retaining the monadic structure. Formally, a monad transformer is a function:

$$T : (\text{Set} \rightarrow \text{Set}) \rightarrow (\text{Set} \rightarrow \text{Set})$$

Such that if $M : \text{Set} \rightarrow \text{Set}$ is a monad, then so is $T M$. Furthermore, $T M$ must retain the structure of M , and so there is required to be a function:

$$\text{lift} : \{A : \text{Set}\} \rightarrow M A \rightarrow T M A$$

with the role of lifting M -computations to $T M$ computations.

The quintessential example of a monad transformer is `StateT`. This monad transformer makes a monad stateful, by threading through a state of type `State` using the following transformation:

$$\text{StateT } M A = \text{State} \rightarrow M (\text{State} \times A).$$

In other words, a `StateT M A` computation is a computation that starts with an initial state and performs some actions in M and produces a new state and a result A .

Figure 10 shows the special case of how we can instantiate the `StateT` monad transformer with the `Fac` monad to create the `FSt` monad (for “faceted state”). The definitions of `return`, `bind`, and `lift` are straightforward. By abuse of notation, we allow the names `return` and `bind`, and so also the use of `do`-notation, to overlap between the M and `StateT M` monads.

On top of these mundane monadic operations, `StateT` also provides functions for accessing and updating the state in a computation, `get` and `put`. Furthermore, with the specific instantiation of the `State` type in the `FacState` module, mapping strings to booleans, we can also define more practical `read` and `write` operations to update a single mapping.

The module `FacState` *abstractly* depends on the `MultefInterface`, and so the extension constitutes what we have previously called “client code”. This means that we have no proof obligation for noninterference for `FSt` computations, it follows from noninterference for the `sec : MultefInterface` instantiation of the `Multef` interface. That is, because all the code of this section treats `Multef` abstractly, we do not need to define any special logical relation or proof, and can use the standard interpretation of Definitions 1 and 2.

The `FSt` monad allows us to implement the following canonical example for faceted state adapted from Austin and Flanagan [27]:

```

example : Fac Bool → FSt Bool
example fx = do
  write "y" True
  write "z" True

  x ← lift fx
  if x then write "y" False else return tt

  y ← read "y"
  if y then write "z" False else return tt

```

```

z ← read "z"
return z

```

The module in Figure 10 is parametric in the implementation `imp : MultefInterface` of the faceted library. One only obtains actual runnable code by linking this module with the implementation of `MultefInterface` one is interested in. Naturally, whatever guarantees are provided by that implementation will then be inherited by the resulting `FacState imp` module. This means that noninterference for `FacState sec` follows automatically from noninterference for `sec : MultefInterface`.

7 Related and Future Work

Dynamic IFC libraries, like LIO [3] and Multef [7], promise to provide noninterference guarantees without the need for a specialized tool-chain. Embedding such libraries in existing languages allows programmers to reuse the functionality and library ecosystem of the host language. Case studies show that this is a promising direction for IFC [3, 13, 43].

Stefan et al. [10] provide a noninterference proof for LIO verified by Coq, which we improve upon in several ways. First, we cover a larger subset of the features of LIO. Crucially, we additionally support one key feature of LIO: allowing the user to catch IFC exceptions. Second, our proofs fit in just under 300 lines of Agda, while the proof of noninterference of Stefan et al. [10] is more than 3000 lines of Coq.

We achieve this order-of-magnitude improvement partly by using logical-relations to reason about LIO. However, this alone is not sufficient to achieve the concise proof that we have presented here. We also rely on Theorem 1 to automatically derive the relational interpretation of the “standard” parts of the language (crucially λ -abstraction): a large part of what one would traditionally need to carry out manually.

Third, and importantly, we certify *the library itself*, rather than a model of the library — an issue which affects much of the library-based IFC literature. This issue is not surprising, because in order to be practical the libraries are implemented in languages that are lacking a formal semantics based on logical relations. Formalizing library-based IFC would require developing such a semantics first — a daunting task for a full-featured language. In contrast, we use a less mainstream language, Agda, but, in return, there is no informal, unverified, modeling step remaining in our approach.

To be sure, this means that we also do not claim to provide guarantees for the *Haskell* implementation of the libraries, but rather the Agda implementation in this paper. However, we believe this paper demonstrates that as techniques for reasoning about logical relations for languages like Haskell are developed further, the necessary meta-theory for practical IFC libraries will come within reach of our techniques.

Specifically, there are a number of Haskell libraries for IFC that rely on complicated language features for controlling concurrency [19, 44] and parallelism [45] and specialized primitives for managing laziness [20]. In order to be able to reason about the implementation of IFC libraries that use these features, meta-theoretical tools akin to the parametricity theory that we rely on will need to be developed for Haskell.

Another difference between Haskell and Agda is that Agda is a total language, so all code written in it is guaranteed to terminate and so the termination channel [46] is a non-issue. Extending our techniques to deal with nontermination, and specifically to deal with libraries that protect against termination leaks (e.g., [7, 19]), is interesting future work. In particular, we hope to build on various monadic representations of nontermination in dependent type theory [47, 48].

