A Categorical Model of Dynamic Information-Flow Policies

Antonio Zegarelli, Niki Vazou, and Marco Guarnieri

IMDEA Software Institute
antonio.zegarelli@imdea.org, niki.vazou@imdea.org,
marco.guarnieri@imdea.org

Abstract. Information-flow policies specify which data may flow to which observers. Systems that permit such specifications to change at runtime are said to enforce *dynamic policies*. Proving security for systems with dynamic policies is challenging, because the dynamic updates alter what an observer can distinguish. In contrast, categorical semantics provides elegant noninterference proofs for systems with static policies. In this work, we provide a categorical proof of noninterference for a system with dynamic policies. We design Dynamic Dependency Core Calculus (DDCC), an extension of DCC with explicit, certified policy changes, and $\mathbf{DSet}_L$, an extension of the category of classified sets ($\mathbf{CSet}_L$) with transformations bounded by independently authorized releases. Finally, we interpret DDCC in $\mathbf{DSet}_L$ and prove noninterference for DDCC programs by reduction to $\mathbf{DSet}_L$'s release-relative constancy.

Keywords: Information-flow control · Dynamic policies · Noninterference · Categorical semantics · Declassification.

1 Introduction

Information-flow control (IFC) enforces a security policy describing which data are permitted to influence which observations [19]. *Dynamic policies* allow these permissions to change during program execution [13,7].

As an example of a dynamic policy, consider a scenario in which a patient at a hospital is transferred to an external specialist. The hospital intends to share the patient's confidential data with this specialist, but data disclosure requires a new flow permission and explicit authorization. The hospital has a copy h of patient data, protected at the hospital's security label H. We borrow the notation of Dependency Core Calculus (DCC) [1] and for the patient data of type D, we type $h : T_H\ \mathsf{D}$, where the type $T_\ell\ A$ denotes data of type A protected at label ℓ. External data is protected at label E while the doctor observes at label D.

Policies specify which information flows are permitted between security labels. In our hospital example, the policy before the transfer, call it p, does not permit hospital data to flow to the specialist. To enable this flow, a new policy p_s adds $H \sqsubseteq_{p_s} D$, meaning that hospital-protected data can be shared with the doctor

under p_s. We name this policy change transfer, writing $\mathsf{transfer} \triangleq p \rightsquigarrow p_s$. The following program applies the change and passes the hospital's copy to the doctor:

$$M \doteq \mathsf{policy\ transfer\ in\ (let}\ y \leftarrow h\ \mathsf{in}\ [y]_D).$$

Here policy transfer in applies the policy change before running its body under the new policy. The binding let $y \leftarrow h$, using the DCC syntax, opens the protected copy as y and $[y]_D$ protects the result at the doctor label. Thus, by dynamically changing the policy, the hospital can enable the transfer of its protected data to the specialist doctor. In §4, *we present Dynamic DCC (DDCC), an extension of DCC that permits explicit changes to the security policy* and we revisit this hospital example with its complete workflow and policy changes.

A policy change, though, may expose too much information: $H \sqsubseteq_{p_s} D$ makes *all* stored hospital-protected data visible to the specialist. We therefore require an explicitly declared *authorization release* map that—per Sabelfeld and Myers' *delimited release* [20]—specifies which information may be disclosed. The policy author supplies this map as a trusted specification, independently of the security proof. DDCC then checks a security certificate proving that the policy change exposes no more than the authorization release specification permits.

Back to our hospital example, and assuming a user-specified release function $\mathsf{release}_{\mathsf{transfer}}$, the transfer policy change must reveal no more than $\mathsf{release}_{\mathsf{transfer}}$ permits. If two runs look the same to the specialist doctor before the transfer and have the same release, they must still look the same afterwards. DDCC checks these guarantees statically: typing requires a proof of the predicate Secure at each policy change (definition 6). This predicate requires that, for every observer, initially indistinguishable runtime environments with equal releases remain indistinguishable after the change. For example, a constant release that authorizes no disclosure would not certify the transfer, thus type checking would fail for such a policy change. The strength of this guarantee depends on the chosen authorization: an injective release distinguishes every runtime environment, authorizing all distinctions and making the certificate condition automatic.

Having designed DDCC, we set out to prove it safe, meaning that it satisfies noninterference (NI) relative to the authorized releases. NI guarantees have been, in the past, proved for systems with dynamic policies. Yet, such proofs are notoriously challenging. For example, reasoning about what an observer learns requires relating changing runtime values to reference memories and input histories, as in Askarov and Sabelfeld [5], and controlling implicit flows caused by policy-change decisions, as in SLIO [8]. Observable behavior introduces further obligations: Askarov and Chong [3] account for information revealed by whether events occur, in addition to their values, while DAISY [12] must reason jointly about application and database behavior, including triggers and policy updates.

On the other hand, Kavvos [14] provides a categorical proof of NI: interpreting a calculus in classified sets reduces NI to constancy of maps from protected inputs to visible outputs. Soundness and adequacy connect this semantic argument to program behavior, separating the general categorical proof from language-specific obligations. Kavvos's development, even though it was only used for systems with static policies, is simple and elegant.

In this work, *we prove NI of DDCC using the categorical NI argument of Kavvos [14].* Our NI theorem guarantees equal observable final results for initially indistinguishable inputs with equal accumulated releases, but, unlike [8] and [3], does not guarantee knowledge-based security at intermediate steps of execution. Instead, we construct a category that uses a policy to describe indistinguishability and a release to specify which distinctions may become observable. Refining the input relation by equality of that release lets us express a secure change as a classified-set morphism. Such maps compose by pairing releases and protected-to-visible constancy becomes constancy within each release class. We call the resulting category $\mathbf{DSet}_L$ and use it to interpret DDCC.

Concretely, our list of contributions is the following:

1. **We construct $\mathbf{DSet}_L$ (§3). $\mathbf{DSet}_L$** is the category designed by extending $\mathbf{CSet}_L$ [14], the category of classified sets, with transformations bounded by authorized releases. Refining indistinguishability by equality of releases expresses security as preservation of relations. We prove that $\mathbf{DSet}_L$ is in fact a category (theorem 5) and that its morphisms from protected inputs to visible outputs are constant within each release class (theorem 7). This gives NI relative to the authorized releases. These results provide a language-independent foundation for proving NI under policy changes.
2. **We design DDCC (§§4.1–4.3).** We design DDCC as an extension of DCC that supports explicit policy changes together with a sequential computation primitive that permits the combination of policy changes with the other DCC expressions. To type policy changes, a security certificate needs to confirm that the change is explicitly authorized by comparison to a declared release.
3. **We interpret DDCC in $\mathbf{DSet}_L$ and prove NI (§4.4).** We interpret the elements of DDCC in $\mathbf{DSet}_L$, showing how each construct corresponds to a morphism in the category (theorem 8). Then release-relative constancy of $\mathbf{DSet}_L$'s morphisms ensures semantic NI. Finally, we prove adequacy of DDCC (theorem 9), meaning that the semantic interpretation in $\mathbf{DSet}_L$ accurately reflects the operational behavior of DDCC programs. Thus, we obtain NI for DDCC programs (theorem 10).

§ 2 starts with a description of the categorical argument for NI in the static fragment of DCC and concludes with an overview of the required extensions to support the dynamic features of DDCC. § 5 presents the extensions of DDCC with conditionals, functions, and recursion, whose formal development is deferred to the appendix. Finally, we present related work (§6) and conclude (§7).

The supplementary material contains the proofs of all the claims of this paper, both in an appendix and in a Lean mechanization generated by OpenAI's Codex.

2 The Categorical Proof of Noninterference

This section summarizes the categorical proof of noninterference (NI) for the Dependency Core Calculus (DCC) [1] as conducted by Kavvos [14]. Its purpose is

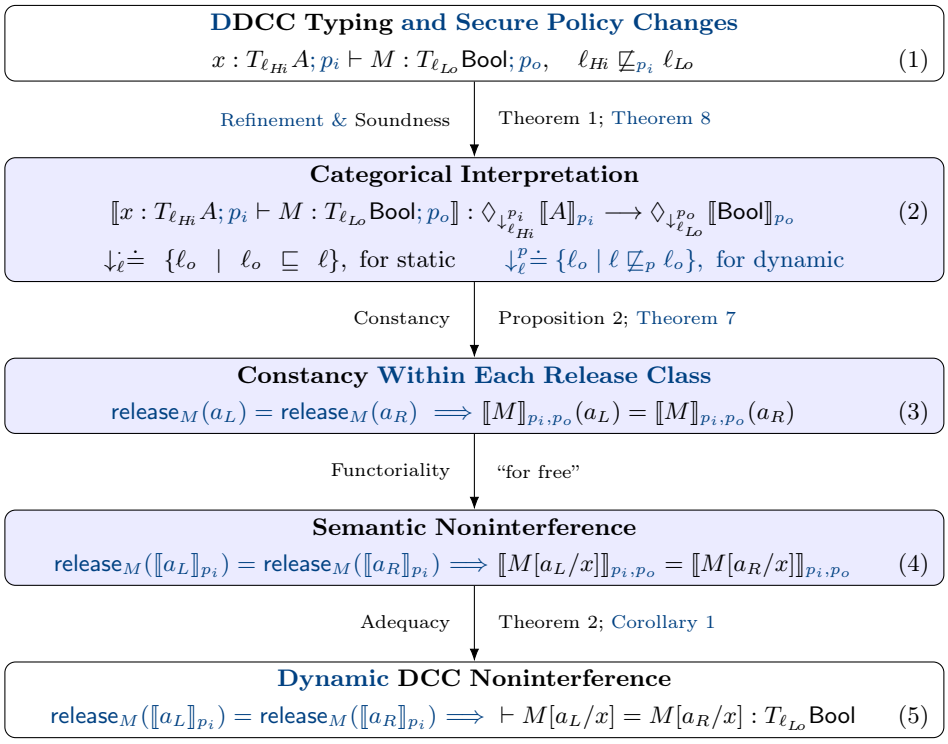


Fig. 1. Categorical proof of static and dynamic DCC noninterference. Black gives the static argument; blue extends it with the dynamic requirements. The interpretation of a judgment is a morphism; $[\![M]\!]_{p_i,p_o}$ denotes its underlying function. Shaded boxes form the categorical argument, while the rest present the syntactic derivations.

twofold: to present how the categorical concepts can simplify proofs, that would be cumbersome if done entirely syntactically and, at the same time, present the basis upon which we will develop the dynamic proof.

Figure 1 summarizes the static and dynamic proofs; the dynamic extensions are indicated in blue and will be detailed in the rest of the paper. The shaded boxes refer to the categorical interpretation of the proof, while the rest presents the syntax of DCC. Concretely, the NI property is the implication from the top to the bottom box of the diagram and states that if a term takes an input protected at level ℓ_{Hi} and returns a boolean protected at level ℓ_{Lo}, where $\ell_{Hi} \not\sqsubseteq \ell_{Lo}$ *(box (1),* ***DCC Typing****)*, then the result is independent of the protected input *(box (5),* ***DCC Noninterference****)*.

The proof goes through the shaded boxes of the categorical interpretation. First (§ 2.2), soundness sends the typing judgment to a morphism *(box (2),* ***Categorical Interpretation****)*. Then, the classified-set model (§ 2.3) forces this morphism to be constant *(box (3),* ***Constancy****)*. Functoriality then gives equal denotations (§ 2.4) after substitution of any two closed protected inputs *(box (4),*

Semantic Noninterference), and adequacy reflects this semantic equality back to equality in DCC. Next, we start by detailing the static proof and finally, in § 2.5, we explain how this categorical architecture extends to dynamic policies.

2.1 DCC and Noninterference

DCC [1] is a typed calculus in which dependencies are made explicit in types. For a fixed poset $(L, \sqsubseteq)$ of security levels, DCC extends the simply typed λ-calculus with a family of type constructors T_ℓ, one for each $\ell \in L$. A term of type $T_\ell\ A$ contains an A-value protected at level ℓ.

For example, let Hi be a high security level and suppose that $s : T_{Hi}\ \mathsf{Bool}$ contains a secret boolean. DCC permits the secret to be unpacked when the result remains protected:

$$\mathsf{let}\ x \leftarrow s\ \mathsf{in}\ [x]_{Hi} : T_{Hi}\ \mathsf{Bool} \qquad \texttt{-- OK} \qquad \text{with } s : T_{Hi}\ \mathsf{Bool}$$

Here $[x]_{Hi}$ protects the result before it leaves the scope of the unpacked secret. In contrast, the program

$$\mathsf{let}\ x \leftarrow s\ \mathsf{in}\ x \quad : \mathsf{Bool} \qquad \texttt{-- ERROR} \quad \text{with } s : T_{Hi}\ \mathsf{Bool}$$

is not typable: it would expose the secret as an unprotected boolean. In general, DCC allows a protected value to be opened only when the final result type still protects the information at the required level. This way, data protection is enforced statically, during type checking.

The typing judgment $\Gamma \vdash M : A$ states that the expression M has type A in context Γ. Here we use the first-order fragment: booleans, unit, products, and protected types $T_\ell\ B$. Concretely, consider the typing judgment (1) of fig. 1:

$$x : T_{\ell_{Hi}}\ A \vdash M : T_{\ell_{Lo}}\ \mathsf{Bool}, \qquad \ell_{Hi} \not\sqsubseteq \ell_{Lo}.$$

It types M under one input x, whose value of type A is protected at ℓ_{Hi} and the result is a boolean protected at ℓ_{Lo}. The side condition says that an observer at ℓ_{Lo} is not permitted to observe information protected at ℓ_{Hi}.

NI states that the observable result cannot depend on the protected input. Equivalently, for all closed $a_L, a_R : T_{\ell_{Hi}}\ A$, NI requires judgment (5) in fig. 1:

$$\vdash M[a_L/x] = M[a_R/x] : T_{\ell_{Lo}}\ \mathsf{Bool}$$

where program equality means that the two closed terms have the same canonical normal form—in this case, the same protected boolean.

Note that we expressed NI using a boolean result type, because it provides the smallest nontrivial observable result: an observer can distinguish true from false, so equal results express the absence of a leaked bit. The exact same argument generalizes to any ground result type whose distinct constants have distinct denotations.

Proofs of NI go back to the original work of Abadi et al. [1]. Their proof is also semantic and categorical: they interpret DCC in a dependency category whose objects are cpos equipped with observer-indexed relations and whose morphisms

preserve those relations. They then use this model, together with adequacy, to derive NI for analyses translated into DCC, such as the functional SLam calculus [1, Theorem 4.2]. Here, we follow Kavvos's more general categorical argument, which organizes the proof around classified sets, modalities, and protected-to-visible constancy [14, §§ 5–6], with DCC only one of many applications.

2.2 From Syntax to Semantics and Back

A categorical proof connects programs to their denotations through soundness and adequacy. *Soundness* ensures that well-typed terms denote morphisms and that program equality implies denotational equality. *Adequacy* supplies the converse at observable ground types: equal denotations imply equal program results. We first describe these requirements abstractly and then, in § 2.3, construct the model that satisfies them. For now, assume a categorical model whose objects have underlying sets of values and whose morphisms act as functions on those sets. Throughout the static argument, the flow preorder policy p is fixed, so we omit the policy index on interpretations, while § 4.4 makes these indices explicit.

Soundness. Following the standard denotational interpretation of typed calculi [17], a typing judgment $\Gamma \vdash M : A$, is interpreted by a product of semantic objects for the context Γ, an object $[\![A]\!]$ for the type A, and a morphism

$$[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!].$$

The judgment interpretation combines the underlying function ($[\![M]\!] : |[\![\Gamma]\!]| \to |[\![A]\!]|$) with a proof that the observer relations are preserved (revisited in § 4.4).

In the case of the monadic DCC calculus, the standard soundness proof of Moggi [18] can be adapted to ensure that well-typed terms denote morphisms in the semantic category.

Theorem 1 (DCC Soundness$^{\mathsf{L}\exists\forall\mathsf{N}}$). *If $\Gamma \vdash M : T_\ell\ A$, then $[\![\Gamma \vdash M : T_\ell\ A]\!] : [\![\Gamma]\!] \to \Diamond_{\downarrow\ell}[\![A]\!]$ is a morphism.*

Soundness requires every typing derivation to determine such a morphism and every derivable program equality to give equal denotations. With this DCC soundness claim, we move from the typing judgment (1) of fig. 1 to the categorical morphism (2).

Adequacy connects the model back to program terms: at observable ground types, equality of denotations implies equality of the corresponding programs. Let G be a ground type with canonical constants c_i, and write $\eta_G : [\![G]\!] \to [\![T_\ell\ G]\!]$ for the semantic map interpreting protection $[\,\cdot\,]_\ell$. Assume that closed terms normalize to canonical forms, distinct constants have distinct denotations, and η_G is monic, so it can be cancelled from equal composites.

Theorem 2 (DCC Adequacy$^{\mathsf{L}\exists\forall\mathsf{N}}$). *For a closed $N : T_\ell\ G$, if $[\![N]\!] = [\![[c_i]_\ell]\!]$, then $\vdash N = [c_i]_\ell : T_\ell\ G$.*

This adequacy lemma supplies the return step of the categorical proof back to the syntax of the language. Applied to protected booleans, this lemma turns semantic equality in box (4) of fig. 1 into program equality in box (5).

