A graded modal approach to relaxed semantic declassification

Vineet Rajani
University of Kent
United Kingdom
V.Rajani@kent.ac.uk

Alex Coleman
University of Kent
United Kingdom
ac2049@kent.ac.uk

Hrutvik Kanabar
*University of Kent**
United Kingdom
hrk32@cantab.ac.uk

Abstract—**In this paper we present Declassification Core Calculus (DeCC), a graded modal type theory for relaxed semantic declassification, a declassification criterion inspired from Delimited Release and Relaxed Noninterference. We build upon Dependency Core Calculus (DCC) that already has a graded monad for classification of information. DeCC inherits DCC's graded monad, but adds a new modality for the purpose of declassification. We build a logical relation model describing both the unary and relational semantics of the types including the two graded modalities, and use this model to prove the soundness of DeCC. We describe how our new modality interacts with DCC's graded monad via distributive laws, and also describe the conditions under which our new modality forms a comonad. This work has been mechanised in the HOL4 theorem prover.**

Index Terms—**Graded Modal Types; Information Flow Control; Semantic Declassification**

I. Introduction

Ensuring that programs do not leak confidential information is subtle. Mere access control and cryptography are not enough to provide confidentiality guarantees [1], as they can only control who gets access to the information, but not what happens to the information after it has been released. Providing end-to-end confidentiality guarantees require tracking and controlling the flow of information as it percolates through the system, this is achieved using *Information Flow Control (IFC)*. In a language-based setting, IFC works by associating security labels (denoting a confidentiality level) with data and tracking these labels through the execution of the program. The security labels are related using a flow relation (often described using a lattice structure) defining the permitted flow of information. A sound enforcement of IFC must guarantee that only permitted flows are allowed.

A soundness criterion for IFC is often defined as a variant of *noninterference* [2] (there is no interference/influence of secret inputs on the public outputs of a program). Technically, noninterference is an instance of a *hyperproperty* [3] which involves reasoning about two executions of the same program on different secrets. Modal type systems provide an attractive avenue for enforcing and reasoning about noninterference statically, modal type constructors record security labels as grades and typing rules for those modal types help track and prevent the flow of undesired information. Dependency Core Calculus (DCC) [4] is a stereotypical example of a graded modal type theory for information flow control. DCC makes use of $\Diamond_\ell\tau$[1] a modal possibility operator graded with a security label, often just referred to as a graded monad in this setting, to track and control the flow of information for a core functional calculus and describe its categorical semantics. Such a use of graded monads was later extended to track the flow of information for a language with higher-order mutable state in the CG (short for Coarse Grained) type system [5], [6]. CG uses a doubly graded monad with two separate labels to track information via both read and write effects. The static semantics of CG is defined using a Kripke possible-world model, which is then used to derive the noninterference theorem.

While there is a strong evidence supporting the connection between modal logic and information classification in a language-based setting, such a connection (both proof theoretically and semantically) is not well understood when it comes to aspects pertaining to properties of information declassification. This work aims to bridge this gap by introducing a new graded modal type theory which uncovers some of those connections. Central to our type theory is a new modal operator, $\Box_\phi\tau$ (simplified form), intuitively specifying the type of terms that can be inspected/declassified using the policy ϕ. Unlike the graded modalities in DCC and CG, the grade ϕ of our $\Box$ modality is a lambda expression from the term language of our calculus, and hence has a flavour of a dependent graded modality [7] where grades depend on terms. To the best of our knowledge, this is the first work that describes the use of dependent graded types to reason about information declassification.

The focus of this work is on the "what" dimension [8] of information disclosure. This dimension of declassification has been classically studied in approaches like delimited release [9] and relaxed noninterference [10]. Delimited release studies this problem in an imperative and heap-based setting, and allows for reasoning about declassification of secrets wrt initial memory. Our approach is complementary, we study this problem in a functional and higher-order setup, and our approach permits reasoning about composition and decom-

* Part of this research was undertaken while this author was affiliated with the University of Kent.

[1]DCC uses the symbol T for the graded monad, but we prefer $\Diamond$ as DCC's monad is a graded version of the possibility ($\Diamond$) modality from the modal logic S4.

position of policies in an elegant manner. This allows us to express scenarios where a policy can itself return a policy-controlled data which can be declassified at a later point (Section VI-C), we believe such examples are not directly expressible in the delimited release framework.

Our approach also enhances another classical approach called relaxed noninterference [10]. The approach of relaxed noninterference is more type-driven like ours, it allows for declassification policies to appear in types and is also setup in a higher-order setting. However, the relaxed noninterference framework is designed to work with a security lattice constructed over sets of policies, while in our approach we keep the syntactic category of security labels and declassification policies completely separate. We believe this makes our approach more general and can allow for choosing different policies to declassify to different principals. Also, the relaxed noninterference approach is proven sound wrt a syntactic slicing-based soundness criterion, while we choose a semantic indistinguishability-based security condition. We also allow for reasoning with semantically-secure policies, policies that are actually secure but cannot be proved so under the restrictions of a security type system. This allows us to reason about scenarios pertaining to partial and conditional declassification (Section VI-D) via operation like split, which we believe cannot be directly expressed in the relaxed-noninterference framework.

Overview of our work: In this paper we present *DeCC*, short for *Declassification Core Calculus*, a core language with a graded modal type theory to reason about information declassification. In particular, DeCC uses two graded modalities, one for classification and the other for declassification of information. For classification, DeCC inherits the graded monad $\Diamond_\ell \tau$ of DCC, which allows for annotating a type τ with a security label ℓ. For declassification, DeCC introduces a new graded modality, $\Box_{\phi,\tau'}\tau$. The new box modality ascribes the type of a term of type τ which can be consumed by a policy ϕ of type $\tau \to \tau'$. In particular, τ can be the type of a secret like $\Diamond_H \mathbb{N}$ (a high/secret labelled natural number) and the policy ϕ can have a type like $\Diamond_H \mathbb{N} \to \Diamond_L \mathbb{N}$ (a function which reads a secret number as input and then returns a low/public number as output). Thus, applying ϕ to the given secret can lead to a potential declassification. Intuitively, only terms inside the box are declassifiable and the typing rules of DeCC are setup to ensure that the only way to declassify a secret inside a box is through the application of the allowed policy in its type.

DeCC also incorporates an ability to reason about declassification with semantically secure policies (functions which are actually secure but cannot be proved so within the syntactic constraints of a security type system). For example, the function which takes a secret number, doubles it and returns the parity of the result as a low output, $\lambda x.(2 * x)\%2$ where $\%$ denotes the remainder operator, will be rejected by most information flow type systems because there is a syntactic dependency of the secret x on the output. However, this program is actually secure, as no matter what the value of x is it will always produce a 0 as the result. In DeCC we can use such functions as policies to release information. We formalise such a semantic security using the technique of logical relations inspired from the semantics of CG [5], [6] (which is a type system for vanilla noninterference and does not incorporate any notion of declassification). We describe several operations over terms of this new box modality which permit a variety of declassification use cases. On the meta-theory side, we describe how our new box modality interacts with the DCC's graded monad via distributive laws and reflect on their practical significance in the context of information flow. We also describe the conditions under which our box modality forms a comonad. Finally, we provide a logical relations based proof of soundness against Relaxed Semantic Declassification (our soundness criterion), inspired from delimited release and relaxed noninterference.

Summary of contributions: To summarise, we make the following contributions in this paper.

- We present DeCC, a core calculus for declassification with a graded modal type theory for enforcing declassification guarantees statically over higher-order functional programs. To reason about declassification, DeCC introduces a new graded modality ($\Box_{\phi,\tau'}\tau$) which allows controlled inspection and release of secrets using the policy ϕ. The policy ϕ is a function from the term calculus of DeCC, and hence introduces a flavour of dependent grades for reasoning about declassification.
- Additionally, our type theory is built to accommodate reasoning with policies which can be both declassifying and semantically noninterfering. This allows us to express a variety of practical declassification scenarios like partial and conditional declassification, and state machine-based declassification.
- On the meta theory side, we describe how our new declassification modality interacts with the graded monad of DCC via distributive laws and also describe the conditions under which our new modality forms a comonad.
- We build a logical relation model for the types of DeCC, which we use to prove our type theory sound and also to provide semantic invariants on the declassification policies.
- Finally, we have mechanised the entire type theory including the logical relation models, meta-theoretic results and the soundness proof using the HOL4 theorem prover [11].

Organisation: We begin with a brief background on the relevant ideas from prior work in Section II. In Section III we describe our language, DeCC, both its statics and its dynamics. After that we describe the type-theoretic aspects of DeCC in Section IV. This includes describing its typing rules, semantic model and soundness. In Section V we discuss some meta-theoretic aspects which includes showing the comonadic nature of the box modality under certain restrictions, and describing its interaction with the graded monad. Section VI describes several examples to demonstrate the applicability of DeCC for encoding different kinds of declassification policies. In Section VII we describe a weaker type system to type check policy code. We also briefly talk about an alternate development of DeCC with call-by-name semantics, and finally share some thoughts on extension of DeCC with other language features. We provide a description of related work in Section VIII and finally conclude in Section IX.

II. Background

In this section we provide a brief background on the ideas we build upon: from graded modal type theories namely DCC and CG, and relevant declassification approaches.

A. Graded modal types for IFC

Graded modal types provide an elegant formalisation and enforcement of information flow properties. To the best of our knowledge the first application of graded modal types for information flow control was demonstrated via DCC [4] (short for Dependency Core Calculus). The core idea of DCC is the use of a graded monad to annotate types with security labels, and tracking of such labels via typing of the monads unit and bind. Doing so allows treating security labels as effects of computation and confines the task of dependency tracking to just a few rules while keeping the rest of the language effect free (or pure). The proof theory of DCC is proved sound against noninterference by building a categorical model for its types. This approach of enforcing information flow properties has become synonymous with the coarse-grained style of IFC, as labels are only associated with the monadic type and not with others. This is in contrast with the fine-grained style of information flow tracking which assigns and tracks label with every type [12] in the language.

The core idea of DCC was later generalised in another type-theoretic framework called CG [5], [6] (short for Coarse Grained) to languages with other channels of information leaks, in particular dynamic memory allocation and mutable heaps. CG uses two different graded modalities one for associating labels and the other for tracking information across memory locations. It was also shown in [5], [6] that such graded modalities can be semantically interpreted using a possible-worlds model (inherent to many modal logics) built using a logical relations framework, which was used to prove the soundness of CG against noninterference. The type theory CG is quite expressive and in fact is equally expressive to the fine-grained style of information tracking.

B. Delimited release

Delimited release [9] is a classical approach of controlling "what" information is allowed to be leaked. The core idea of delimited release is to only allow information to be leaked via specific terms called an escape hatches. These escape hatches are specified using a declassify construct which takes two arguments: an expression e which is to be declassified and a security label ℓ to which this information is allowed to be declassified.

Definition 1 (Delimited Release [9])**.**
Suppose the command c contains within it exactly n $\mathsf{declassify}$ *expressions,* $\mathsf{declassify}(e_1, \ell_1), \ldots, \mathsf{declassify}(e_n, \ell_n)$. *Command c is secure if for all security levels ℓ we have $\forall H_1, H_2$.*

$$(H_1 \approx H_2 \;\&\; \forall i \in \{i | \ell_i \sqsubseteq \ell\}.(H_1, e_i) \approx (H_2, e_i))$$
$$\implies$$
$$(H_1, c) \approx (H_2, c)$$

The formal definition of delimited release from [9] is described in Definition 1 above. It is as an embellishment of termination-insensitive noninterference with some preconditions on the information intended to be declassified. Basically, it says that if the escape hatches are semantically indistinguishable by an adversary then so are the commands using those escape hatches. Formally it is stated as follows, given two equivalent heaps i.e. heaps which can only differ in their secret content, but must have the same public data (denoted by $H_1 \approx H_2$), a command c is secure iff the following holds: if none of the escape hatches (i.e. all the declassify expressions) in the command c can distinguish the two equivalent heaps (up to termination), then the command c should not be able to distinguish (again up to termination) between those equivalent heaps either. The termination insensitivity is baked into the indistinguishability of the commands configuration $(H_1, c) \approx (H_2, c)$ (and similarly for expressions).