Last but not least, Algehed and Bernardy [26] pioneered the technique of using parametricity for dependent types to show noninterference for a security library. They are concerned with *static* IFC libraries, while we show that the technique extends naturally to libraries for *dynamic* IFC libraries. Furthermore, in Theorem 4 we show that the parametricity proof-technique works well for proving meta-theoretical properties other than noninterference for security libraries using this approach.

8 Conclusion

This paper shows the versatility of parametricity as a proof technique for language-based security. Specifically, we show that parametricity can be used to prove noninterference for two different dynamic IFC libraries as well as transparency for the faceted values library.

Ours are the first proofs of noninterference for dynamic IFC libraries that are about the actual implementation of a library. Previous proofs of noninterference for dynamic IFC libraries (e.g. [3, 7]) are about separate lambda-calculus that mimics the semantics of the library, rather than about the library itself. Furthermore, using parametricity allows us to give compact yet fully machine-checked proofs.

We believe that the simplicity of the proof techniques used in this paper will be key to scaling noninterference proofs to cover the *actual code* of more feature-rich IFC libraries, like concurrent [8, 19, 20] and cryptographic [4] LIO.

References

- [1] P. Li and S. Zdancewic, “Encoding information flow in Haskell,” in *19th IEEE Computer Security Foundations Workshop (CSFW’06)*. IEEE, 2006, pp. 12–pp.
- [2] A. Russo, “Functional pearl: Two can keep a secret, if one of them uses Haskell,” in *ACM SIGPLAN Notices*, vol. 50, no. 9. ACM, 2015, pp. 280–288.
- [3] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, “Flexible dynamic information flow control in Haskell,” in *ACM Sigplan Notices*, vol. 46, no. 12. ACM, 2011, pp. 95–106.
- [4] L. Waye, P. Buiras, O. Arden, A. Russo, and S. Chong, “Cryptographically secure information flow control on key-value stores,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1893–1907.
- [5] P. Buiras, D. Vytiniotis, and A. Russo, “HLIO: Mixing static and dynamic typing for information-flow control in Haskell,” in *ACM SIGPLAN Notices*, vol. 50, no. 9. ACM, 2015, pp. 289–301.

- [6] M. Algehed and A. Russo, "Encoding DCC in Haskell," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. ACM, 2017, pp. 77–89.
- [7] T. Schmitz, M. Algehed, C. Flanagan, and A. Russo, "Faceted secure multi execution," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1617–1634.
- [8] P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazieres, "A library for removing cache-based attacks in concurrent information flow systems," in *International Symposium on Trustworthy Global Computing*. Springer, 2013, pp. 199–216.
- [9] A. Russo, K. Claessen, and J. Hughes, "A library for light-weight information-flow security in Haskell," in *ACM Sigplan Notices*, vol. 44, no. 2. ACM, 2008, pp. 13–24.
- [10] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo, "Flexible dynamic information flow control in the presence of exceptions," *J. Funct. Program.*, vol. 27, p. e5, 2017. [Online]. Available: <https://doi.org/10.1017/S0956796816000241>
- [11] M. Vassena, A. Russo, P. Buiras, and L. Wayne, "Mac a verified static information-flow control library," *Journal of logical and algebraic methods in programming*, vol. 95, pp. 148–180, 2018.
- [12] M. Vassena, P. Buiras, L. Wayne, and A. Russo, "Flexible manipulation of labeled values for information-flow control libraries," in *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, I. G. Askoxylakis, S. Ioannidis, S. K. Katsikas, and C. A. Meadows, Eds., vol. 9878. Springer, 2016, pp. 538–557. [Online]. Available: https://doi.org/10.1007/978-3-319-45744-4_27
- [13] J. Parker, N. Vazou, and M. Hicks, "Lweb: Information flow security for multi-tier web applications," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.
- [14] M. Algehed, "A perspective on the dependency core calculus," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 24–28. [Online]. Available: <https://doi.org/10.1145/3264820.3264823>
- [15] T. F. J. Pasquier, J. Bacon, and B. Shand, "FlowR: aspect oriented programming for information flow control in ruby," in *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, W. Binder, E. Ernst, A. Peternier, and R. Hirschfeld, Eds. ACM, 2014, pp. 37–48. [Online]. Available: <https://doi.org/10.1145/2577080.2577090>
- [16] D. Terei, S. Marlow, S. L. Peyton Jones, and D. Mazières, "Safe Haskell," in *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, J. Voigtländer, Ed. ACM, 2012, pp. 137–148. [Online]. Available: <https://doi.org/10.1145/2364506.2364524>
- [17] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson *et al.*, "Report on the programming language Haskell: a non-strict, purely functional language version 1.2," *ACM SigPlan notices*, vol. 27, no. 5, pp. 1–164, 1992.
- [18] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.
- [19] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazieres, "Addressing covert termination and timing channels in concurrent information flow systems," *ACM SIGPLAN Notices*, vol. 47, no. 9, pp. 201–214, 2012.
- [20] P. Buiras and A. Russo, "Lazy programs leak secrets," in *Nordic Conference on Secure IT Systems*. Springer, 2013, pp. 116–122.
- [21] M. Vassena, J. Breitner, and A. Russo, "Securing concurrent lazy programs against information leakage," in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 37–52. [Online]. Available: <https://doi.org/10.1109/CSF.2017.39>
- [22] T. Austin, K. Knowles, and C. Flanagan, "Typed faceted values for secure information flow in Haskell," Technical Report UCSC-SOE-14-07, University of California, Santa Cruz, Tech. Rep., 2014.
- [23] J. C. Reynolds, "Types, abstraction and parametric polymorphism," 1983.
- [24] P. Wadler, "Theorems for free!" in *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. ACM, 1989, pp. 347–359.
- [25] J.-p. Bernardy, P. Jansson, and R. Paterson, "Proofs for free: Parametricity for dependent types," *Journal of Functional Programming*, vol. 22, no. 2, pp. 107–152, 2012.
- [26] M. Algehed and J.-P. Bernardy, "Simple noninterference from parametricity," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. [Online]. Available: <https://doi.org/10.1145/3341693>
- [27] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *ACM Sigplan Notices*, vol. 47, no. 1. ACM, 2012, pp. 165–178.
- [28] A. Bove, P. Dybjer, and U. Norell, "A brief overview of agda—a functional language with dependent types," in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 73–78.
- [29] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your IFCEException are belong to us," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 3–17.
- [30] E. Moggi, "Notions of computation and monads," *Inform.*