Soundness and adequacy provide the connections between syntax and semantics. The next subsection constructs the semantic category interpreting DCC.

2.3 The Classified-Set Construction

The classified-set model supplies the semantic category and protection modalities required by the interpretation of DCC.

For a set L of observer levels, a classified set X consists of a carrier $|X|$ and a reflexive relation R_ℓ^X on $|X|$ for each $\ell \in L$. The relation represents observational indistinguishability: $x\, R_\ell^X\, y$ means that an observer at ℓ cannot distinguish x from y. A morphism $f : X \to Y$ is a function satisfying

$$x\, R_\ell^X\, y \quad \Longrightarrow \quad f(x)\, R_\ell^Y\, f(y) \qquad (\ell \in L). \tag{1}$$

Identities and composites preserve these relations, giving the category $\mathbf{CSet}_L$ of classified sets and relation-preserving functions.

For $\pi \subseteq L$, there are two carrier-preserving modalities. The *redaction* modality $\Diamond_\pi$ replaces R_ℓ by the total relation at every $\ell \in \pi$; hence all values become indistinguishable to those observers. The *disclosure* modality $\Box_\pi$ replaces R_ℓ by equality at every $\ell \in \pi$; hence every distinction becomes visible. An object X is *protected at* π when its relations at π are total and *visible at* π when they are equality. Thus $\Diamond_\pi X$ is protected at π, while $\Box_\pi X$ is visible there.

These modalities arise from pre-cohesion [15], whose adjoint functors forget observational relations and restore them as equality or total relations at selected levels [14, §§ 3–6]. In particular, $\Diamond_\pi$ is a strong, strictly idempotent monad and can interpret a DCC protection constructor. For a security level ℓ, let $\downarrow \ell = \{\ell' \in L \mid \ell' \sqsubseteq \ell\}$ be its principal lower set. The (static) DCC protection monad T_ℓ is interpreted by $\Diamond_{\downarrow\ell}$, as presented in (2) of fig. 1:

$$[\![T_\ell\ A]\!] \doteq \Diamond_{\downarrow\ell}[\![A]\!].$$

The boolean type has carrier $\mathbb{B} \doteq \{\mathsf{true}, \mathsf{false}\}$ and is interpreted by the discrete classified set $[\![\mathsf{Bool}]\!] \doteq \Delta\mathbb{B}$, whose relation is equality at every level. Unit is interpreted by a singleton and products use the cartesian product of $\mathbf{CSet}_L$. The monad's unit interprets protection $[\,\cdot\,]_\ell$ while protected elimination uses the monadic structure together with the protection condition on the result type. These interpretations give the morphism of theorem 1.

They also preserve the distinctions between canonical values required by adequacy. Canonical first-order values are booleans, unit, pairs of canonical values, and protected canonical values.

Proposition 1 (Canonical-value Injectivity$^{\mathsf{L}\exists\forall\mathsf{N}}$). *For canonical first-order values $w_L, w_R : A$, $[\![w_L]\!] = [\![w_R]\!]$ implies $w_L = w_R$.*

2.4 Noninterference Reduces to Categorical Constancy

Having interpreted DCC in $\mathbf{CSet}_L$, we can prove data protection within this category. Concretely, a classified set is *visible* at an observer level when that observer can tell any two different values apart. We prove that *protected* arguments, whose values the observer cannot distinguish, cannot affect a result visible to that observer: the corresponding classified-set morphism must be constant.

Proposition 2 (Constancy$^{\mathsf{L}\exists\forall\mathsf{N}}$; Proposition 23 [14]). *Suppose $\pi_{Hi} - \pi_{Lo} \neq \varnothing$, A has nonempty carrier, and B is visible at every level in $\pi_{Hi} - \pi_{Lo}$. Then every morphism $f : \Diamond_{\pi_{Hi}} A \longrightarrow \Diamond_{\pi_{Lo}} B$ has a constant underlying function.*

The DCC assumption $\ell_{Hi} \not\sqsubseteq \ell_{Lo}$ is equivalent to $\downarrow \ell_{Hi} \not\subseteq \downarrow \ell_{Lo}$, and hence $\downarrow \ell_{Hi} - \downarrow \ell_{Lo} \neq \varnothing$. Since $\Delta\mathbb{B}$ is visible at every level, proposition 2 applies to the morphism in (2) of fig. 1, yielding constancy in (3) of fig. 1:

$$\forall a_L, a_R.\ [\![M]\!](a_L) = [\![M]\!](a_R).$$

Functoriality converts constancy in (3) of fig. 1 into semantic noninterference in (4). Substitution is interpreted by composition of the underlying functions, so any two closed inputs $a_L, a_R : T_{Hi}\ A$ give

$$[\![M[a_L/x]]\!] = [\![M]\!] \circ [\![a_L]\!] = [\![M]\!] \circ [\![a_R]\!] = [\![M[a_R/x]]\!].$$

In $\mathbf{CSet}_L$, the components of the map interpreting protection $A \to \Diamond_\pi A$ are identity functions on carriers and therefore monomorphisms. The terminating DCC with booleans has the required canonicity, and proposition 1 distinguishes the canonical results. Theorem 2 therefore reflects semantic equality in (4) back to program equality in (5) of fig. 1. Together, these steps establish the NI theorem.

Theorem 3 (Noninterference for DCC$^{\mathsf{L}\exists\forall\mathsf{N}}$). *Let A be inhabited. If $x : T_{\ell_{Hi}}\ A \vdash M : T_{\ell_{Lo}}\ \mathsf{Bool}$ and $\ell_{Hi} \not\sqsubseteq \ell_{Lo}$, then for all closed $a_L, a_R : T_{\ell_{Hi}}\ A$, $\vdash M[a_L/x] = M[a_R/x] : T_{\ell_{Lo}}\ \mathsf{Bool}$.*

The language-specific soundness and adequacy results thus connect the classified-set constancy argument to static DCC noninterference for ℓ_{Hi} and ℓ_{Lo}.

2.5 Adapting the Architecture to Dynamic Policies

Dynamic policies change the relations used by the proof of NI (theorem 3) while keeping values that have been created. Each policy p_i supplies a flow preorder $\sqsubseteq_{p_i}$ on the same labels, so $[\![A]\!]_{p_i}$ has the same values of A but the relations are determined by that policy. A release declaration specifies which distinctions may become observable during a change. § 3 extends $\mathbf{CSet}_L$ with arrows that carry release bounds and § 4 interprets DDCC under this extended category.

Figure 1 with the blue extensions summarizes the proof architecture of NI under the presence of dynamic policies. Typing now records input and output policies, and each change requires a certificate for every variable in scope. Starting

from the typing judgment in *box (1)*, soundness gives the morphism in *box (2)*, whose source (*i.e.,* $\Diamond_{\downarrow^{p_i}_{\ell_{Hi}}} [\![A]\!]_{p_i}$) keeps its values (*i.e.,* the carrier $|[\![T_{\ell_{Hi}}\ A]\!]_{p_i}|$) and uses the observer relations determined by the input policy. The arrow's release bound requires preservation of these relations only when the authorized releases agree. The initial no-flow condition (*i.e.,* $\ell_{Hi} \not\sqsubseteq_{p_i} \ell_{Lo}$) makes the input protected at ℓ_{Lo}, while reflexivity under p_o makes the boolean output visible there. Thus, the morphism is constant within each release class, as stated in *box (3)*. Functoriality turns this constancy into equal denotations of programs with substituted inputs in *box (4)* and adequacy reflects that equality back to program equality in *box (5)*.

The dynamic redaction set $\downarrow^p_\ell = \{o \in L \mid \ell \not\sqsubseteq_p o\}$ contains exactly the observers to whom ℓ cannot flow. It differs from the static principal lower set $\downarrow \ell$ used in § 2.3. The static constancy proof uses ℓ_{Hi} as its separating observer, while the dynamic proof uses ℓ_{Lo}. Both establish the same static NI conclusion when the policy is fixed and releases are constant, but using different interpretations of protection. The next section constructs the category supporting this argument for arbitrary classified sets: arrows carry release bounds and maps from protected inputs to visible outputs are constant within each release class.

3 Dynamic Classified Sets: $\mathbf{DSet}_L$

The static categorical argument of § 2 relies on the protected-to-visible constancy of relation-preserving morphisms. To extend the same argument to the dynamic setting, we preserve $\mathbf{CSet}_L$'s classified sets as objects, but *we extend the arrows to carry the authorized release bounds.* We call the resulting category $\mathbf{DSet}_L$.

We first express release-bounded security as morphisms from refined sources and characterize this refinement as a pullback (§ 3.1). Then, we construct $\mathbf{DSet}_L$ and prove its category laws (§ 3.2), we extend redaction to an idempotent monad (§ 3.3), and we prove release-relative semantic noninterference (§ 3.4): protected inputs with equal releases give equal visible outputs.

3.1 Release Refinement

In $\mathbf{CSet}_L$ (presented in § 2.3), a classified-set morphism preserves indistinguishability: related inputs must produce related outputs. The authorized transfer example of § 1, however, asks for a weaker condition. Initially, the doctor cannot distinguish hospital copies with different values, but releasing the hospital data permits the doctor to observe the authorized differences. No further distinctions between initially indistinguishable data may become observable.

Thus, for classified sets X, Y, we weaken the preservation requirement of Eq. (1): a transformation $f : |X| \to |Y|$ need preserve indistinguishability only when the authorized releases agree:

$$x\, R^X_\ell\, y \ \wedge\ r_\ell(x) = r_\ell(y) \quad \Longrightarrow \quad f(x)\, R^Y_\ell\, f(y). \tag{2}$$

Here $r_\ell : |X| \to V_\ell$ specifies the release authorized for observer ℓ.

Equation (2) has the same equal-release pattern as Sabelfeld and Myers' delimited release [20, Definition 2]: initially indistinguishable inputs with equal authorized releases must yield indistinguishable outputs. Their definition compares terminating executions under fixed classifications, while our source and target relations may describe different policies.

To express eq. (2) as a classified-set morphism, we refine the source relation by the equality of the release. Writing $\boldsymbol{r}$ for this family, we define $X[\boldsymbol{r}]$ as follows.

Definition 1 (Release Refinement). *The classified set $X[\boldsymbol{r}]$ has carrier $|X|$ and relations*

$$x\, R_\ell^{X[\boldsymbol{r}]}\, y \quad \dot{\iff} \quad x\, R_\ell^X\, y \ \wedge\ r_\ell(x) = r_\ell(y).$$

These relations are reflexive because both conjuncts are reflexive.

The refinement preserves X's carrier and restricts its relations to originally related inputs with equal releases, thereby weakening the morphism condition of $\mathbf{CSet}_L$, *i.e.* eq. (1). If every release is constant, $X[\boldsymbol{r}] = X$, while when a single release r is used at every observer, write $X[r]$.

The following theorem identifies the release-bounded condition of eq. (2) with the ordinary classified-set morphism condition on the refined source $X[\boldsymbol{r}]$.

Theorem 4 (Release-bounded as a Morphism$^{\mathsf{L}\exists\forall\mathsf{N}}$). *A function $f : |X| \to |Y|$ is a morphism $X[\boldsymbol{r}] \to Y$ iff it satisfies eq.* (2) *at every level ℓ.*

Proof. Expand the source relation in the definition of a classified-set morphism.

Categorical Characterization. The release refinement $X[\boldsymbol{r}]$ is a pullback in $\mathbf{CSet}_L$ that combines the original indistinguishability with the authorized release. Let $V = \prod_{\ell \in L} V_\ell$, then define D on V by $v\, R_\ell^D\, w$ *iff* $v_\ell = w_\ell$, and let ∇V have carrier V and total relations at every observer. Finally, let $r(x) = (r_\ell(x))_{\ell \in L}$.

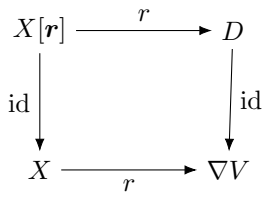


Fig. 2. Release refinement is a pullback in $\mathbf{CSet}_L$.

$X[\boldsymbol{r}]$ in fig. 2 is a pullback$^{\mathsf{L}\exists\forall\mathsf{N}}$: all four legs are morphisms, since the maps into ∇V have codiscrete target and the refinement ensures preservation by the left identity and top release map. Both routes have underlying function r, so the square commutes. For a commuting cone $f : Z \to X$, $g : Z \to D$, the mediating function must be f. Preservation by f and $g = r \circ f$ gives both conjuncts of the refined relation, proving existence and uniqueness of the mediator and hence the pullback property [14, § 2.1].

Thus, by the pullback's universal property, a morphism into $X[\boldsymbol{r}]$ preserves the original security relations and equality of release coordinates.

3.2 $\mathbf{DSet}_L$: The Category of Dynamic Classified Sets

Here, we define $\mathbf{DSet}_L$ and prove that it is a category. To facilitate the proofs, instead of directly using the release maps, we use their kernels. For a release map r and an observer-indexed release family $\boldsymbol{r}$, define

$$K_r \doteq \mathrm{Ker}(r), \quad K_{\boldsymbol{r}} \doteq (K_{r_\ell})_{\ell \in L}, \quad \text{and} \quad K_r(x, y) \ \dot{\iff}\ r(x) = r(y).$$

We call these kernels *release bounds*. Since they capture release equality and simplify reasoning about composition, we use them to define the arrows of $\mathbf{DSet}_L$.

Definition 2 (The $\mathbf{DSet}_L$ Category). *For the fixed observer set L, the objects of $\mathbf{DSet}_L$ are classified sets. The arrows $X \to Y$ are pairs (f, K), where $f : |X| \to |Y|$ and $K = (K_\ell)_{\ell \in L}$ is a family of equivalence relations on $|X|$, satisfying*

$$x \, R_\ell^X \, y \;\wedge\; K_\ell(x, y) \quad \Longrightarrow \quad f(x) \, R_\ell^Y \, f(y).$$

Two arrows are equal when their underlying functions and their bounds are equal. Identities and composition are defined as follows:

$$1_X = (\mathrm{id}, \top), \qquad (g, H) \circ (f, K) = (g \circ f, K \cap f^{-1}H),$$

where $(f^{-1}H)_\ell(x, y)$ means $H_\ell(f(x), f(y))$, intersection is observer-wise and $\top$ is the total relation at every observer.

Composition of $\mathbf{DSet}_L$ arrows accumulates the authorized releases by pairing the first release with the second evaluated on the intermediate result. Specifically, consider arrows $(f, K_{\boldsymbol{r}}) : X \to Y$ and $(g, K_{\boldsymbol{s}}) : Y \to Z$, whose release maps are, *resp.*, $r_\ell : |X| \to V_\ell$ and $s_\ell : |Y| \to W_\ell$. The composed release and its kernel are

$$t_\ell(x) = \big(r_\ell(x), s_\ell(f(x))\big) \text{ and } K_{t_\ell}(x, y) \iff r_\ell(x) = r_\ell(y) \wedge s_\ell(f(x)) = s_\ell(f(y)).$$

Thus, the composite bound is independent of how release tuples are grouped.

Bridge. By theorem 4, the pair $(f, K_{\boldsymbol{r}}) : X \to Y$ is an arrow in $\mathbf{DSet}_L$ exactly when $f : X[\boldsymbol{r}] \to Y$ is a morphism in $\mathbf{CSet}_L$.

Theorem 5 (Category laws$^{\mathsf{L}\exists\forall\mathsf{N}}$). *$\mathbf{DSet}_L$ is a category.*

Proof. We must show that the identity arrows are valid, composite arrows are valid, and the category laws hold for both components of each arrow. Most of the verification is routine and is given in Appendix A.5.

The interesting part is why security survives composition. For $(f, K) : X \to Y$ and $(g, H) : Y \to Z$, the second bound H constrains intermediate values. Its pullback (inverse image) along f, defined by $(f^{-1}H)_\ell(x, y) \iff H_\ell(f(x), f(y))$, expresses this constraint on the original inputs. If $x \, R_\ell^X \, y$ and $(K \cap f^{-1}H)_\ell(x, y)$, then $K_\ell(x, y)$ lets security of (f, K) give $f(x) \, R_\ell^Y \, f(y)$. The pulled-back bound supplies $H_\ell(f(x), f(y))$, so security of (g, H) then gives $g(f(x)) \, R_\ell^Z \, g(f(y))$. Thus the composite bound lets us apply the two security guarantees in sequence.

3.3 $\Diamond_{\boldsymbol{\pi}}$: The Redaction Modality

Next, we extend the redaction $\Diamond_\pi$ (of § 2.3) to $\mathbf{DSet}_L$, so that it changes observational relations while preserving computations and release bounds.