C. Relaxed Noninterference

Another approach to "what" declassification is the one by Li and Zdancewic, it called relaxed noninterference [10]. The core idea of relaxed noninterference is to allow only that information to be released that is allowed by some given policies. The policies are specified as sets of lambda functions, which are treated as labels and are used as annotations on types (unlike delimited release where policies only appear at the term level). For example, a secret number n can be assigned a policy $\lambda x.x == c$, which when applied to n will only allow it to be compared to a constant c and would return a boolean result of that comparison. The soundness criterion for relaxed noninterference approach is not a hyperproperty demanding indistinguishability, but instead it is a syntactic slicing criterion which allows programs to be factored into two parts: one that syntactically contain no secrets, and the other that only has application of declassification policies on secrets. The formalisation of their soundness criterion is not relevant here as we do not use that criterion in this paper, but it can be found in [10].

III. DeCC: The language

In this section we describe the statics and dynamics of DeCC.

A. Statics

As usual in many information flow type systems, in DeCC we use security labels (denoted by ℓ) drawn from an arbitrary security lattice $(\mathcal{L}, \sqsubseteq)$. The least and top elements of the lattice are denoted by $\perp$ and $\top$ respectively. For ease of exposition we only talk about confidentiality lattice here, but nothing in our development is specific to confidentiality lattice only. In fact, our HOL mechanisation [13] is completely parametric in

$$\begin{array}{lll} \text{Types} \quad \tau & ::= & \mathbb{N} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_1 \mid \Diamond_\ell \tau \mid \Box_{\phi,\tau}\tau \\ \text{Expressions} \quad e, \phi & ::= & v \mid x \mid e_1 \odot e_2 \mid e_1\ e_2 \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ x.e;\ y.e \mid \\ & & \mathsf{dec}\ e\ e \mid \mathsf{coret}\ e \mid \mathsf{cojoin}\ e \mid \mathsf{split}\ e \\ \text{Values} \quad v & ::= & \mathsf{N} \mid \lambda x.e \mid (v, v) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{ret}\ e \mid \mathsf{bind}\ e_1 = x\ \mathsf{in}\ e_2 \mid \Box\ e \end{array}$$

Fig. 1. Syntax of types and terms

the kind of labels used, and only requires some abstract lattice axioms. As a result, the whole development will also work for any security lattice including an integrity lattice or a product lattice of confidentiality and integrity.

Types: DeCC's type language (Fig. 1) consists of a base type which we choose as the type of natural numbers $\mathbb{N}$, a function type $(\tau_1 \rightarrow \tau_2)$, products $(\tau_1 \times \tau_2)$ and sums $(\tau_1 + \tau_2)$. In addition, for the purpose of security, we add two modal types to DeCC's type language: 1) a graded monad $\Diamond_\ell \tau$ (which we refer to as the classification or the diamond modality) for classification of information, that we inherit from DCC [4], and 2) a new graded modality $\Box_{\phi,\tau'}\tau$ (which we refer to as the declassification or the box modality) which we introduce for the purpose of incorporating information declassification.

The classification modality $\Diamond_\ell \tau$ describes the type of a term of type τ with a confidentiality label ℓ associated with it. The label ℓ can be thought of as a upper bound on the level of secrets that have been inspected in the *past* to obtain a value of type $\Diamond_\ell \tau$. The declassification modality $\Box_{\phi,\tau'}\tau$ on the other hand describes a term of type τ that can be inspected by a policy ϕ of type $\tau \rightarrow \tau'$ in the *future* producing a result of type τ' after declassification[2].

Note that our declassification modality $(\Box_{\phi,\tau'}\tau)$ is defined for a general τ and τ' and not just for monadic types. Declassification just falls out as a special case here by specialising the τ and τ' to monadic types $\Diamond_\ell -$ to $\Diamond_{\ell'} -$ respectively, where $\ell' \sqsubset \ell$. The more general version comes in handy for proving meta-theoretic properties as defined later in Section V. Also, we believe this generality (of controlling the future use of a term of type τ via ϕ) could have other applications even outside of security.

Terms: The syntax of DeCC's terms is described in Fig. 1. We use $\odot$ to denote an abstract binary operation on natural numbers which we instantiate with specific operators for the sake of examples. Here we only focus on the terms pertaining to the classification and the declassification modalities, as the rest are standard.

For the classification modality, we inherit the DCC's ret and bind operations. The constructor ret e allows for insertion of a term e into the classification monad. The bind $x = e_1$ in e_2 operation on the other hand defines how to sequentially compose two labelled terms, where e_2 can refer to the value of e_1 via x. The multiplication or the join operation can be defined using bind in a standard way join $\triangleq \lambda x.\mathsf{bind}\ y = x\ \mathsf{in}\ x$.

For the declassification modality, the term $\Box e$ defines a value of the box type. Boxed values can be provided from the context, or can be created using an inject constructor. The dec ϕ e is the only construct in the language which allows for declassification of information, it is meant to be read as declassify e using a policy ϕ[3]. Dual to the classification modality we introduce coret and cojoin as the co-unit and co-multiplication constructs. Additionally, we also have a generalisation of cojoin called the split with the same semantics but different proof theory, as we explain in later sections.

B. Dynamics

We define the semantics of DeCC using two evaluation relations, a pure $(\Downarrow)$ and a forcing $(\Downarrow^f)$ evaluation. The pure reduction defines the evaluation of the effect-free or pure terms while the forcing reduction defines the evaluation of the monadic ones. This kind of a separation is fairly standard and has been used in the literature for many Haskell-like languages with a modal type system [5], [14], [15].

The pure reduction is defined using a standard call-by-value semantics. We only describe selected rules pertaining to the declassification modality here (Fig. 2), all the other pure evaluation rules are standard and are included in our HOL mechanisation [13]. **E-coret** allows for extracting out the boxed expression without the application of the policy, if e evaluates to $\Box e'$ and e' evaluates to v then coret e evaluates to v. **E-cojoin** does the reverse, if e evaluates to $\Box e'$ then cojoin e adds another box on top resulting in $\Box\Box e'$. The semantics of split is identical to cojoin, they only differ in their typing rules and their meta theory. **E-dec** defines the semantics of the dec construct, if e evaluates to $\Box e'$ then dec returns the same value as the value returned by the application of the policy ϕ on e'.

The monadic terms ret and bind are treated as values in the pure semantics and their evaluation is defined using the forcing relation $\Downarrow^f$, also described in Fig. 2. **E-ret** defines the semantics of the ret construct, if e evaluates to v under the pure semantics, then ret e evaluates to the same value v under the forcing semantics. Finally, the **E-bind** rule describes the semantics of composing two monadic terms, e_2 after e_1, via the bind construct.

[2]For brevity we may omit the return type τ' wherever it is not required or is implicit from the context, and just write the declassification type as $\Box_\phi \tau$.

[3]Our HOL formalisation [13] introduces dec in the let style, which is just a sugared representation of let x = dec ϕ e in x. Here we present the desugared version, dec ϕ e, for brevity.

Pure evaluation judgement: $\boxed{e \Downarrow v}$ Forcing evaluation judgement: $\boxed{e \Downarrow^f v}$

$$\frac{e \Downarrow \Box\, e' \qquad e' \Downarrow v}{\mathsf{coret}\ e \Downarrow v}\ \text{E-coret} \qquad \frac{e \Downarrow \Box\, e'}{\mathsf{cojoin}\ e \Downarrow \Box\, \Box\, e'}\ \text{E-cojoin} \qquad \frac{e \Downarrow \Box\, e'}{\mathsf{split}\ e \Downarrow \Box\, \Box\, e'}\ \text{E-split} \qquad \frac{e \Downarrow \Box\, e' \qquad \phi\ e' \Downarrow v}{\mathsf{dec}\quad \phi\ e \Downarrow v}\ \text{E-dec}$$

$$\frac{e \Downarrow v}{\mathsf{ret}\ e \Downarrow^f v}\ \text{E-ret} \qquad \frac{e_1 \Downarrow v_1 \qquad v_1 \Downarrow^f v_1' \qquad e_2[v_1'/x] \Downarrow v_2 \qquad v_2 \Downarrow^f v_2'}{\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow^f v_2'}\ \text{E-bind}$$

Fig. 2. Pure and Forcing evaluation (selected) rules

IV. DeCC: The Type theory

In this section we describe the type-theoretical aspects of DeCC. We begin by describing the typing rules. After that we build a logical relation-based semantic model of DeCC's types and use that to prove the soundness of our type system.

A. Type system

The typing judgement of DeCC is given by $\Delta; \Phi; \Gamma \vdash e : \tau$. Here, Γ is a context mapping free variables of e to their types, and Δ, Φ are the two policy contexts containing pairs of closed policies (function combinators) along with their types. The context Δ is the context of policies which are assumed to be semantically secure, while the policies in the context Φ are only assumed to be type safe (under a non IFC type system, which we will describe later). We make this distinction to clearly disentangle two kinds of declassification subsumed in DeCC: releasing information via policies which are noninterfering or semantically secure (such policies reside in Δ), and releasing information via policies which might be interfering and overrides the default policies as given by the labelling (such policies reside in Φ). This distinction also allows us to define operations like inject and split which enables examining meta-theoretical properties of DeCC like distributive laws (Section V-B), and encode scenarios like partial and conditional declassification (Section VI-D).

Typing rules: We describe selected typing rules for the constructs relevant to the classification and declassification modalities in Fig 3. The typing rules for the non-modal types are totally standard so we do not explain them here, but can be found in the accompanying HOL artefacts [13].

We first describe the typing rules for the classification monad. **T-ret** describes the typing for the unit/ret of the monad, it allows for injecting a term of type τ into the monad by assigning the least confidentiality label $\bot$ to it. **T-bind** defines the typing for the composition of two monadic terms e and e'. The rule makes sure that the final label ℓ' of the continuation term e' is at least equal to the label ℓ of the term e inspected before (in the past). This label check, $\ell \sqsubseteq \ell'$[4], is crucial to prevent information leaks. The multiplication/join as mentioned earlier is definable within the calculus, $\mathsf{join} \triangleq \lambda x.\mathsf{bind}\ y = x\ \mathsf{in}\ x$, and it can be given a type, $\mathsf{join} : \Diamond_{\ell_1}(\Diamond_{\ell_2}\tau) \to \Diamond_{\ell_1 \sqcup \ell_2}\tau$.

We now describe the typing rules for the declassification modality. **T-coret** describes the typing of the coret construct, it allows for extraction of a term from under the box. Doing so is sound because declassification is only possible for terms inside the box. Once a term is extracted out it is basically subject to the standard DCC typing, which does not allow declassification. **T-cojoin** allows wrapping an existing boxed value into another box which can only be inspected by an identity policy, and hence cannot be further declassified. Although split and cojoin behave identically in the semantics, their typing rules are very different. **T-split** describes the typing of the split construct. If the input policy ϕ is a composition of two policies, ϕ_2 after ϕ_1, then it can be broken up over two nested boxes with ϕ_1 and ϕ' ($\triangleq \lambda x.\mathsf{let}\ y = \mathsf{dec}\ \phi_1\ x\ \mathsf{in}\ \phi_2\ y$) as their policies. The policy ϕ' uses the dec construct to first declassify using the ϕ_1 and then its results using ϕ_2, thereby simulating the original composition. For this rule to be sound, we require the inner policy ϕ_1 to be semantically secure (on all inputs), and hence must come from Δ. But the constraints required for the outer policy are weaker. In particular, ϕ_2 is only required to be type safe (in a standard non-IFC sense), i.e. ϕ_2 should be a member of Φ, but it can be interfering and can declassify information. This is because the given policy ϕ is only expected to provide indistinguishability guarantees about the *final results* of the application of ϕ on indistinguishable terms of type τ in the box, but not about the *intermediate* ones which ϕ_1 can produce. So, to be able to prove indistinguishability guarantees in a compositional manner we need this additional assumption about ϕ_1 being in Δ. Also, intuitively, one can use a coret over a split construct to get rid of the outer box. So, if the inner policy is not semantically secure, we cannot guarantee indistinguishability. **T-inject** rule allows creation of a boxed term of type $\Box_\phi\tau$ using a well typed term of type τ and a semantically secure policy $\phi : \tau \to \tau'$ from Δ. Again ϕ is required to be in Δ for similar reasons as explained in the T-split rule. Finally, **T-dec** describes the typing of the dec construct. Note that, this rule is sound even under a weaker assumption on the policy ϕ being just type safe (and not necessarily secure). This is because indistinguishability of boxed inputs of type $\Box_{\phi,\tau'}\tau$

[4]Note that our typing for bind differs from that of DCC. DCC ensures that the type of the continuation term e' is protected at level ℓ using a separate protection relation which subsumes our check $\ell \sqsubseteq \ell'$. Generalisation to the full protection relation of DCC is an orthogonal expressivity issue which we leave for future extension.