- tion and computation, vol. 93, no. 1, pp. 55–92, 1991.
- [31] P. Wadler, “Monads for functional programming,” in *International School on Advanced Functional Programming*. Springer, 1995, pp. 24–52.
 - [32] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.
 - [33] L. Cardelli and P. Wegner, “On understanding types, data abstraction, and polymorphism,” *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, pp. 471–523, 1985.
 - [34] W. A. Howard, “The formulae-as-types notion of construction,” *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, vol. 44, pp. 479–490, 1980.
 - [35] P. Wadler, “Propositions as types,” *Commun. ACM*, vol. 58, no. 12, pp. 75–84, 2015. [Online]. Available: <https://doi.org/10.1145/2699407>
 - [36] G. D. Plotkin, “Lambda-definability and logical relations, 1973,” *Memorandum SAI-RM*, vol. 4, 1973. [Online]. Available: http://homepages.inf.ed.ac.uk/gdp/publications/logical_relations_1973.pdf
 - [37] R. Statman, “Logical relations and the typed lambda-calculus,” *Information and Control*, vol. 65, no. 2/3, pp. 85–97, 1985. [Online]. Available: [https://doi.org/10.1016/S0019-9958\(85\)80001-2](https://doi.org/10.1016/S0019-9958(85)80001-2)
 - [38] W. J. Bowman and A. Ahmed, “Noninterference for free,” *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 101–113, 2015.
 - [39] M. Algehed, A. Russo, and C. Flanagan, “Optimising Faceted Secure Multi-Execution,” in *Proc. of the 2019 32nd IEEE Computer Security Foundations Symp.*, ser. CSF ’19. IEEE Computer Society, 2019.
 - [40] W. J. Bowman and A. Ahmed, “Noninterference for free,” *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 101–113, 2015.
 - [41] M. Ngo, D. Naumann, and T. Rezk, “Typed-based relaxed noninterference for free,” *arXiv preprint arXiv:1905.00922*, 2019.
 - [42] S. Liang, P. Hudak, and M. Jones, “Monad transformers and modular interpreters,” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’95. New York, NY, USA: Association for Computing Machinery, 1995, p. 333–343. [Online]. Available: <https://doi.org/10.1145/199448.199528>
 - [43] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, “Hails: Protecting data privacy in untrusted web applications,” in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 47–60.
 - [44] M. Vassena, A. Russo, P. Buiras, and L. Waye, “Mac a verified static information-flow control library,” *Journal of Logical and Algebraic Methods in Programming*, vol. 95, pp. 148 – 180, 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S235222081730069X>
 - [45] M. Vassena, G. Soeller, P. Amidon, M. Chan, J. Renner, and D. Stefan, “Foundations for parallel information flow control runtime systems,” in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds. Cham: Springer International Publishing, 2019, pp. 1–28.
 - [46] D. Hedin and A. Sabelfeld, “A perspective on information-flow control,” *Software Safety and Security*, vol. 33, pp. 319–347, 2012.
 - [47] C. McBride, “Turing-completeness totally free,” in *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, ser. Lecture Notes in Computer Science, R. Hinze and J. Voigtländer, Eds., vol. 9129. Springer, 2015, pp. 257–275. [Online]. Available: https://doi.org/10.1007/978-3-319-19797-5_13
 - [48] L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, “Interaction trees: representing recursive and impure programs in coq,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 51:1–51:32, 2020. [Online]. Available: <https://doi.org/10.1145/3371119>