Definition 3 (Redaction $\Diamond_\pi$). *For $\pi \subseteq L$, the redaction $\Diamond_\pi X$ protects values from observers in π by making every pair indistinguishable to them. We define it by preserving the carrier and making the relation total at every observer in π:*

$$R_\ell^{\Diamond_\pi X}(x,y) \quad \dot{\iff} \quad \ell \in \pi \ \vee \ R_\ell^X(x,y).$$

On arrows it preserves both the computation and its release bound:

$$\Diamond_\pi(f,K) \doteq (f,K) : \Diamond_\pi X \to \Diamond_\pi Y \qquad \big((f,K) : X \to Y\big).$$

The arrow now has the redacted objects as its source and target, so its security condition uses $R_\ell^{\Diamond_\pi X}$ and $R_\ell^{\Diamond_\pi Y}$.

To establish that this construction is a modality, we prove that it is an idempotent monad, which includes functoriality, naturality, and the monad laws.

Theorem 6 (Redaction Modality$^{\mathsf{L}\exists\forall\mathsf{N}}$). *For every $\pi \subseteq L$, $\Diamond_\pi$ is an endofunctor on $\mathbf{DSet}_L$. With unit $\eta_X \doteq (\mathrm{id}, \top) : X \to \Diamond_\pi X$ and multiplication $\mu_X \doteq (\mathrm{id}, \top) : \Diamond_\pi \Diamond_\pi X \to \Diamond_\pi X$, it forms an idempotent monad on $\mathbf{DSet}_L$.*

Proof. Redaction preserves secure arrows: at $\ell \in \pi$ the target relation is total; elsewhere, the original security condition applies. Functions and release bounds are unchanged, so identities and composition are preserved.

The unit is secure since $R_\ell^X \subseteq R_\ell^{\Diamond_\pi X}$. Redacting twice has the same effect as once, so $\Diamond_\pi \Diamond_\pi X = \Diamond_\pi X$ and μ_X is an identity isomorphism. The proofs of naturality and the monad laws are routine and given in Appendix A.5.

3.4 Release-Relative Semantic Noninterference

Finally, we prove that arrows in $\mathbf{DSet}_L$ respect release-relative semantic NI. We say that X is *protected* at ℓ if R_ℓ^X is total and Y is *visible* there if R_ℓ^Y is equality.

Theorem 7 (Release-relative Semantic NI$^{\mathsf{L}\exists\forall\mathsf{N}}$). *Suppose $(f,K) : X \to Y$ is an arrow in $\mathbf{DSet}_L$ and Y is visible at ℓ. Then*

$$\forall x, y.\ x\, R_\ell^X\, y \ \wedge \ K_\ell(x,y) \quad \Longrightarrow \quad f(x) = f(y).$$

In particular, when $K_\ell = K_{r_\ell}$, this gives

$$\forall x, y.\ x\, R_\ell^X\, y \ \wedge \ r_\ell(x) = r_\ell(y) \quad \Longrightarrow \quad f(x) = f(y).$$

If X is protected at ℓ, R_ℓ^X is total, so $x\, R_\ell^X\, y$ trivially holds for all x and y, and:

$$\forall x, y.\ r_\ell(x) = r_\ell(y) \quad \Longrightarrow \quad f(x) = f(y).$$

Proof. The security condition of (f,K) gives $f(x)\, R_\ell^Y\, f(y)$, and visibility turns this relation into equality. When $K_\ell = K_{r_\ell}$, the bound condition is exactly $r_\ell(x) = r_\ell(y)$. Protection makes the initial-relation premise automatic.

Thus, for a protected input and a visible output, the result depends only on the authorized release: for nonempty π, choose $\ell \in \pi$ and suppose B is visible at ℓ. Since $\Diamond_\pi A$ is protected at ℓ, the theorem applies to $(f, K_{\boldsymbol{r}}) : \Diamond_\pi A \longrightarrow B$. So, f is constant on each release class at ℓ. Constant releases thus recover the classified-set constancy results of [14, Propositions 13 and 22], though not their typed-language theorems.

$$
\begin{aligned}
A, B &::= \mathsf{Bool} \mid \mathsf{Unit} \mid A \times B \mid T_\ell\ A, \\
v, u &::= \mathsf{true} \mid \mathsf{false} \mid () \mid \langle v, u \rangle \mid [v]_\ell \\
M, N &::= x \mid \mathsf{true} \mid \mathsf{false} \mid () \mid \langle M, N \rangle \\
&\quad \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid [M]_\ell \mid \mathsf{let}\ x \leftarrow M\ \mathsf{in}\ N \\
&\quad \mid \mathsf{policy}\ \delta\ \mathsf{in}\ M \mid \mathsf{let}\ x = M\ \mathsf{in}\ N.
\end{aligned}
$$

Fig. 3. Syntax of DDCC. It is DCC extended with explicit policy-change and sequential-binding constructs, which are blue. Here δ ranges over names of policy changes.

4 DDCC: Syntax and Interpretation in $\mathbf{DSet}_L$

Dynamic DCC (DDCC) extends DCC with explicit, certified policy changes. We interpret its types as objects and its typing derivations as morphisms in $\mathbf{DSet}_L$.

§ 4.1 presents the syntax and typing rules. § 4.2 defines the authorized-release maps and illustrates them with a patient–hospital–doctor example. § 4.3 defines indistinguishability and the security certificates required for policy changes. § 4.4 develops the categorical interpretation and proves soundness, adequacy, and NI. Later, § 5 extends DDCC with conditionals, functions, and recursion.

4.1 Syntax, Policy Changes, and Typing

Figure 3 presents the types (A, B), values (v, u) and expressions (M, N) of DDCC.

Types include the first-order fragment of DCC: booleans, unit, pairs, and protected values. Each type is inhabited by its expected corresponding values.

Expressions include the DCC fragment of variables, booleans, unit, pairs, and protected values. We call these expressions static and extend them with two dynamic constructs that can change the policy. The policy change $\mathsf{policy}\ \delta\ \mathsf{in}\ M$ applies the named policy change δ to the computation M. General let-bindings sequence static and dynamic expressions.

A policy p is a preorder $\sqsubseteq_p$ on L, with the assertion $\ell_L \sqsubseteq_p \ell_R$ meaning that information at label ℓ_L may flow to label ℓ_R under p. We write $\delta \triangleq p_i \rightsquigarrow p_o$ to name a policy change from p_i to p_o, potentially altering the flow preorder.

Typing. Figure 4 defines the dynamic and static typing rules. At the top, the judgment $\Gamma; p_i \vdash M : A; p_o$ assigns type A to M in context Γ, starting at input policy p_i and ending at output policy p_o.

The Change rule types $\mathsf{policy}\ \delta\ \mathsf{in}\ M$. For a policy change $\delta \triangleq p_i \rightsquigarrow p$, it checks the body M starting at p and records the body's final policy p_o, without restoring p_i. Its premise $\mathsf{Secure}(\Gamma, \delta)$, formalized in definition 6, requires a proof that the policy change exposes only information explicitly authorized for disclosure. The Let rule checks M_1 from p_i to p, then checks M_2 from p to p_o, in a context extended with the result of M_1. Finally, Static embeds a static expression at policy p into a dynamic judgment with both policies set to p.

The static typing judgment $\Gamma \vdash_p M : A$ in the middle of fig. 4 assigns type A to M in context Γ at a fixed policy p. Static expressions contain neither policy changes nor general let-bindings. Most rules are standard; the policy-related rules

Dynamic Policy Typing $\boxed{\Gamma; p_i \vdash M : A; p_o}$

$$\frac{\Gamma \vdash_p M : A}{\Gamma; p \vdash M : A; p}\ \textsc{Static} \qquad \frac{\delta \triangleq p_i \rightsquigarrow p \qquad \mathsf{Secure}(\Gamma, \delta) \qquad \Gamma; p \vdash M : A; p_o}{\Gamma; p_i \vdash \mathsf{policy}\ \delta\ \mathsf{in}\ M : A; p_o}\ \textsc{Change}$$

$$\frac{\Gamma; p_i \vdash M_1 : A; p \qquad \Gamma, x : A; p \vdash M_2 : B; p_o}{\Gamma; p_i \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : B; p_o}\ \textsc{Let}$$

Static Policy Typing $\boxed{\Gamma \vdash_p M : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash_p x : A}\ \textsc{Var} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash_p b : \mathsf{Bool}}\ \textsc{Bool} \qquad \frac{}{\Gamma \vdash_p () : \mathsf{Unit}}\ \textsc{Unit}$$

$$\frac{\Gamma \vdash_p M : A \qquad \Gamma \vdash_p N : B}{\Gamma \vdash_p \langle M, N \rangle : A \times B}\ \textsc{Pair} \qquad \frac{\Gamma \vdash_p M : A \times B}{\Gamma \vdash_p \mathsf{fst}\, M : A}\ \textsc{Fst} \qquad \frac{\Gamma \vdash_p M : A \times B}{\Gamma \vdash_p \mathsf{snd}\, M : B}\ \textsc{Snd}$$

$$\frac{\Gamma \vdash_p M : A}{\Gamma \vdash_p [M]_\ell : T_\ell\ A}\ \textsc{Unit} \qquad \frac{\Gamma \vdash_p M : T_\ell\ A \qquad \Gamma, x : A \vdash_p N : B \qquad p \vdash \ell \preceq B}{\Gamma \vdash_p \mathsf{let}\ x \leftarrow M\ \mathsf{in}\ N : B}\ \textsc{Bind}$$

Protection $\boxed{p \vdash \ell \preceq A}$

$$\frac{}{p \vdash \ell \preceq \mathsf{Unit}} \qquad \frac{p \vdash \ell \preceq A \qquad p \vdash \ell \preceq B}{p \vdash \ell \preceq A \times B} \qquad \frac{\ell \sqsubseteq_p \ell'}{p \vdash \ell \preceq T_{\ell'}\ A} \qquad \frac{p \vdash \ell \preceq A}{p \vdash \ell \preceq T_{\ell'}\ A}$$

Fig. 4. Dynamic DCC typing. Static typing embeds via Static; matching endpoints do not imply it. Static premises contain no policy changes or general let-bindings. The predicate $\mathsf{Secure}(\Gamma, \delta)$ is a premise of Change.

are Unit and Bind. The Unit rule wraps an expression of type A at level ℓ, giving $[M]_\ell$ type $T_\ell\ A$. The Bind rule unwraps a protected value as $x : A$ in its body, but requires the body's result type B to protect ℓ under p.

The protection judgment $p \vdash \ell \preceq A$, at the bottom of fig. 4, states that the type A protects information at level ℓ under policy p. Unit always protects ℓ, a product does so when both components do, and $T_{\ell'}\ A$ does so when either A already protects ℓ or $\ell \sqsubseteq_p \ell'$, where the policy p can be dynamically changed.

4.2 Authorized Release

An authorized release specifies which information a policy change is allowed to disclose. Before we provide the formal definition, we illustrate the concept with the hospital example from the introduction: a hospital copies patient data, revokes patient-to-hospital access, then passes its stored copy to a doctor.

Table 1 summarizes the steps of the hospital example. There are four security levels: P for patient, H for hospital, D for doctor, and E for external. Let

Table 1. The hospital example, with four security levels: P for patient, H for hospital, D for doctor, and E for external. **Code** is the DDCC program, Γ and ρ are the typing context and runtime environment, *resp.*, at each step of the program. **Release** indicates the information explicitly authorized for disclosure by a policy change, and **Policy** shows the policy at each step, where R is the reflexive relation on each security level.

Code	Γ	ρ	**Release**	**Policy**
0 *Input*	$\Gamma \doteq \{p : T_P\ \mathsf{Bool}, e : T_E\ \mathsf{Bool}\}$	$\rho \doteq \{p \mapsto [h_v]_P, e \mapsto [e_v]_E\}$	—	$R \cup \{(P,H)\}$
1 $\mathsf{let}\ h =$	Γ	ρ	—	$R \cup \{(P,H)\}$
2 $\quad(\mathsf{let}\ x \leftarrow p\ \mathsf{in}\ [x]_H)\ \mathsf{in}$	$\Gamma_2 \doteq \Gamma, h : T_H\ \mathsf{Bool}$	$\rho_2 \doteq \rho[h \mapsto [h_v]_H]$	$\star$	$R \cup \{(P,H)\}$
3 $\mathsf{policy\ discharge\ in}$	Γ_2	ρ_2	$\star$	$R \cup \{(P,H)\} \rightsquigarrow R$
4 $\mathsf{policy\ transfer\ in}$	Γ_2	ρ_2	h_v	$R \rightsquigarrow R \cup \{(H,D)\}$
5 $\mathsf{let}\ y \leftarrow h\ \mathsf{in}[y]_D$	$\Gamma_5 \doteq \Gamma_2, y : \mathsf{Bool}$	$\rho_5 \doteq \rho_2[y \mapsto h_v]$	$\star$	$R \cup \{(H,D)\}$

$R \doteq \{(P,P),(H,H),(D,D),(E,E)\}$ be the *reflexive policy*, which only allows information to flow within the same security level. The example uses three policies: 1) *data collection:* $\sqsubseteq_{p_c} \doteq R \cup \{(P,H)\}$, 2) *patient discharge:* $\sqsubseteq_p \doteq R$, and 3) *data sharing:* $\sqsubseteq_{p_s} \doteq R \cup \{(H,D)\}$. The example also uses two named policy changes one for patient's $\mathsf{discharge} \triangleq p_c \rightsquigarrow p$ and one for the $\mathsf{transfer} \triangleq p \rightsquigarrow p_s$.

Line 0: The program starts with an initial policy that permits information to flow from the patient to the hospital, *i.e.* $R \cup \{(P,H)\}$. Further, it starts with a typing environment that contains the binding p that is protected at the patient security level ($T_P\ \mathsf{Bool}$) and a binding e that is protected at the external security level ($T_E\ \mathsf{Bool}$). Our example does not actually use the external binding e, yet its existence showcases that unrelated data do not affect the released information.

Lines 1–2: The example starts by creating a hospital-protected copy of the patient data. Using the static DCC bind, the patient-protected value p is read and returned as a hospital-protected value. This is permitted, since the policy allows information to flow from the patient to the hospital, *i.e.* $P \sqsubseteq_{p_c} H$. Using a let binding, the hospital-protected copy is assigned to a new variable h, and the two environments are accordingly extended.

Line 3: The program mimics the patient's discharge with the `discharge` policy change that cuts the information flow from the patient to the hospital. No information is authorized to be released at this policy change, thus the release column of table 1 is empty (*i.e.* $\star$).

Line 4: Later, the program simulates the transfer of the patient's data to a specialized doctor, with the policy change `transfer` that allows hospital-protected information to flow to that doctor. Concretely, the h_v value, which is protected at the hospital level, is authorized to be released, as marked in the release column

of table 1. Note that, for simplicity, we use the same release at every observer: even though h_v is newly visible only to the doctor, it is authorized at every label.

Line 5: Finally, the information h, which is protected at the hospital level but originally derived from the patient, is released to the doctor.

In short, only the `transfer` policy change releases information. For the complete program (call it M), the accumulated release is the tuple:

$$\mathsf{release}_M(\rho) = (\star, (\star, (h_v, \star))).$$

Now, we can formalize the notation of single and accumulated releases.

Write $|\Gamma|$ for the set of environments assigning values of the declared types to all variables in Γ and V for a set of (release) values.

Definition 4 (Authorized Release). *An authorized release specifies which information a single policy change may disclose. For each change $\delta \triangleq p_i \rightsquigarrow p_o$ and context Γ, the policy author supplies an authorized release map*

$$\mathsf{release}_{\delta,\Gamma} : |\Gamma| \to V.$$

Note that the authorized release is part of the fixed authorization specification for each policy change.

Definition 5 (Accumulated Release). *For a typing derivation $\Gamma; p_i \vdash M : A; p_o$, define the accumulated release of the term M recursively by*

$$\begin{aligned}
\mathsf{release}_M &: |\Gamma| \to V \\
\mathsf{release}_M(\rho) &\doteq \star && \textit{(M static)} \\
\mathsf{release}_{\mathsf{policy}\ \delta\ \mathsf{in}\ M}(\rho) &\doteq (\mathsf{release}_{\delta,\Gamma}(\rho), \mathsf{release}_M(\rho)) \\
\mathsf{release}_{\mathsf{let}\ x=M_1\ \mathsf{in}\ M_2}(\rho) &\doteq (\mathsf{release}_{M_1}(\rho), \mathsf{release}_{M_2}(\rho_1))
\end{aligned}$$

where ρ_1 extends ρ with the binding for x using the interpretation that we will define in § 4.4, i.e. $\rho_1 = \rho[x \mapsto [\![M_1]\!]_{p_i,q}(\rho)]$ with q being the output policy of M_1.

4.3 Indistinguishability and Security Certificates

A security certificate ensures that a policy change exposes only information permitted by its authorized release. To state this condition, we first define when values and environments are indistinguishable to an observer.