Typing judgement: $\boxed{\Delta; \Phi; \Gamma \vdash e : \tau}$

$$\frac{\Delta; \Phi; \Gamma \vdash e : \tau}{\Delta; \Phi; \Gamma \vdash \mathsf{ret}\ e : \Diamond_{\perp}\tau}\ \text{T-ret} \qquad \frac{\Delta; \Phi; \Gamma \vdash e : \Diamond_{\ell}\tau \qquad \Delta; \Phi; \Gamma, x : \tau \vdash e : \Diamond_{\ell'}\tau' \qquad \ell \sqsubseteq \ell'}{\Delta; \Phi; \Gamma \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : \Diamond_{\ell'}\tau'}\ \text{T-bind}$$

$$\frac{\Delta; \Phi; \Gamma \vdash e : \Box_{\phi}\tau}{\Delta; \Phi; \Gamma \vdash \mathsf{coret}\ e : \tau}\ \text{T-coret} \qquad \frac{\Delta; \Phi; \Gamma \vdash e : \Box_{\phi,\tau'}\tau}{\Delta; \Phi; \Gamma \vdash \mathsf{cojoin}\ e : \Box_{id}\Box_{\phi,\tau'}\tau}\ \text{T-cojoin}$$

$$\frac{\Delta; \Phi; \Gamma \vdash e : \Box_{\phi,\tau'}\tau \qquad \Phi(\phi) = \tau \to \tau' \vee \Delta(\phi) = \tau \to \tau'}{\Delta; \Phi; \Gamma \vdash \mathsf{dec}\ \phi\ e : \tau'}\ \text{T-dec} \qquad \frac{\Delta; \Phi; \Gamma \vdash e : \tau \qquad \Delta(\phi) = \tau \to \tau'}{\Delta; \Phi; \Gamma \vdash \Box e : \Box_{\phi,\tau'}\tau}\ \text{T-inject}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash e : \Box_{\phi_2 \cdot \phi_1,\tau'}\tau \\ \Delta(\phi_1) = \tau \to \tau'' \qquad \Phi(\phi_2) = \tau'' \to \tau' \vee \Delta(\phi_2) = \tau'' \to \tau' \qquad \phi' \triangleq \lambda x.\mathsf{let}\ y = \mathsf{dec}\ \phi_1\ x\ \mathsf{in}\ \phi_2\ y\end{array}}{\Delta; \Phi; \Gamma \vdash \mathsf{split}\ e : \Box_{\phi'}\Box_{\phi_1}\tau}\ \text{T-split}$$

Fig. 3. DeCC's type system (selected rules)

will subsume indistinguishability of values of type τ' that gets released. This is formalised in the binary interpretation of the declassification modality which we describe in the next subsection IV-B.

Subtyping: Finally, DeCC also supports subtyping. We write the subtyping judgement as $\Delta; \Phi \vdash \tau <: \tau'$, stating that τ is a subtype of τ' under the two policy contexts Δ and Φ. We only describe the subtyping rules for the classification and the declassification modalities as the rest are totally standard and are included in our mechanised development.

$$\frac{\Delta; \Phi \vdash \tau <: \tau' \qquad \ell \sqsubseteq \ell'}{\Delta; \Phi \vdash \Diamond_{\ell}\tau <: \Diamond_{\ell'}\tau'}\ \text{sub-classify}$$

The subtyping for the classification modality (**sub-classify**) allows raising the level of the associated label, and is covariant in the underlying type. In the current setup, the declassification modality is invariant in all its components, as described in the **sub-declassify** rule.

$$\frac{\Delta(\phi) = \tau \to \tau' \vee \Phi(\phi) = \tau \to \tau'}{\Delta; \Phi \vdash \Box_{\phi,\tau'}\tau <: \Box_{\phi,\tau'}\tau}\ \text{sub-declassify}$$

We can allow for a more liberal subtyping for the declassification modality, but doing so will require additional changes in the typing rules, like T-dec, because otherwise after subtyping the type lookup in the policy context can fail.

B. Semantic model of types

We describe the semantic model of DeCC's types using two kinds of relations, a binary and a unary one. Our model is inspired from the model of CG [5], [6], which is designed for noninterference but not declassification.

At a high level, our binary relations capture the security invariants required to prove indistinguishability of two terms in a compositional manner. Unary relations on the other hand, are used to ensure security of terms that are influenced by secrets, as indistinguishability cannot be proved for such terms. They also provide a way to assert semantic type safety, which we use for policies in the context. Type safety via logical relations is not a requirement, in fact we have built a syntactic type system (essentially a relaxation of the type system we presented earlier) for this purpose. But we avoid getting into those aspects until later (section VII-A).

Unary interpretation: Unary interpretation (described in Fig. 4) of a type defines the set of terms which are semantically in the interpretation of that type. This is defined using two mutually inductive relations, a unary value relation denoted by $\lfloor\cdot\rfloor_V$ and a unary expression relation denoted by $\lfloor\cdot\rfloor_E$. The relations are well founded by induction on types.

The value relation $\lfloor\tau\rfloor_V$ defines the set of values which are in the interpretation of a type τ. For the natural number type $\mathbb{N}$ it says all syntactic inhabitants of $\mathbb{N}$ (written $[\![\mathbb{N}]\!]$) are in the interpretation. For the product type, (v_1, v_2) are in the interpretation of $\tau_1 \times \tau_2$ iff v_1 is in the interpretation of τ_1 and v_2 is in the interpretation of τ_2. The interpretation of the sum type $\tau_1 + \tau_2$ is defined as a disjoint union of the interpretation of τ_1 and τ_2. Next we define the interpretation for the function type, $\lambda x.e$ is in the interpretation of $\tau_1 \to \tau_2$ iff for all values v in the interpretation of the input type τ_1, the body of the function with the v substituted for x, $e[v/x]$, is in the expression interpretation (defined later) of the output type τ_2. The case for the monadic type states that v is in the interpretation at $\Diamond_{\ell}\tau$ if v is a monadic value (i.e. a ret or a bind term), and if v can be forced to a value v' then the resulting value v' must be interpretable at τ using the unary value relation. Notice the label ℓ plays no role whatsoever in the unary interpretation, they are only relevant for security and hence are only relevant in the binary logical relation. The case for the box type states that a boxed value $\Box e$ is in the interpretation of $\Box_{\phi,\tau'}\tau$ if e is in the expression interpretation at type τ and the application of policy ϕ on e is in the interpretation at type τ'. The second clause basically ensures

$$
\begin{array}{lll}
\lfloor \mathbb{N} \rfloor_V & \triangleq & \{n \mid n \in [\![\mathbb{N}]\!]\} \\
\lfloor \tau_1 \times \tau_2 \rfloor_V & \triangleq & \{(v_1, v_2) \mid v_1 \in \lfloor \tau_1 \rfloor_V \wedge v_2 \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 + \tau_2 \rfloor_V & \triangleq & \{\mathsf{inl}\ v \mid v \in \lfloor \tau_1 \rfloor_V\} \cup \{\mathsf{inr}\ v \mid v \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 \to \tau_2 \rfloor_V & \triangleq & \{\lambda x.e \mid \forall v. v \in \lfloor \tau_1 \rfloor_V \implies e[v/x] \in \lfloor \tau_2 \rfloor_E\} \\
\lfloor \Diamond_\ell \tau \rfloor_V & \triangleq & \{v \mid \mathsf{ismonadval}(v) \wedge v \Downarrow^f v' \implies v' \in \lfloor \tau \rfloor_V\} \\
\lfloor \Box_{\phi,\tau'} \tau \rfloor_V & \triangleq & \{\Box\ e \mid e \in \lfloor \tau \rfloor_E \wedge \phi\ e \in \lfloor \tau' \rfloor_E\} \\
 & & \\
\lfloor \tau \rfloor_E & \triangleq & \{e \mid \forall v. e \Downarrow v \implies v \in \lfloor \tau \rfloor_V\} \\
 & & \\
\lfloor \Gamma \rfloor_V & \triangleq & \{\delta \mid dom(\Gamma) \subseteq dom(\delta) \wedge \forall x \in dom(\Gamma). \delta(x) \in \lfloor \Gamma(x) \rfloor_V\}
\end{array}
$$

Fig. 4. Unary interpretation

that the result obtained after application of the policy is also semantically well-typed, it is required for the soundness of our type system.

The expression relation $\lfloor \tau \rfloor_E$ is defined by a set of expressions which are in the interpretation of the type τ. It basically says that e is in the interpretation of τ iff the value v obtained using the pure evaluation of e is in the value interpretation at the same type τ.

Finally, in Fig. 4 we also define a unary substitution relation $\lfloor \Gamma \rfloor_V$ to define semantically well-typed substitutions (represented as a map δ from variables to values) for free variables in Γ. The definition is self explanatory and is defined using the unary value relation.

Binary interpretation: The binary interpretation (described in Fig. 5) defines the conditions required to securely relate (up to indistinguishability wrt an adversary $\mathcal{A}$) two terms of DeCC. As a result binary interpretation is a set of pair of terms (and not a single term like in the unary case). However, like the unary interpretation, the binary interpretation is also defined using two mutually inductive relations, a value relation $\lceil \cdot \rceil_V^{\mathcal{A}}$ and an expression relation $\lceil \cdot \rceil_E^{\mathcal{A}}$. But both of these relations are parameterised by an attacker level $\mathcal{A}$, which is a level in the security lattice. The invariants captured by the binary relations are sufficient to establish a termination-insensitive and indistinguishability-based semantic security criterion (formally defined in section IV-C later).

Like before, we begin with a description of the value relation. The binary value relation $\lceil \tau \rceil_V^{\mathcal{A}}$ defines the set of pair of values which are in the interpretation of a type τ. For the natural number type $\mathbb{N}$ it is defined as the reflexive closure over inhabitants of type $\mathbb{N}$ (written $[\![\mathbb{N}]\!]$). The definition for the products is defined by induction on the components, (v_1, v_2) is related to (v_1', v_2') via $\lceil \tau_1 \times \tau_2 \rceil_V^{\mathcal{A}}$ iff (v_1, v_1') are related via $\lceil \tau_1 \rceil_V^{\mathcal{A}}$ and (v_2, v_2') are related via $\lceil \tau_2 \rceil_V^{\mathcal{A}}$. The interpretation for the sum type $\tau_1 + \tau_2$ is similar to the unary case (defined as the disjoint union of the interpretation at τ_1 and τ_2), but is generalised over a pair of values. The case for the function type is somewhat interesting, it states that two lambda expressions $(\lambda x.e_1, \lambda x.e_2)$ are related at the function type $\tau_1 \to \tau_2$ iff given a pair of related input values (v_1, v_2) at the input type τ_1, we get a pair of related expressions after substitutions $(e_1[v_1/x], e_2[v_2/x])$ at the output type τ_2. Additionally, we also require the two lambda expressions to be in the unary value relations at the function type. This is required for technical reasons (Lemma 2). An explicit inclusion of a similar unary condition is not required at the types discussed before, as those can be obtained directly as an induction hypothesis when inducting on types to prove the following lemma (Lemma 2) relating the unary and binary value relations. A similar property also holds for the expression relations as well, but we elide those details here.

Lemma 2 (Binary value relation subsumes unary value relation)**.**
$(v_1, v_2) \in \lceil \tau \rceil_V^{\mathcal{A}} \implies v_1 \in \lfloor \tau \rfloor_V \wedge v_2 \in \lfloor \tau \rfloor_V$

Next we describe the case for the monadic type $\Diamond_\ell \tau$. It states that if two monadic values v_1 and v_2 can be reduced under forcing semantics to v_1' and v_2', then v_1' and v_2' must be related via the binary value relation at type τ for an adversary who is above ℓ. Otherwise, i.e. when an adversary is not above ℓ, then v_1' and v_2' are only required to be in the unary relation at type τ, as they are secret values and can differ in the two executions. Just like in the case for function type, here also we desire v_1 and v_2 to be in the unary relation at the monadic type. Doing so also provides a semantic justification for the label promotion allowed by the sub-classify rule. Finally we describe the case for the box type. It states that $\Box e_1$ and $\Box e_2$ are related at the $\Box_{\phi,\tau'}\tau$ if e_1 and e_2 are related at τ and $\phi\ e_1$ and $\phi\ e_2$ are related at τ'. Interestingly, adding unary clauses for $\Box e_1$ and $\Box e_2$ (like in the case of function and the monadic types) are problematic for the soundness of split construct. This is because doing so will require us to have the expression under the box to be evaluated, which is not allowed in our semantics. To circumvent this issue, and still obtain the guarantees of Lemma 2, we add the unary clauses to the binary expression relation (described next).

The binary expression relation $\lceil \tau \rceil_E^{\mathcal{A}}$ defines the conditions under which a pair of expressions are related at type τ. It states if (e_1, e_2) are in the interpretation of τ if the value v_1 and v_2 obtained using pure evaluation are related at the value interpretation of the same type τ. We also have the two unary clauses for the expressions e_1 and e_2, for the proof of Lemma 2 at the box type, as mentioned above.

Finally, like in the unary case, Fig. 5 also defines a binary

$$
\begin{array}{lcl}
\lceil \mathbb{N} \rceil_V^{\mathcal{A}} & \triangleq & \{(n,n) \mid n \in [\![\mathbb{N}]\!]\} \\
\lceil \tau_1 \times \tau_2 \rceil_V^{\mathcal{A}} & \triangleq & \{((v_1,v_2),(v_1',v_2')) \mid (v_1,v_1') \in \lceil \tau_1 \rceil_V^{\mathcal{A}} \wedge (v_2,v_2') \in \lceil \tau_2 \rceil_V^{\mathcal{A}}\} \\
\lceil \tau_1 + \tau_2 \rceil_V^{\mathcal{A}} & \triangleq & \{(\mathsf{inl}\ v, \mathsf{inl}\ v') \mid (v,v') \in \lceil \tau_1 \rceil_V^{\mathcal{A}}\} \cup \\
 & & \{(\mathsf{inr}\ v, \mathsf{inr}\ v') \mid (v,v') \in \lceil \tau_2 \rceil_V^{\mathcal{A}}\} \\
\lceil \tau_1 \to \tau_2 \rceil_V^{\mathcal{A}} & \triangleq & \{(\lambda x.e_1, \lambda x.e_2) \mid \\
 & & \big(\forall v_1, v_2.\ (v_1,v_2) \in \lceil \tau_1 \rceil_V^{\mathcal{A}} \implies (e_1[v_1/x], e_2[v_2/x]) \in \lceil \tau_2 \rceil_E^{\mathcal{A}}\big) \wedge \\
 & & \lambda x.e_1 \in \lfloor \tau_1 \to \tau_2 \rfloor_V \wedge \lambda x.e_2 \in \lfloor \tau_1 \to \tau_2 \rfloor_V\} \\
\lceil \Diamond_\ell \tau \rceil_V^{\mathcal{A}} & \triangleq & \{(v_1,v_2) \mid \text{ismonadval}(v_1) \wedge \text{ismonadval}(v_2) \wedge \\
 & & \qquad \big(\ \forall v_1', v_2'.v_1 \Downarrow^f v_1' \wedge v_2 \Downarrow^f v_2' \implies \\
 & & \qquad \begin{cases} (v_1',v_2') \in \lceil \tau \rceil_V^{\mathcal{A}} & \ell \sqsubseteq \mathcal{A} \\ v_1' \in \lfloor \tau \rfloor_V \wedge v_2' \in \lfloor \tau \rfloor_V & \text{otherwise} \end{cases} \\
 & & \qquad \big) \wedge v_1 \in \lfloor \tau \rfloor_V \wedge v_2 \in \lfloor \tau \rfloor_V\} \\
\lceil \Box_{\phi,\tau'} \tau \rceil_V^{\mathcal{A}} & \triangleq & \{(\Box\ e_1, \Box\ e_2) \mid (e_1,e_2) \in \lceil \tau \rceil_E^{\mathcal{A}} \wedge (\phi\ e_1, \phi\ e_2) \in \lceil \tau' \rceil_E^{\mathcal{A}}\} \\
 & & \\
\lceil \tau \rceil_E^{\mathcal{A}} & \triangleq & \{(e_1,e_2) \mid (\forall v_1, v_2.e_1 \Downarrow v_1 \wedge e_2 \Downarrow v_2 \implies (v_1,v_2) \in \lceil \tau \rceil_V^{\mathcal{A}}) \wedge \\
 & & \qquad e_1 \in \lfloor \tau \rfloor_E \wedge e_2 \in \lfloor \tau \rfloor_E\} \\
 & & \\
\lceil \Gamma \rceil_V^{\mathcal{A}} & \triangleq & \{\gamma \mid dom(\Gamma) \subseteq dom(\gamma) \wedge \forall x \in dom(\Gamma).(\pi_1(\gamma(x)), \pi_2(\gamma(x))) \in \lceil \Gamma(x) \rceil_V^{\mathcal{A}}\}
\end{array}
$$

Fig. 5. Binary interpretation

substitution relation $\lceil \Gamma \rceil_V^{\mathcal{A}}$ to define a pair of semantically well typed substitutions for free variables in Γ. It says γ (a map from variables in Γ to a pair of values) is in $\lceil \Gamma \rceil_V^{\mathcal{A}}$ iff the conditions specified in the definition are met. The conditions are similar to those of the unary substitution relation but is defined over pairs of values (represented using projection functions $\pi_1(\cdot)$ and $\pi_2(\cdot)$) and hence is defined using the binary value relation.

C. Relaxed semantic declassification and soundness

Having described the logical relations we now move to defining our security criterion, which we call as Relaxed Semantic Declassification (RSD for short) and show that our binary logical relation provides that guarantee for well-typed programs.

We begin by defining well-formedness and semantic assumptions for the policy contexts. The assumptions for the policies in the context Δ are defined wrt an adversary $\mathcal{A}$ (Definition 3), it basically requires that for all policy and type pairs (ϕ, τ) in Δ three conditions hold: 1) the policy ϕ is a closed term, we do not want the declassification policies to depend on free variables 2) τ is a function type of the form $\tau_1 \to \tau_2$ for some τ_1 and τ_2 and 3) the policy ϕ is related to itself via the binary value relation at the type $\tau_1 \to \tau_2$, i.e. $(\phi, \phi) \in \lceil \tau_1 \to \tau_2 \rceil_V^{\mathcal{A}}$. Intuitively, the conditions formally specify that Δ only contains closed declassification functions which are semantically secure on all inputs. We also get semantic well typing of all the policies in Δ, as a corollary of Lemma 2 described earlier.

Definition 3 (Policy assumptions for Δ)**.**
PA $\Delta\ \mathcal{A} \triangleq \forall (\phi, \tau) \in dom(\Delta).\ closed\ \phi\ \wedge$
$\exists \tau_1, \tau_2.\ \tau = \tau_1 \to \tau_2 \wedge (\phi, \phi) \in \lceil \tau_1 \to \tau_2 \rceil_V^{\mathcal{A}}$

We also define a similar well-formedness and semantic conditions for Φ (Definition 4) but it only demands inclusion of the policy in the unary value relation for reasons of type safety (remember the policies in Φ can be interfering). The semantic typing via our unary logical relation is only a temporary requirement to avoid getting into trivial type safety issues and ease of presentation, we will lift this requirement and derive a simple type system for asserting the type safety of policies later in section VII-A.

Definition 4 (Policy assumptions for Φ)**.**
PA $\Phi \triangleq \forall (\phi, \tau) \in dom(\Phi).\ closed\ \phi\ \wedge$
$\exists \tau_1, \tau_2.\ \tau = \tau_1 \to \tau_2 \wedge \phi \in \lfloor \tau_1 \to \tau_2 \rfloor_V$

We now have all the ingredients to define RSD (Definition 5). RSD defines a termination-insensitive and indistinguishability-based semantic security criterion for an open term e of type τ with free variables in Γ where e can potentially declassify information using policies from either of the policy contexts, Δ or Φ. The guarantee that RSD provides is that, for any adversary $\mathcal{A}$ if the two policy contexts satisfy the assumptions as defined in Definitions 3 and 4, then providing indistinguishable substitutions from γ for free variables in the two executions will lead to indistinguishable outputs from e closed with those substitutions (the substitutions for the two executions are denoted using $\gamma\downarrow_1$ and $\gamma\downarrow_2$). The indistinguishability is formalised in the binary logical relations (as defined earlier).

Definition 5 (RSD)**.**
$\mathsf{RSD}(e, \tau, \Gamma, \Delta, \Phi) \triangleq \forall \mathcal{A}.\ PA\ \Delta\ \mathcal{A} \wedge PA\ \Phi \implies$
$\forall \gamma \in \lceil \Gamma \rceil_V^{\mathcal{A}}.\ (e\ \gamma\downarrow_1, e\ \gamma\downarrow_2) \in \lceil \tau \rceil_E^{\mathcal{A}}$

Next we show that programs that are well typed under DeCC's typing rules satisfy RSD. This result is stated as the fundamental theorem of our binary logical relation, Theorem 6. We prove this theorem by induction on the typing rules, the theorem makes use of a corresponding unary version of the fundamental theorem (Theorem 7) to complete the proof where binary and unary logical relations interact like at the monadic and function types, and also in the expression relation. The unary fundamental theorem also extends the semantic type-safety guarantee to all the well typed programs (in addition to the policies which enjoy semantic type safety by virtue of directly being in the unary relation).

Theorem 6 (Binary fundamental theorem)**.**
$\Delta; \Phi; \Gamma \vdash e : \tau \implies \mathsf{RSD}(e, \tau, \Gamma, \Delta, \Phi)$

Theorem 7 (Unary fundamental theorem)**.**
$\Delta; \Phi; \Gamma \vdash e : \tau \wedge \delta \in \lfloor \Gamma \rfloor_V \wedge \mathit{PA}\ \Delta\ \mathcal{A} \wedge \mathit{PA}\ \Phi$
$\implies e\ \delta \in \lfloor \tau \rfloor_E$

One more ingredient required to finish the proofs of both Theorem 6 and 7, is proving the soundness of our subtyping relation. We prove that our subtyping relation is sound wrt both the unary and binary value relations (Lemma 8). Similar results are also obtained for the expression relations, but we elide those details here.

Lemma 8 (Unary and binary subtyping lemmas)**.**
$\forall \Delta, \Phi, \mathcal{A}.$
$\mathit{PA}\ \Delta\ \mathcal{A} \wedge \mathit{PA}\ \Phi \wedge \Delta; \Phi \vdash \tau <: \tau' \implies$
1) $v \in \lfloor \tau \rfloor_V \implies v \in \lfloor \tau' \rfloor_V$
2) $(v_1, v_2) \in \lfloor \tau \rfloor_V \implies (v_1, v_2) \in \lfloor \tau' \rfloor_V$

Finally, one can immediately derive the following top-level soundness result (Theorem 9) as a corollary of the binary fundamental theorem. The theorem states that given policy contexts Δ and Φ satisfying the policy assumptions stated earlier, and given a declassification policy $\phi : \Diamond_\top \mathbb{N} \to \Diamond_\bot \mathbb{N}$ from either of the policy contexts, the following holds: declassification of secret boxed values of type $\Box_\phi \Diamond_\top \mathbb{N}$ in the two runs would produce indistinguishable results.