Indistinguishable Types For a fixed observer $\ell_o \in L$ and policy p, we recursively define R^A_{p,ℓ_o} on values of type A:

$$\begin{aligned}
u\, R^{\mathsf{Bool}}_{p,\ell_o}\, v &\iff u = v, \\
u\, R^{\mathsf{Unit}}_{p,\ell_o}\, v &\iff \mathsf{True}, \\
u\, R^{A\times B}_{p,\ell_o}\, v &\iff u.1\, R^A_{p,\ell_o}\, v.1 \ \wedge\ u.2\, R^B_{p,\ell_o}\, v.2, \\
u\, R^{T_\ell\ A}_{p,\ell_o}\, v &\iff \big(\ell \sqsubseteq_p \ell_o \implies u.\mathsf{value}\, R^A_{p,\ell_o}\, v.\mathsf{value}\big).
\end{aligned}$$

A protected value is compared only when its label can flow to the observer; otherwise all values of that protected type are indistinguishable.

Indistinguishable Environments For a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we define environment indistinguishability componentwise, for each $\rho_L, \rho_R \in |\Gamma|$.

$$\rho_L \, R^{\Gamma}_{p,\ell_o} \, \rho_R \quad \dot{\iff} \quad \bigwedge_{j=1}^{n} \rho_L(x_j) \, R^{A_j}_{p,\ell_o} \, \rho_R(x_j).$$

Release-refined Environments Given a release $r : |\Gamma| \to V$, the environment indistinguishability can be refined to require equal released values:

$$\rho_L \, R^{\Gamma,r}_{p,\ell_o} \, \rho_R \quad \dot{\iff} \quad \rho_L \, R^{\Gamma}_{p,\ell_o} \, \rho_R \ \wedge \ r(\rho_L) = r(\rho_R).$$

Note that this definition exactly instantiates the release refinement of § 3.1.

Security Certificates Using these relations and the declared release, we state the security predicate required by the CHANGE rule. Each policy change requires a proof of this predicate, which ranges over every observer and pair of environments.

Definition 6 (Secure Policy Change). *A policy change $\delta \triangleq p_i \rightsquigarrow p_o$ is secure in a context Γ if, for every observer ℓ, environments that are initially indistinguishable and agree on the authorized release remain indistinguishable:*

$$\mathsf{Secure}(\Gamma, \delta) \doteq \forall \ell \in L, \rho_L, \rho_R \in |\Gamma|. \ \rho_L \, R^{\Gamma,\mathsf{release}_{\delta,\Gamma}}_{p_i,\ell} \, \rho_R \implies \rho_L \, R^{\Gamma}_{p_o,\ell} \, \rho_R.$$

We call a proof of $\mathsf{Secure}(\Gamma, \delta)$ a *security certificate*. The CHANGE rule requires this proof in the context of use. For example, when typing the program of table 1, `discharge` preserves indistinguishability because it only removes a flow. The `transfer` policy change satisfies $\mathsf{Secure}(\Gamma_2, \texttt{transfer})$ because equal release values force equal hospital values, exactly the new equality required by the doctor, while the relations at all other observers are unchanged.

Note on computability In the first-order fragment of DDCC, every type has finitely many values, so each context has finitely many environments. With a finite label set, decidable policy comparisons, computable release maps, and decidable equality on release values, the security predicate is decidable: it can be computed by enumerating all observers and pairs of environments. This exhaustive check is, of course, too expensive in practice. Computationally efficient sound approximations could instead check sufficient conditions for security. But developing such approximations is not in the scope of this paper, since we focus on the development of the semantic criterion for accepting policy changes.

4.4 Interpretation and Noninterference in $\mathbf{DSet}_L$

We combine the releases of § 4.2 and the relations and certificates of § 4.3 to interpret typing judgments in $\mathbf{DSet}_L$. Soundness and adequacy then yield NI.

Types are interpreted in $\mathbf{DSet}_L$ as classified sets:

$$[\![A]\!]_p \doteq \big(|A|, (R^A_{p,\ell_o})_{\ell_o \in L}\big),$$

where $|A|$ is its policy-independent value carrier and R^A_{p,ℓ_o} is the observer relation defined in § 4.3. The carriers of DDCC types are

$$|\mathsf{Bool}| \doteq \mathbb{B} \quad |A \times B| \doteq |A| \times |B| \quad |\mathsf{Unit}| \doteq \{\star\} \quad |T_\ell\ A| \doteq \{[a]_\ell \mid a \in |A|\}.$$

The protected wrappers are canonically identified with their underlying values via $[a]_\ell \mapsto a$. Under this identification, $[\![T_\ell\ A]\!]_p \doteq \Diamond_{\downarrow^p_\ell}[\![A]\!]_p$, where $\downarrow^p_\ell \doteq \{\ell_o \in L \mid \ell \not\sqsubseteq_p \ell_o\}$, which is the redaction modality of § 3.3.

Environments are interpreted as products of their constituent types:

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!]_p \doteq \prod_{j=1}^{n} [\![A_j]\!]_p.$$

The classified-set product combines both the carrier with the observer relations, generating the classified set $[\![\Gamma]\!]_p \doteq \big(|\Gamma|, (R^\Gamma_{p,\ell_o})_{\ell_o \in L}\big)$, with the observer relation defined in § 4.3 as a conjunction of the type relations. The empty environment is denoted by $\star$.

The release-refined interpretation of an environment has the same carrier, but refines the observer relation to account for the release function. For a release $r : |\Gamma| \to V$ and the release-refined relation $R^{\Gamma,r}_{p,\ell_o}$ of § 4.3, we define:

$$[\![\Gamma]\!]_p[r] \doteq \big(|\Gamma|, (R^{\Gamma,r}_{p,\ell_o})_{\ell_o \in L}\big).$$

Terms are interpreted based on their typing derivation. For a static derivation $\Gamma \vdash_p M : A$, the underlying term interpretation, $[\![M]\!]_p : |\Gamma| \to |[\![A]\!]_p|$, is defined as follows:

$$\begin{aligned}
[\![x]\!]_p(\rho) &\doteq \rho(x) & [\![()]\!]_p(\rho) &\doteq \star,\\
[\![\mathsf{true}]\!]_p(\rho) &\doteq \mathsf{true} & [\![\mathsf{fst}\ M]\!]_p(\rho) &\doteq \pi_1\big([\![M]\!]_p(\rho)\big),\\
[\![\mathsf{false}]\!]_p(\rho) &\doteq \mathsf{false} & [\![\mathsf{snd}\ M]\!]_p(\rho) &\doteq \pi_2\big([\![M]\!]_p(\rho)\big),\\
[\![\langle M, N\rangle]\!]_p(\rho) &\doteq \big([\![M]\!]_p(\rho), [\![N]\!]_p(\rho)\big)\\
[\![[M]_\ell]\!]_p(\rho) &\doteq \eta^{p,\ell}_{[\![A]\!]_p}\big([\![M]\!]_p(\rho)\big) \quad (M : A)\\
[\![\mathsf{let}\ x \leftarrow M\ \mathsf{in}\ N]\!]_p(\rho) &\doteq [\![N]\!]_p\big(\rho[x \mapsto [\![M]\!]_p(\rho).\mathsf{value}]\big)
\end{aligned}$$

The policy index determines the unit of the redaction modality, $\eta^{p,\ell}_X : X \to \Diamond_{\downarrow^p_\ell} X$.

For a dynamic derivation $\Gamma; p_i \vdash M : A; p_o$, write $[\![M]\!]_{p_i,p_o} : |\Gamma| \to |[\![A]\!]_{p_o}|$, with the derivation determining the policies of its subterms:

$$\begin{aligned}
[\![S]\!]_{p_i,p_i}(\rho) &\doteq [\![S]\!]_{p_i}(\rho) \qquad \textit{for a } \textsc{Static} \textit{ embedding},\\
[\![\mathsf{policy}\ \delta\ \mathsf{in}\ M]\!]_{p_i,p_o}(\rho) &\doteq [\![M]\!]_{q,p_o}(\rho) \qquad (\delta \triangleq p_i \rightsquigarrow q),\\
[\![\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2]\!]_{p_i,p_o}(\rho) &\doteq [\![M_2]\!]_{q,p_o}\big(\rho[x \mapsto [\![M_1]\!]_{p_i,q}(\rho)]\big)
\end{aligned}$$

Typing Derivations are interpreted categorically as morphisms in $\mathbf{DSet}_L$. Consider the derivation $\Gamma; p_i \vdash M : A; p_o$. Its interpretation pairs the underlying computation with the kernel of its accumulated release, using the same release at every observer:

$$[\![\Gamma; p_i \vdash M : A; p_o]\!] \doteq \big([\![M]\!]_{p_i,p_o}, (K_{\mathsf{release}_M})_{\ell \in L}\big) : [\![\Gamma]\!]_{p_i} \longrightarrow [\![A]\!]_{p_o}.$$

To show that this pair is an arrow in $\mathbf{DSet}_L$, we construct a proof pr that initially indistinguishable environments with equal releases produce results related under the output policy:

$$pr : \forall \ell_o, \rho_L, \rho_R.\ \rho_L \, R^{\Gamma, \mathsf{release}_M}_{p_i,\ell_o} \, \rho_R \implies [\![M]\!]_{p_i,p_o}(\rho_L) \, R^{A}_{p_o,\ell_o} \, [\![M]\!]_{p_i,p_o}(\rho_R)$$

The DDCC soundness theorem constructs this proof and hence establishes that the displayed pair is a morphism in $\mathbf{DSet}_L$.

Theorem 8 (DDCC Soundness$^{\mathsf{L}\exists\forall\mathsf{N}}$). *Assume each policy supplies a preorder relation. The categorical interpretation of the typing derivation of fig. 4:*

$$[\![\Gamma; p_i \vdash M : A; p_o]\!] : [\![\Gamma]\!]_{p_i} \longrightarrow [\![A]\!]_{p_o}$$

is a morphism in $\mathbf{DSet}_L$.

Proof. The proof relies on a stronger invariant that relates the complete context and the result. Concretely, by induction on the typing derivation, we prove that the following map is a morphism in $\mathbf{CSet}_L$:

$$\big(\rho \mapsto (\rho, [\![M]\!]_{p_i,p_o}(\rho))\big) : [\![\Gamma]\!]_{p_i}[\mathsf{release}_M] \longrightarrow [\![\Gamma]\!]_{p_o} \times [\![A]\!]_{p_o}.$$

Thus, initially related environments with equal releases give both related environments and related results under the output policy. Both components are needed for Let: its continuation receives the original variables together with the first computation's result. Projecting onto the result component gives the required proof pr.

The static case uses preservation by induction on static typing. Changes use their context certificates; let extends related environments with related results and composes paired releases. By theorem 4, this preservation property establishes the claimed morphism in $\mathbf{DSet}_L$, with underlying function $[\![M]\!]_{p_i,p_o}$ and release bound $(K_{\mathsf{release}_M})_{\ell \in L}$. Appendix A.4 gives the detailed proof.

Adequacy maps term denotations back to their canonical values.

Theorem 9 (DDCC Adequacy$^{\mathsf{L}\exists\forall\mathsf{N}}$). *For a closed derivation $\varnothing; p_i \vdash M : A; p_o$, M normalizes to a unique value $v : A$ satisfying $[\![M]\!]_{p_i,p_o}(\star) = [\![v]\!]_{p_o}(\star)$. So, equal semantic results of closed terms at the same type imply program equality.*

Proof. Normalization agrees with the interpretation and value denotations are injective by induction on first-order types. Thus equal denotations determine the same canonical form. Appendix A.3 gives the evaluation rules; Appendix A.4 proves normalization and adequacy.

Equal Visible Results As we are approaching to the noninterference proof, we need to compare open DDCC terms. We use a *closing substitution* to assign values to all free variables in an open term, converting it into a closed term. Concretely, when σ assigns a closed canonical value $\sigma(x) : A$ to each variable $x : A$ in Γ, we write $\Gamma \models \sigma$. We write $\vdash M[\sigma] = N[\tau] : A$ when the closed terms have the same canonical normal form and $M[a/x]$ for a one-variable substitution.

Corollary 1 (Equality of Visible Results$^{\mathsf{L}\exists\forall\mathsf{N}}$). *Assume every policy supplies a preorder and $\Gamma; p_i \vdash M : A; p_o$. If $[\![A]\!]_{p_o}$ is visible at ℓ_o, then for two canonical closing substitutions $\Gamma \models \sigma_L, \Gamma \models \sigma_R$ with corresponding semantic environments $\rho_{\sigma_L}, \rho_{\sigma_R} \in |\Gamma|$, where $\rho_{\sigma_k}(x) \doteq [\![\sigma_k(x)]\!]_{p_i}(\star)$ for $k \in \{L, R\}$, the following holds:*

$$\rho_{\sigma_L}\, R^{\Gamma,\mathsf{release}_M}_{p_i,\ell_o}\, \rho_{\sigma_R} \quad \Longrightarrow \quad \vdash M[\sigma_L] = M[\sigma_R] : A.$$

Proof. The soundness theorem 8 supplies the morphism

$$[\![\Gamma; p_i \vdash M : A; p_o]\!] : [\![\Gamma]\!]_{p_i} \to [\![A]\!]_{p_o},$$

whose underlying function is $[\![M]\!]_{p_i,p_o}$. By the definition of the refined relation, the premise gives $\rho_{\sigma_L}\, R^{\Gamma}_{p_i,\ell_o}\, \rho_{\sigma_R}$ and $\mathsf{release}_M(\rho_{\sigma_L}) = \mathsf{release}_M(\rho_{\sigma_R})$. Since $[\![A]\!]_{p_o}$ is visible at ℓ_o, the semantic noninterference theorem 7 in § 3.4 yields

$$[\![M]\!]_{p_i,p_o}(\rho_{\sigma_L}) = [\![M]\!]_{p_i,p_o}(\rho_{\sigma_R}).$$

Finally, the adequacy theorem 9 reflects this equality of semantic results into program equality, concluding that $\vdash M[\sigma_L] = M[\sigma_R] : A$.

Noninterference, as stated in fig. 1, comes as a specialization of the equality of visible results to protected inputs and boolean outputs.

Theorem 10 (Noninterference for Dynamic DCC$^{\mathsf{L}\exists\forall\mathsf{N}}$). *Suppose every policy supplies a preorder. If*

$$x : T_{Hi}\ A; p_i \vdash M : T_{Lo}\ \mathsf{Bool}; p_o, \qquad Hi \not\sqsubseteq_{p_i} Lo,$$

then for all closed canonical inputs $a_L, a_R : T_{Hi}\ A$,

$$\begin{aligned}\mathsf{release}_M(x \mapsto [\![a_L]\!]_{p_i}(\star)) = \mathsf{release}_M(x \mapsto [\![a_R]\!]_{p_i}(\star))\\ \Longrightarrow \quad \vdash M[a_L/x] = M[a_R/x] : T_{Lo}\ \mathsf{Bool}.\end{aligned}$$

Proof. Take the observer to be *Lo* throughout. The initial no-flow condition makes $R^{T_{Hi}\ A}_{p_i,Lo}$ total, because of its definition at § 4.3, so any two input values are related. Reflexivity gives $Lo \sqsubseteq_{p_o} Lo$; consequently, $R^{T_{Lo}\ \mathsf{Bool}}_{p_o,Lo}$ compares boolean values by equality. Soundness supplies the refined-context morphism, and theorem 7 gives equal output denotations whenever the releases agree. Theorem 9 turns this equality into program equality at $T_{Lo}\ \mathsf{Bool}$.

Verification of the hospital example. We verified in Lean$^{\mathsf{L}\exists\forall\mathsf{N}}$ operational noninterference for the hospital example of table 1. Concretely, we proved that the program type checks, thus that each policy change is secure with respect to the authorized releases. We also proved that equal accumulated releases give equal closed doctor results, independently of the external input: a concrete instance of operational noninterference. Finally, we confirmed that typing fails in two unauthorized scenarios:

- ***Failure 1:*** Replacing the transfer release with the constant $\star$ gives an unauthorized variant: $\mathsf{Secure}(\Gamma_2, \mathsf{transfer})$ fails, so the typing rule rejects this variant. Reachable hospital copies with different boolean values are indistinguishable to the doctor before the transfer and have equal constant releases, yet become distinguishable afterward. Lean proves that this variant has no valid security certificate and can produce different doctor results$^{\mathsf{L}\exists\forall\mathsf{N}}$.
- ***Failure 2:*** Extending the initial typing environment with an independent hospital-protected binder $k : T_H\ \mathsf{Bool}$, while the transfer release still authorizes only h. Keeping h fixed and varying k gives initially indistinguishable environments with equal releases that become distinguishable to the doctor after the `transfer` policy change: the new flow exposes k as well. Thus no whole-context certificate exists, even though the body returns only $h^{\mathsf{L}\exists\forall\mathsf{N}}$.

In short, DDCC combines DCC's protection discipline with certified policy changes to support controlled disclosure under evolving policies.

5 Extensions

We extended DDCC with three language features: § 5.1 conditionals, § 5.2 functions, and § 5.3 recursion. We summarize the language and interpretation changes below while Appendix B gives the detailed constructions. § 5.4 combines the three extensions in a recursive computation with an explicitly passed store.

5.1 Conditionals

The conditional extension lets us write in DDCC the program ExtC that encodes the public-choice-based authorization pattern of Sabelfeld and Myers [20, § 3.4]:

$$\begin{aligned}&\mathsf{ExtC} \doteq \mathsf{if}\ c\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\\ &P_i \quad \doteq \mathsf{let}\ y = \mathsf{policy}\ \mathsf{open}_i\ \mathsf{in}\ (\mathsf{let}\ z \leftarrow h_i\ \mathsf{in}\ [z]_\ell)\ \mathsf{in}\ \mathsf{policy}\ \mathsf{close}_i\ \mathsf{in}\ y\end{aligned}$$

where c is public and $h_i : T_{S_i}\ \mathsf{Bool}$. The `open` policy change adds $S_i \sqsubseteq \ell$ and authorizes h_i, while the `close` policy change revokes that edge. Both branches return to the same closed policy and only the selected secret is released.

Language Changes. We extended the language with a dynamic conditional, since each branch may perform different policy changes. The rule requires a pure boolean guard, to avoid secret-dependent control flow, and common input and output policies, while allowing different intermediate policies.

Interpretation Changes. Conditionals are interpreted as a choice between the interpretations of the two branches, so $\mathbf{DSet}_L$ requires no changes.

5.2 Functions

The security of a function depends on the policies under which it is defined and applied. For example, ExtF below changes the label of its input from ℓ_x to ℓ_o.

$$\mathsf{ExtF} \doteq \lambda x : T_{\ell_x}\ \mathsf{Bool}.\mathsf{let}\ z \leftarrow x\ \mathsf{in}\ [z]_{\ell_o}.$$

This function is secure as long as the flow $\ell_x \sqsubseteq_p \ell_o$ is allowed, but once the flow is revoked, the unchanged function is no longer secure$^{\mathsf{L}\exists\forall\mathsf{N}}$.

To address this challenge, each policy change supplies an *environment transport* $t_{\delta,\Gamma} : [\![\Gamma]\!]_{p_i} \to [\![\Gamma]\!]_{p_o}$, which maps the current environment to a valid environment under the new policy. We extend the Secure predicate of definition 6 to account for this transport: initially indistinguishable environments with equal authorized releases must yield indistinguishable transported environments. The CHANGE rule requires a certificate for the generalized Secure predicate. The first-order treatment of § 4 is recovered by taking the transport to be the identity.

ExtF can thus be called under any policy p with $\ell_x \sqsubseteq_p \ell_o$. If this flow is revoked, a transport can replace the stored function with a safe default, *e.g.* $\lambda x : T_{\ell_x}\ \mathsf{Bool}.[\mathsf{false}]_{\ell_o}$, so that subsequent calls use this safe replacement$^{\mathsf{L}\exists\forall\mathsf{N}}$.

Language Changes. We extended the static fragment of DDCC with arrow types $A \to B$, λ-abstraction, and application, all of which are checked at a fixed policy. An arrow type protects a label whenever its result type does.

Interpretation Changes. The interpretation of $\mathsf{policy}\ \delta\ \mathsf{in}\ M$ now applies $t_{\delta,\Gamma}$ to the current environment before interpreting M. Sequential let passes the first computation's transported environment, extended with its result, to the continuation. $\mathbf{DSet}_L$ remains unchanged. Appendix B.2 gives the full details.

5.3 Recursion

To support recursion, we introduce fixed-point terms $\mu f : A.M$, which bind f in its own body M. Recursive calls may diverge, so the result type of a recursive function must be a *pointed type*, *i.e.* one with a least element $\bot$. We therefore use lifted result types $A_\bot$: $\bot$ represents divergence, while $\mathsf{lift}\, M$ embeds a term $M : A$ into $A_\bot$. Following DCC's termination discipline [1, §5], a protected input must not control observable termination when its label does not flow to the observer. The following recursive function, for example, uses public control:

$$\begin{aligned}\mathsf{ExtR} \doteq \mu f : \mathsf{Bool} \to (T_{Lo}\ \mathsf{Bool})_\bot.\\ \lambda c : \mathsf{Bool}.\ \mathsf{if}\ c\ \mathsf{then}\ \mathsf{lift}\,[\mathsf{true}]_{Lo}\ \mathsf{else}\ f\, c.\end{aligned}$$

The function returns a lifted low-security true when c is true and repeats its call forever when c is false. Termination depends only on the public argument.

Language Changes. We extended the static fragment of DDCC with lifted types $A_\perp$, lifting, strict sequencing, and fixed points at types with a least element for divergence. Sequencing lets a continuation use the value of a returned lift; if the preceding computation diverges, sequencing also diverges. Protected bind requires protection of the result type. Protection of A does not imply protection of $A_\perp$, so hiding the returned value alone does not permit a protected input to control termination. Recursion is restricted to the static fragment, so recursive bodies cannot contain policy changes.

Interpretation Changes. Classified sets alone do not supply a general fixed-point operation, so we extend their carriers with ω-complete partial orders and interpret recursion by least fixed points of continuous maps, taking limits of successive approximations starting from the least element. The security relations must be *admissible*: limits of related increasing chains remain related, so the fixed-point interpretation preserves these relations. In $A_\perp$, divergence is related only to divergence while returned values use the original relation on A. Thus, related lifted computations always agree on whether they return. These ordered carriers still give classified sets, leaving $\mathbf{DSet}_L$ unchanged.

Extended soundness and NI. In the continuous admissible model, with certified total continuous context transports, the related initial environments with equal accumulated releases yield related transported contexts and results (Appendix, proposition 1). Policy changes remain restricted to a finite computation. The generic operational corollary in Appendix B.2 assumes adequacy; for our concrete call-by-name semantics, ground adequacy is proved, yielding termination-sensitive NI at protected lifted booleans under unrestricted static recursion$^{\mathsf{L}\exists\forall\mathsf{N}}$.

5.4 Putting it All Together

Store can be encoded in DDCC using higher-order functions and recursion. Let $S = (\mathsf{Bool} \times \mathsf{Bool}) \times T_{Hi}\ \mathsf{Bool}$ contain a public two-bit counter and a protected stored bit. The following function repeatedly applies a state update $u : S \to S$ until the counter is zero:

$$\begin{aligned}\mathsf{ExtAll} \doteq\ & \mu f : (S \to S) \to S \to S_\perp.\\ & \lambda u : S \to S.\ \lambda s : S.\\ & \mathsf{if}\ \mathsf{nonzero}(\mathsf{fst}\, s)\ \mathsf{then}\ f\, u\, (u\, s)\ \mathsf{else}\ \mathsf{lift}\, s.\end{aligned}$$

For a countdown, use the update $u\,\langle c, h\rangle \doteq \langle \mathsf{dec}\, c, \mathsf{let}\ b \leftarrow h\ \mathsf{in}\ [\mathsf{not}\, b]_{Hi}\rangle$, where nonzero, dec, and not abbreviate DDCC expressions built from conditionals, pairs, and projections. Starting at counter 11 and stored bit $[\mathsf{false}]_{Hi}$, the loop visits counters 10, 01, and 00, flipping the protected bit on each step before returning $\mathsf{lift}\,\langle 00, [\mathsf{true}]_{Hi}\rangle$. The public counter controls termination while the stored bit remains protected. An update that preserves a nonzero counter instead makes the loop run forever. Lean checks the repeated updates, the zero-counter case, the diverging cycle, and the changed bit returned by the call-by-name semantics$^{\mathsf{L}\exists\forall\mathsf{N}}$.

6 Related Work

Categorical Foundations. Classical noninterference (NI) excludes observable dependence on secrets [11,19]. DCC uses dependency types and logical relations to establish NI [1]. Kavvos recasts this argument using classified sets, pre-cohesion, and protected-to-visible constancy followed by adequacy [14]. We keep the classified objects, add release-bounded arrows, and extend the redaction modality. Protected-to-visible transformations then become constant within each release class. Kavvos' disclosure modality $\Box_\pi$ replaces the relations at π by equality and forms an idempotent comonad [14, § 5.1]. Our theorem 7 requires only equality of the target relation at the observer, including $\Box_\pi B$ when $\ell \in \pi$, while the source refinement and release bounds account for authorized disclosure.

Sterling and Harper [23] use sheaves and open/closed modalities to obtain termination-insensitive NI for DCC extended with a construct for declassifying termination channels. Our lifted-type discipline instead prevents secret-dependent observable termination. For protected lifted booleans, operational adequacy of the call-by-name semantics yields termination-sensitive NI in the concrete model.

Delimited Release and Declassification. Sabelfeld and Sands [21] organize declassification by *what*, *who*, *where*, and *when* information may be released. Our release maps specify *what*: authorization is declared independently of the computation. Banerjee, Giacobazzi, and Mastroeni [6] use completeness of abstract interpretation with respect to weakest liberal preconditions to detect disclosure beyond a policy and refine it using counterexamples. Our construction verifies and composes computations under a supplied authorization.

Our equal-release criterion is closest to Sabelfeld and Myers' delimited release [20]. It compares initially indistinguishable memories that agree on initial escape-hatch values and requires equal final observations when both runs terminate. Their security lattice and variable classifications are fixed. Their type-and-effect system prevents updates to variables used in subsequent escape hatches, ensuring that later disclosures respect the initial specification. In comparison, our source refinement expresses their equal-hatch criterion categorically. DDCC additionally permits changes to the flow preorder and certifies the whole current context against an independently declared release. These releases may depend on earlier computations. For a semantics-preserving DDCC encoding, recovering their guarantee additionally requires showing that initially indistinguishable memories with equal initial hatch values have equal accumulated releases. Their specification includes hatches from both conditional branches, while our public conditional tags only the selected branch's release. Their language also supports mutable state and while loops, while DDCC encodes state by passing a store through recursive calls.

Dynamic Policies and Attacker Knowledge. Dynamic policies admit different interpretations of time-transitive flow, replay, direct release, and whitelisting [7]. Gradual Release [4] restricts knowledge increases between release events. Askarov and Chong [3] parameterize attacker memory; Ahmadian and Balliu [2] require

policy updates to be consistent with prior knowledge. These accounts constrain what an attacker can learn during execution. Our first-order operational result (corollary 1) compares initially indistinguishable inputs with equal accumulated releases and guarantees equal observable final results after certified policy changes. Observer relations describe which values are distinguishable under each policy without recording an attacker's observation history. Thus, this final-result theorem alone establishes neither Gradual Release's restriction on knowledge growth nor Ahmadian and Balliu's consistency requirement.

Enforcement of Policy Changes. Rx [24] uses role-based labels and statically verified transactions to prevent time-transitive flows. Van Delft, Hunt, and Sands [10] infer dependencies independently of the policy, allowing one analysis to support multiple policy checks. SLIO [8] monitors changes to the label ordering, including secret-dependent updates. DAISY [12] tracks dependencies between application variables and database tuples. DDCC requires each policy change to carry a certificate that the whole current context preserves indistinguishability modulo its authorized release, for every observer, and proves that these certificates compose. Using these typing or monitoring mechanisms to generate our certificates requires proving that their checks imply our certificate condition.

Dynamic Release. Li and Zhang [16] unify downgrading and upgrading through Dynamic Release, with transient and persistent knowledge allowances. Ching and Zhang [9] give a sound type system for this policy and a Rust prototype. Dynamic Release constrains attacker knowledge at each output prefix. Connecting its enforcement theorem to our release-relative final-result theorem requires translating those allowances into release bounds and proving that the typing rules satisfy our certificate condition.

7 Conclusion

We extend the categorical noninterference argument from static to dynamic information-flow policies. To do so, we design DDCC, which adds explicit policy changes to DCC, certified against independently authorized releases. Then, we construct $\mathbf{DSet}_L$, which extends the category of classified sets with release-bounded transformations that compose by pairing their releases. Finally, we interpret DDCC in $\mathbf{DSet}_L$, which reduces noninterference to constancy within release classes: protected inputs with equal accumulated releases yield equal visible outputs. Soundness and operational adequacy of the DDCC interpretation connect the categorical guarantees to program behavior.

Our construction separates the reusable categorical argument from the language-specific interpretation, soundness, adequacy, and security certificates. Future languages can reuse our categorical argument to obtain noninterference by providing an interpretation and discharging the corresponding proof obligations.

AI Use All proofs developed for this paper, including the project-specific Lean proof scripts and their written presentations, were generated and revised by OpenAI Codex (GPT-6, October 2026) in an interactive process directed by the authors. Codex was also used to implement and debug the examples, run the verification commands, and edit their exposition. The mechanized proofs were checked by Lean 4.29.1 and depend only on Lean's standard axioms `propext`, `Classical.choice`, and `Quot.sound`. The authors remain responsible for the paper's claims, definitions, and assumptions, and for the correspondence between the written results and their formalization.

Data-Availability Statement A companion artifact is submitted, containing the Appendix, a README, the Lean 4.29.1 formalization, executable examples, and scripts for building the development. It mechanizes the results marked $^{\mathsf{L}\exists\forall\mathsf{N}}$; Appendix C records its scope and limitations. We plan to submit the artifact for artifact evaluation.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 147–160 (1999). https://doi.org/10.1145/292540.292555
2. Ahmadian, A.M., Balliu, M.: Dynamic policies revisited. In: IEEE European Symposium on Security and Privacy. pp. 448–466 (2022). https://doi.org/10.1109/EuroSP53844.2022.00035
3. Askarov, A., Chong, S.: Learning is change in knowledge: Knowledge-based security for dynamic policies. In: IEEE Computer Security Foundations Symposium. pp. 308–322 (2012). https://doi.org/10.1109/CSF.2012.31
4. Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: IEEE Symposium on Security and Privacy. pp. 207–221 (2007). https://doi.org/10.1109/SP.2007.22
5. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: IEEE Computer Security Foundations Symposium. pp. 43–59 (2009). https://doi.org/10.1109/CSF.2009.22
6. Banerjee, A., Giacobazzi, R., Mastroeni, I.: What You Lose is What You Leak: Information Leakage in Declassification Policies. Electronic Notes in Theoretical Computer Science **173**, 47–66 (2007). https://software.imdea.org/~ab/Publications/wyliwyl.pdf
7. Broberg, N., van Delft, B., Sands, D.: The anatomy and facets of dynamic policies. In: IEEE Computer Security Foundations Symposium. pp. 122–136 (2015). https://doi.org/10.1109/CSF.2015.16
8. Buiras, P., van Delft, B.: Dynamic enforcement of dynamic policies. In: Programming Languages and Analysis for Security. pp. 28–41 (2015). https://doi.org/10.1145/2786558.2786563
9. Ching, J., Zhang, D.: Sound Enforcement of Dynamic Release Information Flow Policy. Proceedings of the ACM on Programming Languages **10**(OOPSLA2), Article 360 (2026). https://doi.org/10.1145/3839492. Full version: https://arxiv.org/abs/2608.09506v2

10. van Delft, B., Hunt, S., Sands, D.: Very static enforcement of dynamic policies. In: Principles of Security and Trust. LNCS, vol. 9036, pp. 32–52 (2015). https://doi.org/10.1007/978-3-662-46666-7_3
11. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20 (1982). https://doi.org/10.1109/SP.1982.10014
12. Guarnieri, M., Balliu, M., Schoepe, D., Basin, D., Sabelfeld, A.: Information-flow control for database-backed applications. In: IEEE European Symposium on Security and Privacy. pp. 79–95 (2019). https://doi.org/10.1109/EuroSP.2019.00016
13. Hicks, M., Tse, S., Hicks, B., Zdancewic, S.: Dynamic updating of information-flow policies. In: Foundations of Computer Security. pp. 7–18 (2005)
14. Kavvos, G.A.: Modalities, cohesion, and information flow. Proceedings of the ACM on Programming Languages **3**(POPL) (2019). https://doi.org/10.1145/3290333
15. Lawvere, F.W., Menni, M.: Internal choice holds in the discrete part of any cohesive topos satisfying stable connected codiscreteness. Theory and Applications of Categories **30**(26), 909–932 (2015), https://www.tac.mta.ca/tac/volumes/30/26/30-26abs.html
16. Li, P., Zhang, D.: Towards a general-purpose dynamic information flow policy. In: IEEE Computer Security Foundations Symposium. pp. 260–275 (2022). https://doi.org/10.1109/CSF54842.2022.9919639
17. Mitchell, J.C.: Foundations for Programming Languages. MIT Press (1996)
18. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
19. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
20. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Software Security—Theories and Systems. LNCS, vol. 3233, pp. 174–191. Springer (2004). https://doi.org/10.1007/978-3-540-37621-7_9
21. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. Journal of Computer Security **17**(5), 517–548 (2009)
22. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: Proceedings of the 2011 ACM SIGPLAN Haskell Symposium. pp. 95–106 (2011). https://doi.org/10.1145/2034675.2034688
23. Sterling, J., Harper, R.: Sheaf semantics of termination-insensitive noninterference. In: 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 228, pp. 5:1–5:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.5
24. Swamy, N., Hicks, M., Tse, S., Zdancewic, S.: Managing policy updates in security-typed languages. In: IEEE Computer Security Foundations Workshop. pp. 202–216 (2006). https://doi.org/10.1109/CSFW.2006.17
25. The mathlib Community: The Lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 367–381. CPP 2020, Association for Computing Machinery (2020). https://doi.org/10.1145/3372885.3373824

Appendix for A Categorical Model of Dynamic Information-Flow Policies

Antonio Zegarelli, Niki Vazou, and Marco Guarnieri

IMDEA Software Institute
antonio.zegarelli@imdea.org, niki.vazou@imdea.org,
marco.guarnieri@imdea.org

This appendix contains DDCC calculations and proofs (Section A), details of conditionals, functions, and recursion (Section B), and the Lean mechanization and Mathlib support (Section C).