Theorem 9 (Soundness)**.**
$\forall \Delta, \Phi, \phi.$
PA $\Delta\ \mathcal{A}$ *and PA* Φ,
$\big(\phi : \Diamond_\top \mathbb{N} \to \Diamond_\bot \mathbb{N} \in \Delta$ *or*
$\phi : \Diamond_\top \mathbb{N} \to \Diamond_\bot \mathbb{N} \in \Phi\big)$
If
$\Delta; \Phi; x : \Box_\phi \Diamond_\top \mathbb{N} \vdash \mathsf{dec}\ \phi\ x : \Diamond_\bot \mathbb{N}$
$\Delta; \Phi; . \vdash \Box v_1 : \Box_\phi\ \Diamond_\top \mathbb{N}$
$\Delta; \Phi; . \vdash \Box v_2 : \Box_\phi\ \Diamond_\top \mathbb{N}$
$(\mathsf{dec}\ \phi\ x)[\Box v_1 / x] \Downarrow \_ \Downarrow^f v_1'$
$(\mathsf{dec}\ \phi\ x)[\Box v_2 / x] \Downarrow \_ \Downarrow^f v_2'$
then
$v_1' = v_2'$

V. Meta theory

In this section we explore the meta-theoretical aspects of the two modalities of DeCC. While part of this investigation is purely of theoretical interest, we also describe some of their practical implications pertaining to information flow.

A. Box is a comonad

We begin by exploring the following question: if the classification modality has a monadic nature [4], does the declassification modality exhibit some comonadic aspects? It turns out the general ϕ graded box does not form a full comonad, but its specialisation the id graded (or equivalently the ungraded) box does.

We begin by adding a specific version of fmap for our declassification modality to the calculus. Our fmap is a generalisation of the standard fmap, it takes three arguments: A mapping function f and an expression to be mapped over e as usual. But additionally, it also takes a new function ϕ as a third argument. The function ϕ specifies a declassification policy that can be used to inspect the boxed result after mapping f over e. Typing of fmap (**T-fmap** rule) requires e to be an expression of the type $\Box_{\phi_1} \tau_1$ and f to be a mapping function of type $\tau_1 \to \tau_2$. However, mapping f over e can only give us a result of type $\Box_{\phi_1} \tau_2$. This is a problem as ϕ_1 expects its input to be of type τ_1, as a result we have to change the policy on the result. This is why we choose ϕ, a universally secure semantic policy of the type $\tau_2 \to \tau_3$ from Δ. Picking a policy from Φ will not be sound as policies in Φ don't have the required security invariants on them. We have proved the soundness of this rule by proving that it satisfies the binary and unary fundamental theorems (Theorems 6 and 7) as stated before.

$$\frac{\Delta; \Phi; \Gamma \vdash e : \Box_{\phi_1} \tau_1 \qquad \Delta; \Phi; \Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Delta(\phi) = \tau_2 \to \tau_3}{\Delta; \Phi; \Gamma \vdash \mathsf{fmap}\ f\ e\ \phi : \Box_\phi \tau_2}\ \text{T-fmap}$$

The evaluation rule, **E-fmap**, describes the semantics of our fmap. Mapping over a boxed value is a pure operation and hence is evaluated using the pure reduction. The rule says, if e evaluates purely to a boxed value $\Box\ e'$ and the application of f on e' results in v, then the final result we obtain is $\Box\ v$. As is clear from the rule, the policy ϕ is only relevant in types and does not play any role in the semantics.

$$\frac{e \Downarrow \Box\ e' \qquad f\ e' \Downarrow v}{\mathsf{fmap}\ f\ e\ \phi \Downarrow \Box\ v}\ \text{E-fmap}$$

Next we derive three properties of $\Box_\phi \tau$ (Fig. 6) wrt our unary expression relation, and use those to show the comonadic nature of $\Box_{id} \tau$. The first property P1 says that for any ϕ and τ, the sequential composition of coret after cojoin is in $\lfloor \Box_\phi \tau_1 \to \Box_\phi \tau_1 \rfloor_E$ iff the identity function is in $\lfloor \Box_\phi \tau_1 \to \Box_\phi \tau_1 \rfloor_E$. The second property P2 shows a similar result but for a term involving fmap. In particular, it says that $\lambda x.\mathsf{fmap}\ \mathsf{coret}\ (\mathsf{cojoin}\ x)\ \phi$ is an inhabitant of $\lfloor \Box_\phi \tau \to \Box_\phi \tau \rfloor_E$ iff the identity is its inhabitant too. Finally, the third property P3 is a result relating the membership of $\lambda x.\mathsf{cojoin}(\mathsf{cojoin}\ x)$ and $\lambda x.\mathsf{fmap}\ \mathsf{cojoin}\ (\mathsf{cojoin}\ x)\ id$ in $\lfloor \Box_\phi \tau \to (\Box_{id}(\Box_{id} \Box_\phi \tau)) \rfloor_E$.

Since all these properties are proved for a general ϕ they can be also be instantiated for id, resulting in proving the three comonad laws for $\Box_{id}\tau$. At this point an avid reader might ask: Does the graded version of the box modality $\Box_{\phi}\tau$ also forms a comonad? The answer to this question is unfortunately negative. While properties P1 and P2 together correspond to counitality axioms for ϕ graded box, but the property P3 does not imply its coassociativity axiom [16]. This is because property P3 only holds for the type $\Box_{\phi}\tau \rightarrow (\Box_{id}(\Box_{id}\Box_{\phi}\tau))$, i.e. with id (and not ϕ) on the outer two boxes of the return type. This is, however, suggestive of some richer structure that warrants further investigation into the categorical model of DeCC. This is an interesting direction of future work. The existing efforts on the categorical interpretation of information flow properties (like [4], [17], [18]) have not considered properties like relaxed semantic declassification at all.

B. Interaction between the modalities

In multi-modal type theories (like DeCC) often we are interested in studying interactions between the different modalities. In DeCC we study such interactions between the classification and the declassification modalities via distributive laws [19], [20] described as coercions between $\Diamond_{\ell}(\Box_{\phi}\tau)$ and $\Box_{\phi'}(\Diamond_{\ell}\tau)$ for some relation between ϕ and ϕ'. Such laws are not only relevant meta theoretically [19], [20], but they also provide a modal justification for contextual declassification and reclassification, as we describe next.

First we describe a coercion from $\Diamond_{\ell}(\Box_{\phi}A)$ to $\Box_{\phi'}(\Diamond_{\ell}A)$. We begin by motivating this coercion from the perspective of information flow. Imagine a principal ℓ who creates a declassifiable secret of type $\Box_{\phi}(\Diamond_{\ell}\tau)$ (where ϕ is the declassification policy of type $\Diamond_{\ell}\tau \rightarrow \Diamond_{\ell'}\tau'$ and $\ell' \sqsubset \ell$). However, since a term of this box type is created by the principal ℓ, the whole term must have a type with a label ℓ on it, i.e. $\Diamond_{\ell}(\Box_{\phi}(\Diamond_{\ell}\tau))$. The outer most label ℓ denotes the context label (of the principal) in which a term of boxed type was created. Note that, in DeCC we can only extract out a term of label ℓ or higher (but not ℓ') from a term of this type, despite trying to use the intended declassification policy via the dec construct. For instance, if e is term of the given type $\Diamond_{\ell}(\Box_{\phi}(\Diamond_{\ell}\tau))$, we can try to attempt a declassification using $\mathsf{bind}\ x = e \ \mathsf{in}\ (\mathsf{dec}\ \phi\ x)$. But this bind expression can only be given a type with a label ℓ or higher (because of how the typing of bind works), but not ℓ' as was intended upon declassification . However, if we have a way to distribute the diamond over the box and obtain a coercion (we call this coercion dist, short for distributive law) from $\Diamond_{\ell}(\Box_{\phi}(\Diamond_{\ell}\tau))$ to $\Box_{\phi}(\Diamond_{\ell}\tau)$[5], then we will be able to perform the desired declassification simply as, $\mathsf{dec}\ \phi\ (\mathsf{dist}\ e)$. Unfortunately, we cannot write a function directly in DeCC which can perform this coercion (from $\Diamond_{\ell}(\Box_{\phi}(\Diamond_{\ell}\tau))$ to $\Box_{\phi}(\Diamond_{\ell}\tau)$) as there is no way to extract a term out of a monad in general. To enable this coercion we add dist as a primitive in the language and prove it sound wrt the semantic model described earlier. The typing rule and the semantics of dist are described in Fig. 7. In summary, the first distributive law provides a way to enable contextual declassification in DeCC, without which such a declassification would not be possible. Understanding generalisation of this law to work with arbitrary types could be an interesting problem for future work.

The second distributive law i.e. a coercion from $\Box_{\phi}(\Diamond_{\ell}\tau)$ (where ϕ is a declassification policy of type $\Diamond_{\ell}\tau \rightarrow \Diamond_{\ell'}\tau'$ and $\ell' \sqsubseteq \ell$) to $\Diamond_{\ell}(\Box_{\phi'}\tau)$ is also possible when $\phi' \triangleq \lambda x.\phi\ (\mathsf{ret}\ x)$ is defined as the composition of ϕ after ret. In fact, this direction of coercion is possible for any arbitrary type τ and can be directly encoded as a well-typed DeCC term. Like the first law, this law also requires that ϕ be a policy that is universally secure on all inputs, i.e. ϕ must exist in Δ. The function for such a coercion is as follows, $\lambda x.\mathsf{bind}\ y = \mathsf{coret}\ x \ \mathsf{in}\ \mathsf{ret}\ (\mathsf{inject}\ y)$. It takes a term of type $\Box_{\phi}(\Diamond_{\ell}\tau)$ extracts the term under the box using coret, then binds it to the variable y of type τ. After that in the continuation expression of bind, the y is injected back into a box controlled by the policy ϕ' and returned as a result giving back a term of type $\Diamond_{\ell}(\Box_{\phi'}\tau)$ as desired. While the first law provided a modal basis of contextual declassification, the second law can be seen as providing a modal justification of contextual reclassification. This is because while the input type (of the second coercion) can enable an extraction of a term protected at label ℓ' (via declassification using the dec construct), the result type (of the second coercion) can only provide us a term protected at label ℓ because of the outer contextual label on the monad, thereby reclassifying the declassified output to a higher protection.

VI. Examples

In this section we describe some examples to demonstrate the applicability of DeCC. We focus on describing the declassification aspects using our modal types. Full typing derivations of all the examples can be found in the appendix below.

A. Semantically secure program on all inputs

Our first example is a program that is semantically secure universally (all secret inputs) but cannot be accepted by a standard information-flow type systems like DCC [4] or CG [5], [6].

Consider the following function ϕ which takes a secret number with some label ℓ ($\neq \bot$), unpacks the secret using the bind and multiplies the unpacked natural number by 4, then computes a remainder by dividing it by 2, and finally adds 10 to it.

$$\phi : \Diamond_{\ell}\mathbb{N} \rightarrow \Diamond_{\bot}\mathbb{N}$$
$$\phi \triangleq \lambda x.\mathsf{bind}\ y = x \ \mathsf{in}\ \mathsf{ret}\ ((4 * y)\%2 + 10)$$

Irrespective of the secret input that ϕ gets applied to, it will always produce the number 10 as its output. Consequently, it is easy to see that $(\phi, \phi) \in \lceil \Diamond_{\ell}\mathbb{N} \rightarrow \Diamond_{\bot}\mathbb{N} \rceil_{V}^{\mathcal{A}}$, for any attacker $\mathcal{A} \sqsupseteq \bot$. As a result the function $f \triangleq \lambda x.\ \mathsf{dec}\ \phi\ x$ can be type checked in DeCC under the typing derivation, $[(\phi : \Diamond_{\ell}\mathbb{N} \rightarrow \Diamond_{\bot}\mathbb{N})]; \cdot; \cdot \vdash f : \Diamond_{\ell}\mathbb{N} \rightarrow \Diamond_{\bot}\mathbb{N}$.