A DDCC: Supporting Calculations and Proofs

A.1 Complete first-order interpretation

This subsection gives the complete release definitions from § 4.2 and every interpretation used in § 4.4, including all constructors of Figure 3. Fix the label set L, the policy preorders $\sqsubseteq_p$, and the named policy changes with their independently specified release maps. All term-indexed definitions refer to a fixed typing derivation; the notation M suppresses that derivation.

Type carriers and observer relations. Write Ty for the first-order types and $\mathbb{B}$ for $\{\mathsf{true}, \mathsf{false}\}$. The carrier interpretation $|-| : \mathrm{Ty} \to \mathrm{Obj}(\mathbf{Set})$ is

$$\begin{aligned} |\mathsf{Bool}| &= \mathbb{B}, & |\mathsf{Unit}| &= \{\star\}, \\ |A \times B| &= |A| \times |B|, & |T_\ell\ A| &= \{[a]_\ell \mid a \in |A|\}. \end{aligned}$$

The protected carrier is canonically isomorphic to $|A|$ via $[a]_\ell \mapsto a$. We use this identification for the redaction modality; $[a]_\ell.\mathsf{value} = a$ extracts the underlying value. Carriers are independent of the policy. For each p, $\ell_o \in L$, and A, the relation has signature

$$R^A_{p,\ell_o} : |A| \times |A| \longrightarrow \mathsf{Prop}$$

and is defined by

$$\begin{aligned} b\, R^{\mathsf{Bool}}_{p,\ell_o}\, b' &\iff b = b', \\ u\, R^{\mathsf{Unit}}_{p,\ell_o}\, u' &\iff \mathsf{True}, \\ (a,b)\, R^{A\times B}_{p,\ell_o}\, (a',b') &\iff a\, R^A_{p,\ell_o}\, a' \ \wedge\ b\, R^B_{p,\ell_o}\, b', \\ u\, R^{T_\ell\ A}_{p,\ell_o}\, v &\iff \big(\ell \sqsubseteq_p \ell_o \Rightarrow u.\mathsf{value}\, R^A_{p,\ell_o}\, v.\mathsf{value}\big). \end{aligned}$$

Thus the classified type interpretation has signature

$$\begin{aligned}[\![-]\!]_p &: \mathrm{Ty} \to \mathrm{Obj}(\mathbf{CSet}_L),\\ [\![A]\!]_p &= \big(|A|, (R^A_{p,\ell_o})_{\ell_o \in L}\big).\end{aligned}$$

The same objects serve in $\mathbf{DSet}_L$. A static classified-set morphism f embeds into $\mathbf{DSet}_L$ as $(f, \top)$, where $\top$ is the total release bound at every observer.

Redaction, return, and protected bind. The hidden-observer mask is

$$\downarrow^p_\ell = \{\ell_o \in L \mid \ell \not\sqsubseteq_p \ell_o\} \in \mathcal{P}(L).$$

For any $\pi \subseteq L$, redaction is the functor $\Diamond_\pi : \mathbf{CSet}_L \to \mathbf{CSet}_L$ given by

$$\begin{aligned}|\Diamond_\pi X| &= |X|,\\ R^{\Diamond_\pi X}_{\ell_o} &= \begin{cases} |X| \times |X| & \ell_o \in \pi,\\ R^X_{\ell_o} & \ell_o \notin \pi,\end{cases}\\ \Diamond_\pi f &= f.\end{aligned}$$

In particular, $[\![T_\ell\ A]\!]_p = \Diamond_{\downarrow^p_\ell}[\![A]\!]_p$. For the remainder of this paragraph, abbreviate $\Diamond = \Diamond_{\downarrow^p_\ell}$. Its unit, multiplication, and strength have signatures

$$\begin{aligned}\eta^{p,\ell}_X &: X \to \Diamond X,\\ \mu^{p,\ell}_X &: \Diamond\Diamond X \to \Diamond X,\\ \mathsf{st}^{p,\ell}_{X,Y} &: X \times \Diamond Y \to \Diamond(X \times Y).\end{aligned}$$

All three are identity functions on their canonically identified carriers. For $p \vdash \ell \preceq B$, the relations of $Y = [\![B]\!]_p$ are total at every observer in $\downarrow^p_\ell$. Hence $\Diamond Y = Y$, and

$$\alpha^{p,\ell}_Y : \Diamond Y \to Y, \qquad \alpha^{p,\ell}_Y = (\eta^{p,\ell}_Y)^{-1}$$

is also the identity on the carrier. Given $G = [\![\Gamma]\!]_p$, $X = [\![A]\!]_p$, and a continuation $n : G \times X \to Y$, define

$$\overline{n} \doteq \alpha^{p,\ell}_Y \circ \Diamond(n) \circ \mathsf{st}^{p,\ell}_{G,X} : G \times \Diamond X \to Y.$$

For $m : G \to \Diamond X$, its protected bind is

$$\mathsf{bind}_{p,\ell}(m, n) \doteq \overline{n} \circ \langle \mathrm{id}_G, m\rangle : G \to Y.$$

This construction requires the protection premise for B. On carriers, $\overline{n}(\rho, u) = n(\rho, u.\mathsf{value})$.

Contexts, environments, and substitutions. Write Ctx for finite typed contexts. The context interpretation and its carrier are

$$\begin{aligned}
[\![-]\!]_p &: \mathrm{Ctx} \to \mathrm{Obj}(\mathbf{CSet}_L),\\
[\![\varnothing]\!]_p &= 1,\\
[\![\Gamma, x : A]\!]_p &= [\![\Gamma]\!]_p \times [\![A]\!]_p,\\
|\Gamma| &\doteq |[\![\Gamma]\!]_p|,\\
|\varnothing| &= \{\star\}.
\end{aligned}$$

Environments are policy-independent assignments of values to variables. Their relations have signature

$$R^{\Gamma}_{p,\ell_o} : |\Gamma| \times |\Gamma| \to \mathsf{Prop}$$

and satisfy

$$\begin{aligned}
\star \, R^{\varnothing}_{p,\ell_o} \, \star &\iff \mathsf{True},\\
(\rho, a)\, R^{\Gamma, x:A}_{p,\ell_o}\, (\rho', a') &\iff \rho\, R^{\Gamma}_{p,\ell_o}\, \rho' \;\wedge\; a\, R^{A}_{p,\ell_o}\, a'.
\end{aligned}$$

Lookup and extension have signatures

$$\begin{aligned}
&\mathsf{get}_x : |\Gamma| \to |A| && (x : A \in \Gamma),\\
&\mathsf{ext}_x : |\Gamma| \times |A| \to |\Gamma, x : A|.
\end{aligned}$$

They are given by $\mathsf{get}_x(\rho) = \rho(x)$ and $\mathsf{ext}_x(\rho, a) = \rho[x \mapsto a]$; the latter maps x to a and retains the previous assignments for the other variables. A canonical closing substitution σ denotes the environment $\rho_\sigma(x) = [\![\sigma(x)]\!]$.

Every static term constructor. For a static derivation $\Gamma \vdash_p S : A$, write $s_{p,S} = [\![S]\!]_p : |\Gamma| \to |A|$ for its carrier denotation. The complete recursion for the static syntax of Figure 3 is

$$\begin{aligned}
s_{p,x}(\rho) &= \rho(x),\\
s_{p,\mathsf{true}}(\rho) &= \mathsf{true},\\
s_{p,\mathsf{false}}(\rho) &= \mathsf{false},\\
s_{p,()}(\rho) &= \star,\\
s_{p,\langle S,T\rangle}(\rho) &= \big(s_{p,S}(\rho), s_{p,T}(\rho)\big),\\
s_{p,\mathsf{fst}\,S}(\rho) &= \pi_1(s_{p,S}(\rho)),\\
s_{p,\mathsf{snd}\,S}(\rho) &= \pi_2(s_{p,S}(\rho)),\\
s_{p,[S]_\ell}(\rho) &= \eta^{p,\ell}_{[\![A]\!]_p}(s_{p,S}(\rho)),\\
s_{p,\mathsf{let}\ x \leftarrow S\ \mathsf{in}\ T}(\rho) &= s_{p,T}\big(\rho[x \mapsto s_{p,S}(\rho).\mathsf{value}]\big).
\end{aligned}$$

In the return clause $S : A$; in bind $S : T_\ell\ A$, $\Gamma, x : A \vdash_p T : B$, and $p \vdash \ell \preceq B$. Categorically, variables are product projections, constants are constant maps, pairs are pairings, and fst and snd compose with π_1 and π_2. Return composes with η; bind uses $\mathsf{bind}_{p,\ell}$ above. These maps are morphisms of $\mathbf{CSet}_L$.

Every dynamic term constructor. For a fixed derivation D of $\Gamma; p_i \vdash M : A; p_o$, the underlying denotation has signature

$$d_D \doteq [\![M]\!]_{p_i,p_o} : |\Gamma| \longrightarrow |A|.$$

The three dynamic typing rules give the following complete recursion:

$$\begin{aligned} [\![S]\!]_{p_i,p_i}(\rho) &\doteq s_{p_i,S}(\rho) && \text{for a STATIC embedding,} \\ [\![\mathsf{policy}\ \delta\ \mathsf{in}\ M]\!]_{p_i,p_o}(\rho) &\doteq [\![M]\!]_{q,p_o}(\rho), \\ [\![\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2]\!]_{p_i,p_o}(\rho) & \\ &\doteq [\![M_2]\!]_{q,p_o}(\widehat{d_1}(\rho)). \end{aligned}$$

Here the static premise is $\Gamma \vdash_{p_i} S : A$. For $\Gamma; p_i \vdash M_1 : A; q$, context extension is

$$\widehat{d_1} : |\Gamma| \to |\Gamma, x : A|, \qquad \widehat{d_1}(\rho) \doteq \mathsf{ext}_x(\rho, [\![M_1]\!]_{p_i,q}(\rho)).$$

The change clause uses $\delta \triangleq p_i \rightsquigarrow q$ and interprets the body from q to p_o. Its identity carrier transport is well-defined because first-order environment carriers do not depend on the policy. No computation clause receives a release value as an argument.

Single authorized releases. For each $\delta \triangleq p_i \rightsquigarrow q$ and context Γ, the primitive release data consist of a specified set $V_{\delta,\Gamma}$ and a function

$$\mathsf{release}_{\delta,\Gamma} : |\Gamma| \to V_{\delta,\Gamma}.$$

The policy author supplies these data as a fixed authorization specification, independently of term evaluation and before proving security. The security predicate checks the named change against this specification; its endpoints alone do not determine its release.

Accumulated release codomains and maps. The codomain $V_D = V_M$ and map $r_D = \mathsf{release}_M : |\Gamma| \to V_M$ are defined on the same fixed derivation as d_D. For a STATIC embedding S, their full recursion is

$$\begin{aligned} V_S &= \{\star\}, \\ V_{\mathsf{policy}\ \delta\ \mathsf{in}\ M} &= V_{\delta,\Gamma} \times V_M, \\ V_{\mathsf{let}\ x=M_1\ \mathsf{in}\ M_2} &= V_{M_1} \times V_{M_2}, \\ \mathsf{release}_S(\rho) &= \star, \\ \mathsf{release}_{\mathsf{policy}\ \delta\ \mathsf{in}\ M}(\rho) &= \big(\mathsf{release}_{\delta,\Gamma}(\rho), \mathsf{release}_M(\rho)\big), \\ \mathsf{release}_{\mathsf{let}\ x=M_1\ \mathsf{in}\ M_2}(\rho) & \\ &= \big(\mathsf{release}_{M_1}(\rho), \mathsf{release}_{M_2}(\widehat{d_1}(\rho))\big). \end{aligned}$$

Here the release of M_2 has domain $|\Gamma, x : A|$; $\widehat{d_1}$ supplies its input. Thus release accumulation uses the computed value of M_1, while computation does not use release values. Every static constructor, including return and protected bind, is covered by the constant-release embedding clause.

Refined contexts. For a release $r : |\Gamma| \to V$, the environment refinement is

$$\rho \, R^{\Gamma,r}_{p,\ell_o} \, \rho' \quad \dot{\iff} \quad \rho \, R^{\Gamma}_{p,\ell_o} \, \rho' \ \wedge \ r(\rho) = r(\rho').$$

The refined context retains the carrier and packages these relations:

$$[\![\Gamma]\!]_p[r] \doteq \big(|\Gamma|, (R^{\Gamma,r}_{p,\ell_o})_{\ell_o \in L}\big).$$

Full interpretation of judgments. Let $\mathrm{Deriv}(\Gamma, p_i, A, p_o)$ denote the typing derivations of Figure 4, including their security proofs. For such a derivation D, let $d_D = [\![M]\!]_{p_i,p_o}$ and $r_D = \mathsf{release}_M$ be the underlying computation and accumulated release defined above. Soundness constructs a proof

$$\begin{aligned} h_D : \forall \ell_o \in L, \ \rho, \rho' \in |\Gamma|. \\ \rho \, R^{\Gamma,r_D}_{p_i,\ell_o} \, \rho' \implies d_D(\rho) \, R^{A}_{p_o,\ell_o} \, d_D(\rho'). \end{aligned}$$

Write $\mathcal{I}(D) \doteq [\![\Gamma; p_i \vdash M : A; p_o]\!]$ for the judgment denotation used in § 4.4. Define the release bound $K_D \doteq \mathrm{Ker}(r_D)$, meaning $K_{D,\ell_o}(\rho, \rho') \dot{\iff} r_D(\rho) = r_D(\rho')$ at every observer ℓ_o. With primitive release data fixed, the full interpretation has signature

$$\begin{aligned} \mathcal{I} : \prod_{D : \mathrm{Deriv}(\Gamma, p_i, A, p_o)} \mathbf{DSet}_L\big([\![\Gamma]\!]_{p_i}, [\![A]\!]_{p_o}\big), \\ \mathcal{I}(D) \doteq (d_D, K_D). \end{aligned}$$

Here $\mathbf{DSet}_L(X, Y)$ denotes the set of release-bounded arrows from X to Y. The source is the unrefined input context; the arrow carries the bound K_D, which depends on D through the accumulated release r_D. The proof h_D establishes the security condition for this arrow. In Lean, d_D is `Term.denote`, and h_D is the proof of `Classified.Preserves` supplied by `Term.classified_preserves` using the local security proofs in `CertifiedTerm.secure`. By Theorem 4, the same preservation proof equivalently presents d_D as a morphism $[\![\Gamma]\!]_{p_i}[r_D] \to [\![A]\!]_{p_o}$ in $\mathbf{CSet}_L$.

For sequential composition the stronger, retained-context map is used, displayed here in its equivalent refined-source $\mathbf{CSet}_L$ form:

$$\begin{aligned} \widehat{d}_D : [\![\Gamma]\!]_{p_i}[r_D] \to [\![\Gamma]\!]_{p_o} \times [\![A]\!]_{p_o}, \\ \widehat{d}_D(\rho) = (\rho, d_D(\rho)). \end{aligned}$$

The proof of Theorem 8 establishes preservation for both maps from the local security certificates. The typing judgment itself has no release argument; its interpretation computes the accumulated release from the fixed primitive authorizations and the chosen derivation.

A.2 Full elaboration of the patient–hospital–doctor example

Use the policies and release maps of § 4.2. For this calculation, rename the variables p, e, h of table 1 to pat, b, hos and write

$$\begin{aligned}
\Gamma &\doteq pat : T_P\ \mathsf{Bool},\ b : T_E\ \mathsf{Bool},\\
\Gamma' &\doteq \Gamma, hos : T_H\ \mathsf{Bool},\\
C &\doteq \mathsf{let}\ x \leftarrow pat\ \mathsf{in}\ [x]_H,\\
N &\doteq \mathsf{let}\ y \leftarrow hos\ \mathsf{in}\ [y]_D,\\
F &\doteq \mathsf{let}\ hos = C\ \mathsf{in\ policy\ discharge\ in}\ (\mathsf{policy\ transfer\ in}\ N).
\end{aligned}$$

Thus F is the program M of the table, with renamed variables. Write $G_p = [\![\Gamma]\!]_p$, $G'_p = [\![\Gamma']\!]_p$, $\rho_{h,b_0} = \{pat \mapsto [h]_P, b \mapsto [b_0]_E\}$, and $\rho^+_{h,b_0,q} = \rho_{h,b_0}[hos \mapsto [q]_H]$.