[5]Technically from the perspective of distributive laws, we require a coercion from $\Diamond_{\ell}(\Box_{\phi}(\Diamond_{\ell}\tau))$ to $\Box_{\phi'}(\Diamond_{\ell}\Diamond_{\ell}\tau)$, where $\phi' = \lambda x.\phi\ (\mathsf{join}\ x)$. Once we have dist in the language, even this coercion can be obtained as $\lambda x.\mathsf{fmap}\ (\mathsf{dist}\ x)\ \mathsf{ret}\ \phi'$

Theorem 10 (P1).
$\forall \phi, \tau.\ \lambda x.\mathsf{coret}(\mathsf{cojoin}\ x) \in \lfloor \Box_\phi \tau \to \Box_\phi \tau \rfloor_E \iff id \in \lfloor \Box_\phi \tau \to \Box_\phi \tau \rfloor_E$

Theorem 11 (P2).
$\forall \phi, \tau.\ \lambda x.\mathsf{fmap}\ \mathsf{coret}\ (\mathsf{cojoin}\ x)\ \phi \in \lfloor \Box_\phi \tau \to \Box_\phi \tau \rfloor_E \iff id \in \lfloor \Box_\phi \tau \to \Box_\phi \tau \rfloor_E$

Theorem 12 (P3).
$\forall \phi, \tau.\ \lambda x.\mathsf{cojoin}(\mathsf{cojoin}\ x) \in \lfloor \Box_\phi \tau \to (\Box_{id}(\Box_{id}\Box_\phi \tau)) \rfloor_E \iff \lambda x.\mathsf{fmap}\ \mathsf{cojoin}\ (\mathsf{cojoin}\ x)\ id \in \lfloor \Box_\phi \tau \to (\Box_{id}(\Box_{id}\Box_\phi \tau)) \rfloor_E$

Fig. 6. Properties of the graded box modality

$$\frac{\Delta; \Phi; \Gamma \vdash e : \Diamond_\ell(\Box_\phi(\Diamond_\ell \tau)) \qquad \Delta(\phi) = \Diamond_\ell \tau \to \Diamond_{\ell'} \tau' \qquad \ell' \sqsubseteq \ell}{\Delta; \Phi; \Gamma \vdash \mathsf{dist}\ e : \Box_\phi(\Diamond_\ell \tau)}\ \text{T-dist} \qquad \frac{e \Downarrow \mathsf{ret}(\Box e')}{\mathsf{dist}\ e \Downarrow \Box e'}\ \text{E-dist}$$

Fig. 7. Typing and evaluation rules for dist

B. *Semantically secure program on some inputs*

Our next example is a program that is semantically secure on some inputs (but not universally).

Consider the following function ϕ which takes a secret pair of numbers (of some label $\ell \neq \bot$), unpacks the secret using the bind and returns the addition of its two components.

$\phi : \Diamond_\ell(\mathbb{N} \times \mathbb{N}) \to \Diamond_\bot \mathbb{N}$
$\phi \triangleq \lambda x.\mathsf{bind}\ y = x\ \mathsf{in}\ \mathsf{ret}\ (\mathsf{fst}\ y + \mathsf{snd}\ y)$

Clearly ϕ is not universally secure, but can produce indistinguishable result on some inputs like when the inputs are permuted, for instance, $(2, 4)$ and $(4, 2)$. For such inputs, we should be able to guarantee indistinguishability of results obtained on application of ϕ. Technically, this means if we have a variable x of type $\Box_\phi \Diamond_\ell(\mathbb{N} \times \mathbb{N})$ in the context (say Γ), and a binary substitution $\gamma \in \lceil \Gamma \rceil_V^{\mathcal{A}}$, for any attacker $\mathcal{A} \sqsupseteq \bot$, then we should be able to allow information to be released via an expression like $e \triangleq \mathsf{dec}\ \phi\ x : \Diamond_\bot \mathbb{N}$. The indistinguishability of the declassified output really comes from the fact that the substitutions/inputs for x in the two runs (coming from γ) are in the binary value relation at the type $\Box_\phi \Diamond_\ell(\mathbb{N} \times \mathbb{N})$. This would basically ensure that ϕ is non distinguishing on the supplied inputs. This is why $\mathsf{RSD}(e, \Diamond_\bot \mathbb{N}, \Gamma, \Delta, \Phi)$ is satisfied and declassification via the expression e can be allowed. Here $\Phi \triangleq [\phi : \Diamond_\ell(\mathbb{N} \times \mathbb{N}) \to \Diamond_\bot \mathbb{N}]$ and Δ is empty. Consequently, the expression e can be type checked as $\cdot; \Phi; \Gamma \vdash e : \Diamond_\bot \mathbb{N}$.

C. *A state machine as a policy*

Our next example describes an encoding of state machine as a policy which allows information to be released in a step-wise manner depending the state of the system.

To understand this, consider the function f below. It takes in a boxed secret and declassifies it in two steps. In the first step, it declassifies the input using a policy $\phi_1 : \Diamond_\top \mathbb{N} \to \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Box_{\phi_2} \Diamond_\top \mathbb{N})$ which itself returns a boxed secret of type $\Box_{\phi_2} \Diamond_\top \mathbb{N}$, along with a public number. Basically, ϕ_1 makes a part of the secret publicly available in the first step and controls the future inspection of the remaining secret by returning a boxed value which can only be inspected by $\phi_2 : \Diamond_\top \mathbb{N} \to \Diamond_\bot \mathbb{N}$. The second step of declassification with ϕ_2 happens next in f and finally the two public parts of the input secret are returned as the output. The function f can be type checked as $\Delta; \Phi; \cdot \vdash f : \Box_{\phi_1}(\Diamond_\top \mathbb{N}) \to \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Diamond_\bot \mathbb{N})$, where Δ is described below and Φ is empty.

```
$f : \Box_{\phi_1}(\Diamond_\top \mathbb{N}) \to \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Diamond_\bot \mathbb{N})$
$f \triangleq \lambda x.$ bind $y =$ dec $\phi_1\ x$ in
        let $z =$ dec $\phi_2$ (snd $y$) in ret (fst $y, z$)
where
$\Delta = [\phi_1 : \Diamond_\top \mathbb{N} \to \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Box_{\phi_2} \Diamond_\top \mathbb{N}),$
      $\phi_2 : \Diamond_\top \mathbb{N} \to \Diamond_\bot \mathbb{N}]$
```

D. *Partial and conditional declassification*

Our final program is an example of a conditional declassification using coret and split, as described in the function f below. The function f takes two arguments, a boxed secret number (of type $\Box_{\phi_2 \cdot \phi_1} \Diamond_\ell \mathbb{N}$) which is boxed under a composite policy $\phi_2 \cdot \phi_1$ and a unlabelled boolean flag (encoded using the sum type in a standard way), and returns a natural number labelled with ℓ' s.t. $\bot \sqsubseteq \ell' \sqsubseteq \ell$. Based on the boolean flag b the function f either completely declassifies the secret in the left branch using the complete policy $\phi_2 \cdot \phi_1$, or partially declassifies it in the right branch. In the right branch, the function splits the boxed secret into a secret which is nested under two boxes using the split construct, split $x : \Box_{\phi'}(\Box_{\phi_1} \Diamond_\ell \mathbb{N})$, where ϕ' is an outer policy simulating the composition ϕ_2 after ϕ_1 (as described earlier in the typing of the split construct). Then it strips off the outer box (and hence the possibility of declassification using the outer policy ϕ_2) using the coret construct and binds the result to a variable y which is of type $\Box_{\phi_1} \Diamond_\ell \mathbb{N}$. Finally, it declassifies only using the inner policy ϕ_1 via the dec construct, dec $\phi_1\ y$. The function f can be type checked as $\Delta; \cdot; \cdot \vdash f : \Box_{\phi_2 \cdot \phi_1}(\Diamond_\ell \mathbb{N}) \to (\mathbb{N} + \mathbb{N}) \to \Diamond_{\ell'} \mathbb{N}$, where Δ is described below.

```
$f : \Box_{\phi_2 \cdot \phi_1}(\Diamond_\ell \mathbb{N}) \to (\mathbb{N} + \mathbb{N}) \to \Diamond_{\ell'} \mathbb{N}$
$f \triangleq \lambda\ x\ b.$
      case $b$ of
      _. dec $\phi\ x$
```

$$\_.\ \mathsf{let}\ y = \mathsf{coret}\ (\mathsf{split}\ x)\ \mathsf{in}\ \mathsf{dec}\ \phi_1\ y$$

$$\Delta = [\phi_1 : \Diamond_\ell \mathbb{N} \to \Diamond_{\ell'} \mathbb{N}, \phi_2 : \Diamond_{\ell'} \mathbb{N} \to \Diamond_\perp \mathbb{N}]$$

VII. Discussion

A. Syntactic type checking of policies

As mentioned earlier, policies in the context Φ can be interfering and are only required to be well typed, which we specified so far using the semantic typing provided by our unary logical relation. However, use of logical relations for type safety is not necessary, type safety of policy code can be enforced using a simple non-IFC syntactic system (typing judgement given by $\Gamma \Vdash e : \tau$). This type system can be thought of as a weaker version of the type system presented earlier in Fig. 3, which only cares about type safety and not security. As a result, the typing of bind no longer requires a constraint on labels anymore. All the other rules are as expected and can be found in our HOL development [13]. We also describe some selected rules in the appendix below.

$$\frac{\Gamma \Vdash e : \Diamond_\ell \tau \qquad \Gamma, x : \tau \Vdash e' : \Diamond_{\ell'} \tau'}{\Gamma \Vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : \Diamond_{\ell'} \tau'}\ \text{T-bind}$$

We prove that well typed terms under this type system are also semantically well typed (i.e. are in the unary logical relation) by proving the following theorem.

Theorem 13 (Semantic type safety)**.**
$\Gamma \Vdash e : \tau \wedge \delta \in \lfloor \Gamma \rfloor_V \implies e\ \delta \in \lfloor \tau \rfloor_E$

As a result, we can change the definition of the policy assumptions predicate for the unary policy context (new Definition 14) to use the syntactic instead of the semantic typing (contrast this to the previous Definition 4 which requires a semantic argument).

Definition 14 (Policy assumptions for Φ modified)**.**
$PA\ \Phi \triangleq \forall(\phi, \tau) \in dom(\Phi).\ closed\ \phi \wedge$
$\exists \tau_1, \tau_2.\ \tau = \tau_1 \to \tau_2 \wedge \cdot \Vdash \phi : \tau_1 \to \tau_2$

The fundamental theorems and the soundness of DeCC with this change can be proved as expected and can be found in our accompanying HOL artefact [13].

B. Call-by-name semantics

Since DCC [4] was studied for noninterference in a Call-By-Name (CBN) setting, we were curious to understand if our results can be obtained for DeCC with a CBN semantics too. It turns out, they can be. In fact, we built an alternate version of DeCC (which we refer to as DeCC$_N$) with minimal (and only expected) changes pertaining to the CBN evaluation and showed similar results. We do not describe those changes here but the complete development with proofs can be found in our accompanying HOL artefacts [13]. However, we would like to point out that the CBN version has a slightly different meta theory as far as the properties of the declassification modality are concerned. Out of the three properties that we described in Fig. 6, while $P1$ and $P3$ still hold for DeCC$_N$, the property $P2$ for DeCC$_N$ (Theorem 15) is somewhat different.

Theorem 15 (P2 for the declassification modality of DeCC$_N$)**.**
$\forall \tau.\ \lambda x.\mathsf{fmap\ coret}\ (\mathsf{cojoin}\ x)\ id \in \lfloor \Box_{id}\tau \to \Box_{id}\tau \rfloor_E \iff$
$id \in \lfloor \Box_{id}\tau \to \Box_{id}\tau \rfloor_E$

In particular, $P2$ for DeCC$_N$ only holds for the id-graded declassification modality. This is still sufficient to show that the ungraded box (i.e. the id graded box) is a comonad and its policy graded version provides guarantees wrt RSD like our Call-By-Value (CBV) version. However, it is still an open question to understand the implications of the extra generality that the property $P2$ of the CBV version offers.

C. Other language features

DeCC has been deliberately kept as a minimal core calculus. This is done to focus our attention on the modal aspects of semantic declassification in a higher-order setting. Addition of other language features like fixpoints (like in [4]) and polymorphism (like in [21]) can be done but will require adding step indices [22] to the model. Addition of higher-order mutable state like in CG [5] will involve more careful changes. In addition to step indices in the semantic model, we can also anticipate changes in the proof theory to track and restrict updates to locations involved in declassified terms like in delimited release. Extending DeCC to accommodate such language features is future work.

VIII. Related work

In this section we describe prior work that is closely related to DeCC both from both type-theoretic and declassification perspective.

Related work on graded modal types for IFC: The use of graded modal types for IFC is fairly limited in prior work.

DCC [4] is the first paper in our knowledge that shows use of graded monads for labelling and tracking of information for a core functional calculus. As mentioned earlier, DCC is one of the building blocks for our work. Our work inherits DCC's monadic approach for tracking of information flow using security labels. But additionally in our work we also introduce a new graded modality for the purpose of semantic declassification, which is missing from DCC. Also, while DCC uses noninterference as its soundness criterion, for us RSD serves that purpose.

Another building block for our work has been the CG [5], [6] calculus. In particular, DeCC's semantic type theory is heavily inspired from that of CG. This is reflected in the choice of unary and binary logical relations to build our semantic model. However, there are several differences between CG and DeCC both in the proof theoretic and semantic aspects of the type theory. First, CG uses a doubly graded monad (to track information from read and write effects on state), as opposed to the DeCC's monad which only has one grade (to track information on the term inside the monad). As a result, CG's semantic model (both unary and binary relations) differ considerably from ours. Another difference from CG is that

CG uses two different modalities for labelling and tracking of labels, while for DeCC the diamond modality handles both. CG also doesn't have any counterpart for our box modality and also cannot reason about semantically secure code, this results in substantial differences in both the syntactic and semantic aspects of the two type theories. Finally, logical relations in CG are only required for proving soundness (which is wrt noninterference), while for DeCC logical relations also play a crucial role for enabling semantic declassification in addition to the proof of soundness (which is wrt RSD).

SLIO [23] is another system similar to CG. It uses a different but still a doubly graded monad for enforcing noninterference over programs with state. But yet, it was shown to be equally expressive to CG [6]. The interpretation of SLIO types is described using its predecessor LIO [24] which is a dynamic enforcement system for IFC. Our approach towards interpretation of types is very different from SLIO, as we do not interpret our types using LIO but instead use logical relations defined over the pure and forcing semantics which are free from labels.

Finally, there is also some work on the "who" dimension of declassification like [25], [26] which uses a *says* modality to annotate types with labels containing both confidentiality and integrity components. The soundness property in that setting is a variant of robustness (and endorsement) against adversarial code modelled as evaluation context. In our current development we have only targeted declassification via semantic policies, which is often categorised under the "what" dimension of declassification [8]. Extending DeCC to reason about variants of robustness (and endorsement) would be an interesting direction of future work.

Related work on declassification: As mentioned earlier, this work builds upon two classical notions of "what" declassification namely delimited release [9] and relaxed noninterference [10]. We have already elaborated extensively on the difference from these approaches in the introduction of the paper. So, now we compare our work to other ideas in this space.

Closest to our work are the approaches which have tried to extend relaxed noninterference style policies with indistinguishability-based soundness. One such approach is the work by Ngo et al. [27] which leverages parametricity theorem [27] to achieve this. The goal there is to use a standard non-IFC type system coupled with the abstraction theorem to reason about security in the style of theorems for free [28]. There are several differences between their approach and ours. At the outset, their approach rests upon ideas from parametricity while ours on modal logic, this leads to development of very different proof theory and semantic foundations. Also, in our approach we maintain a clear distinction between security labels and policies that can downgrade those labels. This distinction does not exist in their approach, in their system only policy functions exist which determine what can be declassified, but not to what level. This design decision leads to considerable differences in the semantic model. In particular, their logical relations are not label aware and they only consider a binary logical relation but not unary. Also, in DeCC we incorporate ways to reason about semantically secure policies, which allows us to build operations like inject and split. It's not immediately clear if the effect of such operations can be simulated in their system. In particular, DeCC allows for conditional and partial release via a combination of coret and split as demonstrated using an example in Section VI-D, its unclear if such use cases can be modelled in their system without requiring further modifications. We believe answering such questions would require investigating relative expressiveness of these systems in a more formal sense, which could be an interesting direction for future work. There is also work on use of type-based abstraction from existing languages to encode relaxed noninterference policies in an object-oriented setting [29]. This is again a very different approach from the one that we take in DeCC.

On the use of logical relations in the declassification setting, there has also been a recent work [30] studying similar logical relation models for the "where" dimension of declassification (following Paralocks [31]) in a higher-order setting. However, the approach taken in [30] uses a type and effect system in the style of what is called as fine-grained tracking approach for secure information flow, as used in FG [5], [6] and FlowCaml [12], for instance. We believe similar results can be obtained in a modal setup of DeCC but it will require adding constructs for opening and closing locks, and tracking their state at the type level.

Finally, state machines based policies for declassification have been studied in the context of reactive information flow labels [32]. However, such policies were studied wrt piecewise noninterference [32] which only ensure indistinguishability over subtraces and not extensionally like RSD. Also, enforcement of such policies was studied in an imperative setting with no higher-order functions and using a non-modal approach.

IX. Conclusion

In this work we presented DeCC, a graded-modal-type-theoretic framework for reasoning about semantic declassification in higher-order functional programs. DeCC builds on prior work which uses graded-modal types for reasoning about noninterference namely DCC and CG. DeCC uses an existing graded monad for classification of information which is inherited from DCC, but introduces a new graded modality for declassification. We describe interaction between the two graded modalities, and also conditions under which our new graded modality forms a comonad. We build logical relations model for the types of DeCC and use that to prove Relaxed Semantic Declassification, a security criterion which is inspired from delimited release and relaxed noninterference. Finally, we show encoding of several declassification scenarios within DeCC and provide machine verified results in HOL4 as accompanying artefacts.

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful comments and feedback. Vineet Rajani was supported in part by the EPSRC grants EP/Y003535/1 and EP/X015076/1, and by the ARIA's Safeguarded AI programme. Hrutvik Kanabar was supported in part by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS).

References

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, 2003. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121

[2] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, 1982, pp. 11–20. [Online]. Available: https://doi.org/10.1109/SP.1982.10014

[3] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, 2010. [Online]. Available: https://doi.org/10.3233/JCS-2009-0393

[4] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, 1999, pp. 147–160. [Online]. Available: https://doi.org/10.1145/292540.292555

[5] V. Rajani and D. Garg, "Types for information flow control: Labeling granularity and semantic models," in *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 2018, pp. 233–246. [Online]. Available: https://doi.org/10.1109/CSF.2018.00024

[6] ——, "On the expressiveness and semantics of information flow types," *J. Comput. Secur.*, vol. 28, no. 1, pp. 129–156, 2020. [Online]. Available: https://doi.org/10.3233/JCS-191382

[7] B. Moon, H. E. III, and D. Orchard, "Graded modal dependent type theory," in *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, vol. 12648. Springer, 2021, pp. 462–490. [Online]. Available: https://doi.org/10.1007/978-3-030-72019-3_17

[8] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *J. Comput. Secur.*, vol. 17, no. 5, pp. 517–548, 2009. [Online]. Available: https://doi.org/10.3233/JCS-2009-0352

[9] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, vol. 3233. Springer, 2003, pp. 174–191. [Online]. Available: https://doi.org/10.1007/978-3-540-37621-7_9

[10] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, 2005, pp. 158–170. [Online]. Available: https://doi.org/10.1145/1040305.1040319

[11] K. Slind and M. Norrish, "A brief overview of HOL4," in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, 2008, pp. 28–32. [Online]. Available: https://doi.org/10.1007/978-3-540-71067-7_6

[12] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, pp. 117–158, 2003. [Online]. Available: https://doi.org/10.1145/596980.596983

[13] V. Rajani, A. Coleman, and H. Kanabar, "A graded modal approach to relaxed semantic declassification (HOL mechanisation)," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15421629

[14] V. Rajani, M. Gaboardi, D. Garg, and J. Hoffmann, "A unifying type-theory for higher-order (amortized) cost analysis," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–28, 2021. [Online]. Available: https://doi.org/10.1145/3434308

[15] V. Rajani, G. Barthe, and D. Garg, "A modal type theory of expected cost in higher-order probabilistic programs," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, 2024. [Online]. Available: https://doi.org/10.1145/3689725

[16] B. Jacobs, *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge University Press, 2016, vol. 59. [Online]. Available: https://doi.org/10.1017/CBO9781316823187

[17] G. A. Kavvos, "Modalities, cohesion, and information flow," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 20:1–20:29, 2019. [Online]. Available: https://doi.org/10.1145/3290333

[18] J. Sterling and R. Harper, "Sheaf semantics of termination-insensitive noninterference," in *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, 2022, pp. 5:1–5:19. [Online]. Available: https://doi.org/10.4230/LIPIcs.FSCD.2022.5

[19] J. Power and H. Watanabe, "Combining a monad and a comonad," *Theoretical Computer Science*, vol. 280, no. 1, pp. 137–162, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030439750100024X

[20] M. Gaboardi, S.-y. Katsumata, D. Orchard, F. Breuvart, and T. Uustalu, "Combining effects and coeffects via grading," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, 2016, p. 476–489. [Online]. Available: https://doi.org/10.1145/2951913.2951939

[21] M. Abadi, "Access control in a core calculus of dependency," *SIGPLAN Not.*, vol. 41, no. 9, p. 263–273, 2006. [Online]. Available: https://doi.org/10.1145/1160074.1159839

[22] A. Ahmed, "Step-indexed syntactic logical relations for recursive and quantified types," in *Proceedings of the 15th European Conference on Programming Languages and Systems*. Springer-Verlag, 2006, p. 69–83. [Online]. Available: https://doi.org/10.1007/11693024_6

[23] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: mixing static and dynamic typing for information-flow control in haskell," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, 2015, pp. 289–301. [Online]. Available: https://doi.org/10.1145/2784731.2784758

[24] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, 2011, pp. 95–106. [Online]. Available: https://doi.org/10.1145/2034675.2034688

[25] O. Arden and A. C. Myers, "A calculus for flow-limited authorization," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 135–149. [Online]. Available: https://doi.org/10.1109/CSF.2016.17

[26] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2017, p. 1875–1891. [Online]. Available: https://doi.org/10.1145/3133956.3134054

[27] M. Ngo, D. A. Naumann, and T. Rezk, "Type-based declassification for free," in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, 2020, pp. 181–197. [Online]. Available: https://doi.org/10.1007/978-3-030-63406-3_11

[28] P. Wadler, "Theorems for free!" in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. Association for Computing Machinery, 1989, p. 347–359. [Online]. Available: https://doi.org/10.1145/99370.99404

[29] R. Cruz, T. Rezk, B. P. Serpette, and É. Tanter, "Type abstraction for relaxed noninterference," in *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, 2017, pp. 7:1–7:27. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2017.7

[30] J. Menz, A. K. Hirsch, P. Li, and D. Garg, "Compositional security definitions for higher-order where declassification," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 406–433, 2023. [Online]. Available: https://doi.org/10.1145/3586041

[31] N. Broberg and D. Sands, "Paralocks: role-based information flow control and beyond," in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, 2010, pp. 431–444. [Online]. Available: https://doi.org/10.1145/1706299.1706349

[32] E. Kozyri and F. B. Schneider, "RIF: reactive information flow labels," *J. Comput. Secur.*, vol. 28, no. 2, pp. 191–228, 2020. [Online]. Available: https://doi.org/10.3233/JCS-191316

Appendix

Typing judgement: $\boxed{\Gamma \Vdash e : \tau}$

$$\dfrac{\Gamma \Vdash e : \Diamond_{\ell}\tau \qquad \Gamma, x : \tau \Vdash e' : \Diamond_{\ell'}\tau'}{\Gamma \Vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : \Diamond_{\ell'}\tau'}\ \text{T-bind} \qquad\qquad \dfrac{\Gamma \Vdash e : \Box_{\phi,\tau'}\tau \qquad \Gamma \Vdash \phi : \tau \to \tau'}{\Gamma \Vdash \mathsf{dec}\ \phi\ e : \tau'}\ \text{T-dec}$$

$$\dfrac{\Gamma \Vdash e : \tau \qquad \Gamma \Vdash \phi : \tau \to \tau'}{\Gamma \Vdash \Box e : \Box_{\phi,\tau'}\tau}\ \text{T-inject}$$

$$\dfrac{\Gamma \Vdash e : \Box_{\phi_2 \cdot \phi_1,\tau'}\tau \qquad \Gamma \Vdash \phi_1 : \tau \to \tau'' \qquad \Gamma \Vdash \phi_2 : \tau'' \to \tau' \qquad \phi' \triangleq \lambda x.\mathsf{let}\ y = \mathsf{dec}\ \phi_1\ x\ \mathsf{in}\ \phi_2\ y}{\Gamma \Vdash \mathsf{split}\ e : \Box_{\phi'}\Box_{\phi_1}\tau}\ \text{T-split}$$