Typing every subterm. The copy uses the variable and protection rules to obtain

$$\Gamma \vdash_{p_c} pat : T_P\ \mathsf{Bool}, \qquad \Gamma, x : \mathsf{Bool} \vdash_{p_c} [x]_H : T_H\ \mathsf{Bool},$$
$$P \sqsubseteq_{p_c} H \quad \Longrightarrow \quad p_c \vdash P \preceq T_H\ \mathsf{Bool}.$$

Protected bind and Static therefore give $\Gamma; p_c \vdash C : T_H\ \mathsf{Bool}; p_c$. Similarly, in the final context,

$$\Gamma' \vdash_{p_s} hos : T_H\ \mathsf{Bool}, \qquad \Gamma', y : \mathsf{Bool} \vdash_{p_s} [y]_D : T_D\ \mathsf{Bool},$$
$$H \sqsubseteq_{p_s} D \quad \Longrightarrow \quad p_s \vdash H \preceq T_D\ \mathsf{Bool}.$$

Hence $\Gamma'; p_s \vdash N : T_D\ \mathsf{Bool}; p_s$. With the certificates below, the two applications of Change yield

$$\begin{aligned}
&\Gamma'; p \vdash \mathsf{policy\ transfer\ in}\ N : T_D\ \mathsf{Bool}; p_s,\\
&\Gamma'; p_c \vdash \mathsf{policy\ discharge\ in}\ \big(\mathsf{policy\ transfer\ in}\ N\big)\\
&\qquad\qquad : T_D\ \mathsf{Bool}; p_s.
\end{aligned}$$

The final application of Let binds the result of C as hos and gives $\Gamma; p_c \vdash F : T_D\ \mathsf{Bool}; p_s$.

Observer relations for all retained variables. Consider two arbitrary environments of Γ', $\rho^+_{h,b_0,q}$ and $\rho^+_{h',b'_0,q'}$. The entries below are exactly the conditions under which those environments are related at the indicated policy and observer:

observer	p_c	p	p_s
P	$h = h'$	$h = h'$	$h = h'$
H	$h = h' \ \wedge\ q = q'$	$q = q'$	$q = q'$
D	True	True	$q = q'$
E	$b_0 = b'_0$	$b_0 = b'_0$	$b_0 = b'_0$

This table follows by applying the type relation to each retained variable. In particular, the only relation strengthened by transfer is the doctor's: it changes from total to equality of q.

The two security certificates. For discharge, the p_c column implies the p column at every observer. The constant release has total kernel, so this proves $\mathsf{Secure}(\Gamma', \mathsf{discharge})$ and the morphism

$$\mathrm{id} : G'_{p_c}[\mathsf{release}_{\mathsf{discharge},\Gamma'}] \longrightarrow G'_p.$$

For transfer, equal primitive releases mean $q = q'$. Intersecting the p column with that equality implies the p_s column at every observer. This proves $\mathsf{Secure}(\Gamma', \mathsf{transfer})$ and

$$\mathrm{id} : G'_p[\mathsf{release}_{\mathsf{transfer},\Gamma'}] \longrightarrow G'_{p_s}.$$

These certificates quantify over all h, b_0, q, including $q \neq h$. The equality $q = h$ is established by the copy in this workflow; it is not an assumption of the primitive transfer certificate.

Failure 1: a constant transfer release. With a constant transfer release, the two environments $\rho^{+}_{h,b_0,\mathsf{false}}$ and $\rho^{+}_{h,b_0,\mathsf{true}}$ are related at observer D under p and have equal releases, but are unrelated under p_s. This directly refutes the certificate for the unauthorized variant.

Failure 2: an unreleased retained input. Extend the initial context with an independent hospital-protected input:

$$\Gamma_k \doteq \Gamma, k : T_H\ \mathsf{Bool}, \qquad \Gamma'_k \doteq \Gamma_k, hos : T_H\ \mathsf{Bool}.$$

Use the same workflow code as F, now called F_k, and keep the transfer release equal to the payload of hos. After the copy, for fixed patient and external payloads h, b_0, the retained environments are

$$\tau_a \doteq \rho_{h,b_0}[k \mapsto [a]_H, hos \mapsto [h]_H] \qquad (a \in \mathbb{B}).$$

The doctor cannot distinguish k under p, and the release ignores it. After transfer, both hospital-protected variables become visible:

$$\mathsf{release}_{\mathsf{transfer},\Gamma'_k}(\tau_a) = h \qquad (a \in \mathbb{B}),$$
$$\tau_{\mathsf{false}}\, R^{\Gamma'_k}_{p,D}\, \tau_{\mathsf{true}}, \qquad \neg\big(\tau_{\mathsf{false}}\, R^{\Gamma'_k}_{p_s,D}\, \tau_{\mathsf{true}}\big).$$

Indeed, the doctor's relation after transfer requires equality of both the hos and k payloads. Thus $\mathsf{Secure}(\Gamma'_k, \mathsf{transfer})$ fails, despite equal releases. These witnesses arise from initial environments with the same patient and external values and different k values, so they are reachable after the copy and discharge.

The raw workflow still returns $[h]_D$ and has accumulated release $(\star, (\star, (h, \star)))$, independently of k. The failure concerns the retained context, even though the body only returns the authorized hospital copy. Lean defines this complete workflow as `workflowWithUnreleasedInput` and proves its result, accumulated release, operational evaluation, and rejection by the certificate condition$^{\mathsf{L}\exists\forall\mathsf{N}}$.

Computation and release, with their signatures. The underlying static maps and environment extension are

$$\begin{aligned} d_C &: |G_{p_c}| \to |[\![T_H\ \mathsf{Bool}]\!]_{p_c}|, \\ d_C(\rho_{h,b_0}) &= \eta^{p_c,H}_{[\![\mathsf{Bool}]\!]_{p_c}}(h), \\ \widehat{d}_C &: |G_{p_c}| \to |G'_{p_c}|, \\ \widehat{d}_C(\rho_{h,b_0}) &= \rho^{+}_{h,b_0,h}, \\ d_N &: |G'_{p_s}| \to |[\![T_D\ \mathsf{Bool}]\!]_{p_s}|, \\ d_N(\rho^{+}_{h,b_0,q}) &= \eta^{p_s,D}_{[\![\mathsf{Bool}]\!]_{p_s}}(q). \end{aligned}$$

The two identity carrier transports give $d_F = d_N \circ \mathrm{id} \circ \mathrm{id} \circ \widehat{d}_C$. As classified-set maps, those transports use different input and output relations and the release refinements specified by their certificates. The primitive release signatures are

$$\begin{aligned} \mathsf{release}_{\mathsf{discharge},\Gamma'} &: |G'_{p_c}| \to \{\star\}, \\ \mathsf{release}_{\mathsf{transfer},\Gamma'} &: |G'_{p}| \to \mathbb{B}. \end{aligned}$$

The static embeddings of C and N have singleton release codomains. The recursive release clauses give

$$\begin{aligned} \mathsf{release}_N(\rho^{+}_{h,b_0,q}) &= \star, \\ \mathsf{release}_{\mathsf{policy\ transfer\ in\ }N}(\rho^{+}_{h,b_0,q}) &= (q,\star), \\ \mathsf{release}_{\mathsf{policy\ discharge\ in\ (policy\ transfer\ in\ }N)}(\rho^{+}_{h,b_0,q}) &= (\star,(q,\star)), \\ \mathsf{release}_F(\rho_{h,b_0}) &= (\star,(\star,(h,\star))). \end{aligned}$$

Thus the primitive transfer kernel compares q and q', while the kernel pulled back through this workflow's copy compares h and h':

$$\rho_{h,b_0}\ \mathrm{Ker}(\mathsf{release}_F)\,\rho_{h',b'_0} \quad\Longleftrightarrow\quad h = h'.$$

The full interpretation is therefore the arrow

$$(d_F, \mathrm{Ker}(\mathsf{release}_F)) : G_{p_c} \longrightarrow [\![T_D\ \mathsf{Bool}]\!]_{p_s} \quad \text{in } \mathbf{DSet}_L.$$

At observer D, the initial context relation is total and the result relation is Boolean equality. Equal accumulated releases give equal doctor results for arbitrary unrelated bits b_0, b'_0, as mechanized in § 4.4.

A.3 Conditional release and operational semantics

For $N = \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2$, let $b = \mathsf{eval}(M,\rho)$ and $V_N = V_{N_1} + V_{N_2}$. Then

$$[\![N]\!](\rho) = \begin{cases} [\![N_1]\!](\rho) & b = \mathsf{true}, \\ [\![N_2]\!](\rho) & b = \mathsf{false}, \end{cases}$$

$$\mathsf{release}_N(\rho) = \begin{cases} \mathsf{inl}(\mathsf{release}_{N_1}(\rho)) & b = \mathsf{true}, \\ \mathsf{inr}(\mathsf{release}_{N_2}(\rho)) & b = \mathsf{false}. \end{cases}$$

The tag distinguishes the branches and only the chosen branch contributes authorization. It does not authorize disclosure of a secret guard: the static Boolean typing premise already ensures equal guards on related inputs. A Static embedding retains constant release; a If derivation uses the tagged release, as described in Section 5.1.

A closing substitution σ assigns a canonical value to each variable in Γ. Write $M[\sigma]$ for the expression evaluated under that assignment, and $M[\sigma] \Downarrow w$ for its big-step evaluation. These substitutions can equivalently be represented by canonical-value environments. Program equality means agreement of canonical results:

$$\vdash M[\sigma] = M'[\tau] : A \quad \Longleftrightarrow \quad \exists w.\ M[\sigma] \Downarrow w \ \wedge\ M'[\tau] \Downarrow w.$$

For a one-variable context $x : A$, the notation is $M[a_L/x]$. A change leaves the closing substitution unchanged and evaluates its body:

$$\frac{M[\sigma] \Downarrow w}{(\mathsf{policy}\ \delta\ \mathsf{in}\ M)[\sigma] \Downarrow w}.$$

Its typing derivation records the policy transition; evaluation computes the canonical result of the body. Static constructors use the first-order DCC evaluation rules. Sequential let-binding extends the canonical environment with the first term's result:

$$\frac{M[\sigma] \Downarrow v \qquad N[\sigma, x \mapsto v] \Downarrow w}{(\mathsf{let}\ x = M\ \mathsf{in}\ N)[\sigma] \Downarrow w}.$$

The conditional first evaluates its guard and then only the chosen branch:

$$\frac{M[\sigma] \Downarrow \mathsf{true} \qquad N_1[\sigma] \Downarrow w}{(\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)[\sigma] \Downarrow w}, \qquad \frac{M[\sigma] \Downarrow \mathsf{false} \qquad N_2[\sigma] \Downarrow w}{(\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)[\sigma] \Downarrow w}.$$

A.4 Soundness and adequacy proof details

Static preservation$^{\mathsf{L}\exists\forall\mathsf{N}}$. At a fixed policy, static terms map related environments to related results.

First, induction on protection proves that $p_i \vdash \ell \preceq B$ and $\ell \not\sqsubseteq_{p_i} \ell_o$ imply that $R^B_{p_i,\ell_o}$ is total. For P-Level, transitivity gives $\ell' \not\sqsubseteq_{p_i} \ell_o$ from $\ell \sqsubseteq_{p_i} \ell'$ and $\ell \not\sqsubseteq_{p_i} \ell_o$. Hence $T_{\ell'}\ A$ is hidden from ℓ_o. Unit, products, and nested protection follow directly from the definitions.

Now induct on the typing derivation. Variables, constants, pairs, and projections preserve the corresponding relations. Protected introduction compares related values when their label may flow to ℓ_o and has a total result relation otherwise. For protected elimination, if $\ell \sqsubseteq_{p_i} \ell_o$, related inputs have related values, so the induction hypothesis applies to the body in extended environments. Otherwise, the protection side condition makes the result relation total. For a static conditional, related Boolean guards are equal, so both executions select the same branch and its induction hypothesis applies. All premises use the same policy p_i.

Proof (Theorem 8). Let $\widehat{M}(\rho) = (\rho, [\![M]\!](\rho))$. The joint invariant says that $\widehat{M}$ is a morphism in $\mathbf{CSet}_L$

$$[\![\Gamma]\!]_{p_i}[\mathsf{release}_M] \longrightarrow [\![\Gamma]\!]_{p_o} \times [\![A]\!]_{p_o}.$$

We prove this invariant by induction, composing refined maps as follows.

A transformation f with authorized release r gives the arrow $(f, \mathrm{Ker}(r))$; its security condition is ordinary relation preservation by $f : X[r] \to Y$ (Theorem 4).

If $f : X[\boldsymbol{r}] \to Y$ and $g : Y[\boldsymbol{s}] \to Z$ are classified-set morphisms, their underlying composite is a morphism$^{\mathsf{L}\exists\forall\mathsf{N}}$

$$g \circ f : X[\langle \boldsymbol{r}, \boldsymbol{s} \circ f \rangle] \to Z,$$

where the composite release at ℓ is $x \mapsto (r_\ell(x), s_\ell(f(x)))$. Related inputs with equal first releases have related f-images. Equality of the second releases then lets us apply preservation by g.

Equal composite releases make f land in $Y[\boldsymbol{s}]$, where g applies. The first component of the paired release records what f may disclose; the second records what g may disclose after applying f.

For a pure term, the policy is unchanged and the static preservation argument above gives result preservation. For $\mathsf{policy}\ \delta\ \mathsf{in}\ M$, equality of accumulated releases gives equality of the change releases and of the body releases. The change certificate makes the retained environments related under the new policy; the induction hypothesis for M then gives both final-context and result preservation. For let-binding, the first induction hypothesis gives related original environments and related bound values at the intermediate policy. Hence the extended environments are related. Equality of the second release components allows the induction hypothesis for the body. Projecting its final context relation onto the original variables and retaining its result relation proves both claims. This is composition of refined maps with explicit context extension. For a general conditional, static preservation makes the guards equal on related environments. Equality of tagged releases gives equal releases in the selected branch. That branch's induction hypothesis proves both final-context and result preservation. No relation between the outputs of different branches is needed, since these inputs cannot select different branches$^{\mathsf{L}\exists\forall\mathsf{N}}$. Projecting to the result and applying Theorem 4 gives the claimed $\mathbf{DSet}_L$ arrow $([\![M]\!]_{p_i,p_o}, \mathrm{Ker}(\mathsf{release}_M))$ from $[\![\Gamma]\!]_{p_i}$ to $[\![A]\!]_{p_o}$.

Proof (Theorem 9). Induct on the term structure, evaluating static constructors in the usual way. Protected elimination evaluates its argument to a protected canonical value and evaluates its body with that value bound to the variable. A policy expression evaluates its body in the same environment; a sequential let evaluates its first term and then its body in the extended environment. A conditional evaluates its Boolean guard and the selected branch. Each case terminates by the induction hypotheses and agrees with its denotation. The evaluation rules are deterministic. Canonical-value denotations are injective by induction on types: boolean and unit cases are immediate, and product and protection cases reduce to their components. Thus equal denotations determine the same canonical result.

A.5 Category and redaction laws for $\mathbf{DSet}_L$

This subsection gives the proofs of Theorems 5 and 6 from Sections 3.2 and 3.3.

Category laws.

Proof (Theorem 5). We prove that 1) the identity arrows are valid, 2) composite arrows are valid, and 3) the category laws hold for both components of each arrow: its underlying function and release bound.

1) Identity arrows. For every object X, the proposed identity $1_X = (\mathrm{id}, \top)$ has an equivalence bound and preserves every security relation of X. Thus it is a valid arrow.

2) Composite arrows. For $(f, K) : X \to Y$ and $(g, H) : Y \to Z$, the bound $K \cap f^{-1}H$ is an equivalence relation at each observer, since inverse image and intersection preserve equivalence relations. Suppose $x\, R^X_\ell\, y$ and $(K \cap f^{-1}H)_\ell(x, y)$. Security of (f, K) gives $f(x)\, R^Y_\ell\, f(y)$; the remaining premise $H_\ell(f(x), f(y))$ lets security of (g, H) give $g(f(x))\, R^Z_\ell\, g(f(y))$. Thus every composable pair has a valid composite.

3) Category laws. The underlying functions satisfy the left and right identity laws and associativity of function composition. For bounds, the identity laws follow from $K \cap f^{-1}\top = K = \top \cap \mathrm{id}^{-1}K$. For a third arrow (h, J), inverse image preserves intersections and satisfies $f^{-1}(g^{-1}J) = (g \circ f)^{-1}J$, so $(K \cap f^{-1}H) \cap (g \circ f)^{-1}J = K \cap f^{-1}(H \cap g^{-1}J)$. Thus the bounds satisfy associativity as well.

Redaction modality.

Proof (Theorem 6). Functoriality. The relations of $\Diamond_\pi X$ are reflexive, so it is a classified set. If $(f, K) : X \to Y$ is secure, then its image $(f, K) : \Diamond_\pi X \to \Diamond_\pi Y$ is secure: at $\ell \in \pi$ the target relation is total and at $\ell \notin \pi$ the original security condition applies. Since $\Diamond_\pi$ retains both functions and bounds, $\Diamond_\pi(1_X) = 1_{\Diamond_\pi X}$ and $\Diamond_\pi(b \circ a) = \Diamond_\pi(b) \circ \Diamond_\pi(a)$.