Fig. 8. Type system for policies (selected rules)

Typing derivation of examples

Example 1: Semantically secure program on all inputs

Let
$\phi : \Diamond_{\ell}\mathbb{N} \to \Diamond_{\bot}\mathbb{N}$
$\phi \triangleq \lambda x.\mathsf{bind}\ y = x\ \mathsf{in}\ \mathsf{ret}\ ((4 * y)\%2 + 10)$
$\Delta \triangleq [(\phi : \Diamond_{\ell}\mathbb{N} \to \Diamond_{\bot}\mathbb{N})]$

$$\dfrac{\dfrac{\dfrac{}{\Delta; \cdot; x : \Diamond_{\ell}\mathbb{N} \vdash x : \Diamond_{\ell}\mathbb{N}} \qquad \Delta(\phi) = \Diamond_{\ell}\mathbb{N} \to \Diamond_{\bot}\mathbb{N}}{\Delta; \cdot; x : \Diamond_{\ell}\mathbb{N} \vdash \mathsf{dec}\ \phi\ x : \Diamond_{\bot}\mathbb{N}}}{\Delta; \cdot; \cdot \vdash \lambda x.\ \mathsf{dec}\ \phi\ x : \Diamond_{\ell}\mathbb{N} \to \Diamond_{\bot}\mathbb{N}}$$

Example 2: Semantically secure program on some inputs

Let
$\phi : \Diamond_{\ell}(\mathbb{N} \times \mathbb{N}) \to \Diamond_{\bot}\mathbb{N}$
$\phi \triangleq \lambda x.\mathsf{bind}\ y = x\ \mathsf{in}\ \mathsf{ret}\ (\mathsf{fst}\ y + \mathsf{snd}\ y)$
$\Phi \triangleq [\phi : \Diamond_{\ell}(\mathbb{N} \times \mathbb{N}) \to \Diamond_{\bot}\mathbb{N}]$

$$\dfrac{\dfrac{}{\cdot; \Phi; x : \Box_{\phi}\Diamond_{\ell}(\mathbb{N} \times \mathbb{N}) \vdash x : \Box_{\phi}\Diamond_{\ell}(\mathbb{N} \times \mathbb{N})} \qquad \Phi(\phi) = \Diamond_{\ell}(\mathbb{N} \times \mathbb{N}) \to \Diamond_{\bot}\mathbb{N}}{\cdot; \Phi; x : \Box_{\phi}\Diamond_{\ell}(\mathbb{N} \times \mathbb{N}) \vdash \mathsf{dec}\ \phi\ x : \Diamond_{\bot}\mathbb{N}}$$

Example 3: A state machine as a policy

Let

$$\begin{array}{rcl}
\Delta & \triangleq & [\phi_1 : \Diamond_{\top}\mathbb{N} \to \Diamond_{\bot}(\Diamond_{\bot}\mathbb{N} \times \Box_{\phi_2}\Diamond_{\top}\mathbb{N}), \phi_2 : \Diamond_{\top}\mathbb{N} \to \Diamond_{\bot}\mathbb{N}] \\
f & \triangleq & \lambda x.\ \mathsf{bind}\ y = \mathsf{dec}\ \phi_1\ x\ \mathsf{in}\ e_1 \\
e_1 & \triangleq & (\lambda z.\mathsf{ret}\ (\mathsf{fst}\ y, z))\ (\mathsf{dec}\ \phi_2\ (\mathsf{snd}\ y))
\end{array}$$

D4:

$$\dfrac{\dfrac{\dfrac{}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_{\top}\mathbb{N}), y : (\Diamond_{\bot}\mathbb{N} \times \Box_{\phi_2}\Diamond_{\top}\mathbb{N}) \vdash y : (\Diamond_{\bot}\mathbb{N} \times \Box_{\phi_2}\Diamond_{\top}\mathbb{N})}}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_{\top}\mathbb{N}), y : (\Diamond_{\bot}\mathbb{N} \times \Box_{\phi_2}\Diamond_{\top}\mathbb{N}) \vdash \mathsf{snd}\ y : \Box_{\phi_2}\Diamond_{\top}\mathbb{N}} \qquad \Delta(\phi_2) = \Diamond_{\top}\mathbb{N} \to \Diamond_{\bot}\mathbb{N}}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_{\top}\mathbb{N}), y : (\Diamond_{\bot}\mathbb{N} \times \Box_{\phi_2}\Diamond_{\top}\mathbb{N}) \vdash (\mathsf{dec}\ \phi_2\ (\mathsf{snd}\ y)) : \Diamond_{\bot}\mathbb{N}}$$

D3:

$$\dfrac{\dfrac{}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_{\top}\mathbb{N}), y : (\Diamond_{\bot}\mathbb{N} \times \Box_{\phi_2}\Diamond_{\top}\mathbb{N}), z : \Diamond_{\bot}\mathbb{N} \vdash \mathsf{ret}\ (\mathsf{fst}\ y, z) : \Diamond_{\bot}(\Diamond_{\bot}\mathbb{N} \times \Diamond_{\bot}\mathbb{N})}}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_{\top}\mathbb{N}), y : (\Diamond_{\bot}\mathbb{N} \times \Box_{\phi_2}\Diamond_{\top}\mathbb{N}) \vdash (\lambda z.\mathsf{ret}\ (\mathsf{fst}\ y, z)) : \Diamond_{\bot}\mathbb{N} \to \Diamond_{\bot}(\Diamond_{\bot}\mathbb{N} \times \Diamond_{\bot}\mathbb{N})}$$

D2:

$$\dfrac{\dfrac{D3 \qquad D4}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_\top \mathbb{N}), y : (\Diamond_\bot \mathbb{N} \times \Box_{\phi_2}\Diamond_\top \mathbb{N}) \vdash (\lambda z.\mathsf{ret}\ (\mathsf{fst}\ y, z))\ (\mathsf{dec}\ \phi_2\ (\mathsf{snd}\ y)) : \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Diamond_\bot \mathbb{N})}}{\Delta; \cdot; x : \Box_{\phi_1}, y : (\Diamond_\bot \mathbb{N} \times \Box_{\phi_2}\Diamond_\top \mathbb{N}) \vdash e_1 : \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Diamond_\bot \mathbb{N})}$$

D1:

$$\dfrac{\dfrac{}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_\top \mathbb{N}) \vdash x : \Box_{\phi_1}(\Diamond_\top \mathbb{N})} \qquad \Delta(\phi_1) = \Diamond_\top \mathbb{N} \to \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Box_{\phi_2}\Diamond_\top \mathbb{N})}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_\top \mathbb{N}) \vdash \mathsf{dec}\ \phi_1\ x : \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Box_{\phi_2}\Diamond_\top \mathbb{N})}$$

Main derivation:

$$\dfrac{\dfrac{D1 \qquad D2}{\Delta; \cdot; x : \Box_{\phi_1}(\Diamond_\top \mathbb{N}) \vdash \mathsf{bind}\ y = \mathsf{dec}\ \phi_1\ x\ \mathsf{in}\ e_1 : \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Diamond_\bot \mathbb{N})}}{\Delta; \cdot; \cdot \vdash \lambda x.\ \mathsf{bind}\ y = \mathsf{dec}\ \phi_1\ x\ \mathsf{in}\ e_1 : \Box_{\phi_1}(\Diamond_\top \mathbb{N}) \to \Diamond_\bot(\Diamond_\bot \mathbb{N} \times \Diamond_\bot \mathbb{N})}$$

Example 4: Partial and conditional declassification

$$\begin{array}{lcl}
\Delta & \triangleq & [\phi : \Diamond_\ell \mathbb{N} \to \Diamond_\bot \mathbb{N}, \phi_1 : \Diamond_\ell \mathbb{N} \to \Diamond_{\ell'} \mathbb{N}, \phi_2 : \Diamond_{\ell'} \mathbb{N} \to \Diamond_\bot \mathbb{N}] \\
f & \triangleq & \lambda\ x\ b.\ \mathsf{case}\ b\ \mathsf{of}\ \_.e_1\ \_.e_2 \\
e_1 & \triangleq & \mathsf{dec}\ \phi\ x \\
e_2 & \triangleq & (\lambda y.\mathsf{dec}\ \phi_1\ y)\ (\mathsf{coret}\ (\mathsf{split}\ x))
\end{array}$$

D5:

$$\dfrac{}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash x : \Box_\phi(\Diamond_\ell \mathbb{N})}$$

D4:

$$\dfrac{\dfrac{D5 \qquad \phi = \phi_2 \cdot \phi_1 \qquad \Delta(\phi_1) = \Diamond_\ell \mathbb{N} \to \Diamond_{\ell'} \mathbb{N} \qquad \Delta(\phi_2) = \Diamond_{\ell'} \mathbb{N} \to \Diamond_\bot \mathbb{N} \qquad \phi' \triangleq \lambda x.\mathsf{let}\ y = \mathsf{dec}\ \phi_1\ x\ \ \mathsf{in}\ \phi_2\ y}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash \mathsf{split}\ x : \Box_{\phi'}(\Box_{\phi_1}\Diamond_\ell \mathbb{N})}}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash \mathsf{coret}\ (\mathsf{split}\ x) : \Box_{\phi_1}\Diamond_\ell \mathbb{N}}$$

D3:

$$\dfrac{\dfrac{\dfrac{}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}), y : \Box_{\phi_1}\Diamond_\ell \mathbb{N} \vdash y : \Box_{\phi_1}\Diamond_\ell \mathbb{N}} \qquad \Delta(\phi_1) = \Diamond_\ell \mathbb{N} \to \Diamond_{\ell'} \mathbb{N}}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}), y : \Box_{\phi_1}\Diamond_\ell \mathbb{N} \vdash \mathsf{dec}\ \phi_1\ y : \Diamond_{\ell'} \mathbb{N}}}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash (\lambda y.\mathsf{dec}\ \phi_1\ y) : \Box_{\phi_1}\Diamond_\ell \mathbb{N} \to \Diamond_{\ell'} \mathbb{N}}$$

D2:

$$\dfrac{\dfrac{D3 \qquad D4}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash (\lambda y.\mathsf{dec}\ \phi_1\ y)\ (\mathsf{coret}\ (\mathsf{split}\ x)) : \Diamond_{\ell'} \mathbb{N}}}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash e_2 : \Diamond_{\ell'} \mathbb{N}}$$

D1:

$$\dfrac{\dfrac{\dfrac{}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash x : \Box_\phi(\Diamond_\ell \mathbb{N})} \qquad \Delta(\phi) = \Diamond_\ell \mathbb{N} \to \Diamond_\bot \mathbb{N}}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash \mathsf{dec}\ \phi\ x : \Diamond_\bot \mathbb{N}}}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash e_1 : \Diamond_{\ell'} \mathbb{N}}$$

Main derivation:

$$\dfrac{\dfrac{\dfrac{}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : \Diamond_\bot(\mathbb{N} + \mathbb{N}) \vdash b : (\mathbb{N} + \mathbb{N})} \qquad D1 \qquad D2}{\Delta; \Phi; x : \Box_\phi(\Diamond_\ell \mathbb{N}), b : (\mathbb{N} + \mathbb{N}) \vdash \mathsf{case}\ b\ \mathsf{of}\ \_.e_1\ \_.e_2 : \Diamond_{\ell'} \mathbb{N}}}{\Delta; \Phi; \cdot \vdash \lambda\ x\ b.\ \mathsf{case}\ b\ \mathsf{of}\ \_.e_1\ \_.e_2 : \Box_\phi(\Diamond_\ell \mathbb{N}) \to (\mathbb{N} + \mathbb{N}) \to \Diamond_{\ell'} \mathbb{N}}$$