Unit, multiplication, and naturality. The unit is secure because $R^X_\ell \subseteq R^{\Diamond_\pi X}_\ell$. Redacting twice gives the same relations as redacting once, so $\Diamond_\pi \Diamond_\pi X = \Diamond_\pi X$ and μ_X is secure as well. For any arrow $a = (f, K) : X \to Y$, naturality requires

$$\Diamond_\pi(a) \circ \eta_X = \eta_Y \circ a, \qquad \Diamond_\pi(a) \circ \mu_X = \mu_Y \circ \Diamond_\pi(\Diamond_\pi(a)).$$

Both sides of each equation have underlying function f and bound K, by the identity laws. Thus η and μ are natural transformations.

Monad laws. The required component equations are

$$\mu_X \circ \eta_{\Diamond_\pi X} = 1_{\Diamond_\pi X}, \quad \mu_X \circ \Diamond_\pi(\eta_X) = 1_{\Diamond_\pi X}, \quad \mu_X \circ \Diamond_\pi(\mu_X) = \mu_X \circ \mu_{\Diamond_\pi X}.$$

Every component and its image under $\Diamond_\pi$ has underlying function id and bound $\top$. Their composites also have function id and bound $\top$, so all equations hold.

Idempotence. Since $\Diamond_\pi \Diamond_\pi X = \Diamond_\pi X$, the arrow $(\mathrm{id}, \top) : \Diamond_\pi X \to \Diamond_\pi \Diamond_\pi X$ is an inverse to μ_X. Thus every component of the multiplication is an isomorphism, and the monad is idempotent.

A.6 Relation to the static interpretation.

The redaction set here differs from $\downarrow \ell$ in § 2.3: reflexivity puts ℓ in $\downarrow \ell$ and excludes it from $\downarrow_\ell^{p_i}$. The static proof uses *Hi* as a separating observer: the input is redacted there, whereas the output at *Lo* is visible when $\mathit{Hi} \not\sqsubseteq \mathit{Lo}$. Here the observer is the receiving label *Lo*: the same no-flow condition hides the input, and the output is visible by reflexivity. Both choices therefore give the same static NI conclusion, using different families of relations.

The new family is compatible with protection: transitivity gives $\ell \sqsubseteq_{p_i} \ell' \Rightarrow \downarrow_\ell^{p_i} \subseteq \downarrow_{\ell'}^{p_i}$. The static preservation argument in Appendix A.4 proves the typing interpretation for this family. Fixing a policy and taking constant releases recovers the static DCC NI statement on this first-order fragment, not a pointwise identification with $\Diamond_{\downarrow\ell}$. Independently, constant releases leave classified sets and their relation-preserving maps unchanged.

B Extensions

This appendix gives the detailed constructions for conditionals, functions, and recursion summarized in Section 5. Each instance records its assumptions and mechanized coverage.

B.1 Conditionals

We extend the term grammar of Figure 3 by

$$M, N ::= \cdots \mid \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2.$$

The following rules extend Figure 4:

$$\frac{\Gamma \vdash_p M : \mathsf{Bool} \qquad \Gamma \vdash_p N_1 : A \qquad \Gamma \vdash_p N_2 : A}{\Gamma \vdash_p \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : A}\ \textsc{If-Static}$$

$$\frac{\Gamma \vdash_{p_i} M : \mathsf{Bool} \qquad \Gamma; p_i \vdash N_1 : A; p_o \qquad \Gamma; p_i \vdash N_2 : A; p_o}{\Gamma; p_i \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : A; p_o}\ \textsc{If}$$

The static rule chooses between computations at one policy; the dynamic rule may choose different finite policy-change sequences.

A conditional must provide the same interface to its continuation whichever branch is selected. Thus If permits different intermediate policies but requires both branches to finish at p_o. Its static Boolean guard supplies public control: pure preservation gives agreement of branch selection on related inputs. A protected Boolean cannot serve directly as a guard. A static protected bind may branch on its value when the result protects that label, but its static body excludes policy changes. Supporting secret-controlled changes would require a rule tracking protected control effects. For a general conditional, the denotation evaluates the static guard and only the selected branch. Its release type is $V_{N_1} + V_{N_2}$; its release

injects the selected branch's release into this tagged sum. The untaken branch contributes no authorization. The tag does not authorize a secret guard: Boolean typing ensures equal guards on related inputs. A conditional embedded by Static retains constant release; tagged releases belong to If derivations. Appendix A.3 gives the equations.

A literature example: release one of two secrets. Sabelfeld and Myers consider a public choice between releasing h_1 and releasing h_2, and explain that their original delimited-release condition cannot express the requirement that only the chosen secret be released [22, § 3.4]. For a Boolean instance, prepare $q_1 : T_{S_1}\ \mathsf{Bool}$ and $q_2 : T_{S_2}\ \mathsf{Bool}$, with distinct summary labels, and a public guard c. The selected arm opens only $S_i \sqsubseteq \ell_o$, copies q_i to $T_{\ell_o}\ \mathsf{Bool}$, and revokes that edge. Both arms return to the same closed policy:

$$P_i = \mathsf{let}\ y = \mathsf{policy}\ \mathsf{open}_i\ \mathsf{in}\ (\mathsf{let}\ z \leftarrow q_i\ \mathsf{in}\ [z]_{\ell_o})\ \mathsf{in}\ \mathsf{policy}\ \mathsf{close}_i\ \mathsf{in}\ y,$$
$$C = \mathsf{if}\ c\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2.$$

Here $\mathsf{open}_i \triangleq p_c \rightsquigarrow p_i$ adds only $S_i \sqsubseteq \ell_o$, and $\mathsf{close}_i \triangleq p_i \rightsquigarrow p_c$ revokes it. The primitive release must independently authorize q_i in the context where P_i opens its edge. The whole-context certificate covers every retained value at S_i; it must not make S_{3-i} newly visible. Any additional retained copies would also need to be covered by the certificate. The accumulated release is $\mathsf{inl}(\mathsf{release}_{P_1}(\rho))$ or $\mathsf{inr}(\mathsf{release}_{P_2}(\rho))$ according to c. Thus the untaken arm contributes no release, and equal public guards and equal selected releases give equal results. This is a single-execution bound; repeated calls with different guards can cumulatively reveal both secrets. The Boolean instance is now mechanized in `Examples/Conditionals.lean`: its intrinsically typed DDCC term carries the opening and closing certificates for the full retained context. Lean checks evaluation to $[h_1]_{\ell_o}$ when c is true and $[h_2]_{\ell_o}$ otherwise, the selected-release kernel, and operational $\mathrm{NI}^{\mathsf{L}\exists\forall\mathsf{N}}$.

Soundness and adequacy. Theorems 8 and 9 extend to the additional rules above. The shared Lean implementation checks this larger terminating first-order language. Pure preservation makes related guards $\mathrm{equal}^{\mathsf{L}\exists\forall\mathsf{N}}$; equal tagged releases give equal releases for that selected arm. Its induction hypothesis supplies the context-and-result invariant. Evaluation runs only that arm and therefore preserves the terminating first-order adequacy argument. These cases are checked in the shared first-order implementation and detailed in Appendix A.4.

B.2 Functions and recursion

Functions, lifted types, and policy changes We extend the static fragment of DDCC with function and lifted types, following DCC [1, §3]. The additional syntax is

$$A, B ::= \cdots \mid A \to B \mid A_\perp,$$
$$M, N ::= \cdots \mid \lambda x : A.M \mid M\,N \mid \mathsf{lift}\,M$$
$$\mid \mathsf{seq}\ x \leftarrow M\ \mathsf{in}\ N \mid \mu f : A.M.$$

All added term premises use the static judgment $\Gamma \vdash_p M : A$ of Figure 4, at a fixed policy p. The predicate $\mathsf{Pointed}$ is generated by $\mathsf{Pointed}(A_\perp)$, products of two pointed types, and $A \to B$ and $T_\ell\ B$ when B is pointed. In particular, Bool is not pointed; recursion at a Boolean result uses $\mathsf{Bool}_\perp$.

$$\frac{\Gamma, x : A \vdash_p M : B}{\Gamma \vdash_p \lambda x : A.M : A \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash_p M : A \to B \qquad \Gamma \vdash_p N : A}{\Gamma \vdash_p M\,N : B}\ \textsc{App}$$

$$\frac{p \vdash \ell \preceq B}{p \vdash \ell \preceq A \to B}\ \textsc{P-Arrow} \qquad \frac{\Gamma \vdash_p M : A}{\Gamma \vdash_p \mathsf{lift}\,M : A_\perp}\ \textsc{Lift}$$

$$\frac{\Gamma, f : A \vdash_p M : A \qquad \mathsf{Pointed}(A)}{\Gamma \vdash_p \mu f : A.M : A}\ \textsc{Rec}$$

$$\frac{\Gamma \vdash_p M : A_\perp \qquad \Gamma, x : A \vdash_p N : B \qquad \mathsf{Pointed}(B)}{\Gamma \vdash_p \mathsf{seq}\ x \leftarrow M\ \mathsf{in}\ N : B}\ \textsc{Seq}$$

The protection rule constrains the result: applying a function may not reveal the protected input through an unprotected codomain.

At a fixed policy, interpret DCC in the continuous relational domain model of [1, §§3–4]. A carrier is an ω-complete partial order, with a reflexive relation at each observer. Relations are *admissible*: if corresponding members of two increasing chains are related, their suprema are related. Boolean and unit carriers are discrete; products and protection use the componentwise and hidden-value relations. The arrow carrier contains continuous maps preserving every observer relation, with pointwise order. Its relation is

$$f\ R^{A \to B}_{p,\ell_o}\ g \quad \Longleftrightarrow \quad \forall a_L, a_R.\ a_L\ R^{A}_{p,\ell_o}\ a_R \Longrightarrow f(a_L)\ R^{B}_{p,\ell_o}\ g(a_R).$$

This is a carrier of admissible maps, rather than all set functions: reflexivity follows from the carrier restriction. Suprema remain in the carrier by admissibility of the result relations. Application preserves relations by the displayed implication; abstraction uses preservation of its body under related environments and related arguments. Lean checks these arguments and the admissible arrow object$^{\mathsf{L}\exists\forall\mathsf{N}}$. The same codomain argument extends the protection lemma to arrows.

For comparison, the constant function $\lambda x : T_{Hi}\ \mathsf{Bool}.[\mathsf{true}]_{Lo}$ (from [1, §2.1]) is represented by `Examples.Functions.constantTrue`. Its interpretation returns the protected Boolean true for every argument$^{\mathsf{L}\exists\forall\mathsf{N}}$.

Transporting closures across policies. The arrow carrier can vary with the policy. For example, when $\ell_s \sqsubseteq_p \ell_o$, the closure $\lambda x : T_{\ell_s}\ \mathsf{Bool.let}\ z \leftarrow x\ \mathsf{in}\ [z]_{\ell_o}$ is admissible. After revoking that flow, its unchanged behavior distinguishes related hidden inputs at the visible result, and is no longer admissible. An identity map on arbitrary closures therefore does not give a policy change.

Use the general model's policy-indexed carriers. For each retained context, require a total continuous transport $t_{\delta,\Gamma} : D_{p_i}\Gamma \to D_{p_o}\Gamma$, together with its independently authorized release $\mathsf{release}_{\delta,\Gamma}$. Its release-bounded certificate is

$$R^{\Gamma}_{p_i,\ell_o} \cap \mathrm{Ker}(\mathsf{release}_{\delta,\Gamma}) \;\subseteq\; (t_{\delta,\Gamma} \times t_{\delta,\Gamma})^{-1} R^{\Gamma}_{p_o,\ell_o}. \tag{1}$$

Totality must include membership in the target arrow carrier, and hence captured environments and behavior on all admissible arguments. The release bound does not waive this membership requirement. A workflow may drop an invalid closure before changing policy, or provide an explicit valid replacement; it cannot merely assert that all closures survive revocation. For the checked first-order instance $t_{\delta,\Gamma} = \mathsf{id}$ and Eq. (1) reduces to Definition 6.

In the extension, change evaluates its body on $t_{\delta,\Gamma}\rho$. Its release is the pair

$$(\mathsf{release}_{\delta,\Gamma}\rho, \mathsf{release}_M(t_{\delta,\Gamma}\rho)).$$

Let-binding transports its retained environment as well as the bound value. Thus the context-and-result invariant compares *transported* environments $\tau_M\rho$ and $\tau_M\rho'$, not the original environments at the new policy. The Lean interpreter `DynamicTerm.run` in `RecursiveDCC/HigherOrder.lean` implements these transports over supplied policy-dependent carriers; its soundness proof establishes this invariant. `ConcreteTransports.lean` supplies a certified family: identity changes retain the context. For revocation and publication, ground data is retained componentwise, lifting preserves the outer bottom, and functions are replaced by canonical constant closures in the target policy's carrier. Revocation requires a shrinking flow relation and has constant release; publication releases the entire transported context$^{\mathsf{L}\exists\forall\mathsf{N}}$. No policy change occurs inside a lambda body or an application premise. Allowing such bodies would require latent policy and release effects.

Recursion and termination

The DCC lifting discipline. The constructs lift and seq follow DCC's recursion discipline [1, §3]: $A_\perp$ adds a fresh bottom for divergence; $\mathsf{lift}\, M$ injects an A-computation into this lifted type. Strict sequencing forces the outer lift before binding its value. Thus $\mathsf{seq}\ x \leftarrow \mathsf{lift}\, M\ \mathsf{in}\ N$ reduces to $N[M/x]$, whereas a divergent scrutinee makes the sequence diverge. Its result must have a bottom, expressed by $\mathsf{Pointed}(B)$. This is distinct from protected bind, which eliminates $T_\ell\ A$ under a protection condition, and from the outer policy-sequencing let. Functions alone need neither lifting construct; general recursion uses pointed types.

A discrete Boolean carrier has no least element; its lifting $\mathsf{Bool}_\perp$ supplies one. Lifting also distinguishes an outer failure to return from a returned value whose evaluation may itself diverge. The strict sequencing operation eliminates that outer distinction: its body runs only once the scrutinee exposes a lift. Ordinary let and protected bind therefore do not replace it in this call-by-name discipline.

Lean checks these constructors and their side conditions in `RecursiveDCC.Term`. The recursive body is static at p. Recursion is unrestricted unfolding at a pointed type, not bounded iteration. It does not repeat policy changes; those still occupy the finite outer sequence and conditional branches.

The lifted carrier is $D_p A$ with a fresh least element $\bot$ and elements $\mathsf{up}(a)$, ordered by the original order above $\bot$. Use the *strong* DCC relation:

$$\bot \, R^{A_\bot}_{p,\ell_o} \, \bot, \qquad \mathsf{up}(a_L) \, R^{A_\bot}_{p,\ell_o} \, \mathsf{up}(a_R) \iff a_L \, R^{A}_{p,\ell_o} \, a_R, \qquad \neg(\bot \, R^{A_\bot}_{p,\ell_o} \, \mathsf{up}(a_R)).$$

These clauses are checked separately from the domain order$^{\mathsf{L}\exists\forall\mathsf{N}}$. There is *no* rule inferring $p \vdash \ell \preceq A_\bot$ from $p \vdash \ell \preceq A$. Hiding a value inside a lift does not hide the outer termination distinction. For example, when $Hi \not\sqsubseteq_p Lo$, binding a $T_{Hi}\ \mathsf{Bool}$ and using its value to choose between divergence and a returned value at $(T_{Hi}\ \mathsf{Bool})_\bot$ fails the protection side condition. This is the original discipline, rather than the weaker lifting relation of vDCC [1, §5].

For pointed B, sequencing returns $\bot_B$ on $\bot$, and substitutes the value on $\mathsf{up}(a)$. The corresponding relation argument uses $\bot_B \, R^B_{p,\ell_o} \, \bot_B$ and preservation by the body on related values$^{\mathsf{L}\exists\forall\mathsf{N}}$. Pointedness supplies a least element related to itself: this follows for lifting and propagates through product, protection, and arrow codomains.

For $F_\rho(a) = [\![M]\!]_p(\rho, a)$, define

$$[\![\mu f : A.M]\!]_p \rho = \mathsf{lfp}(F_\rho) = \bigsqcup_{n<\omega} F^n_\rho(\bot_A).$$

Continuity gives the unfolding equation and leastness. If ρ and ρ' are related, preservation of the body relates every corresponding finite approximation, beginning with the two bottoms. Admissibility relates their suprema. Lean proves this argument for two possibly different domains and an admissible cross-relation, together with unfolding and leastness$^{\mathsf{L}\exists\forall\mathsf{N}}$. The full fixed-policy fundamental theorem is checked for every static constructor, including lambda, application, lift, seq, and unrestricted least fixed points$^{\mathsf{L}\exists\forall\mathsf{N}}$. The Lean structures `Model` and `StaticSemantics` specify the assumed carriers, relation clauses, continuous recursion functionals, and denotation equations. They do not assume preservation by terms: the theorem derives it by induction on syntax. `ConcreteModel.lean` and `ConcreteSemantics.lean` construct inhabitants of both structures, including continuous function objects and strong lifting$^{\mathsf{L}\exists\forall\mathsf{N}}$. The same clauses yield unrestricted unfolding and the syntactic loop's bottom denotation$^{\mathsf{L}\exists\forall\mathsf{N}}$.

Operational meaning and termination. Follow DCC's call-by-name substitution rules for the static fragment:

$$\begin{aligned} (\lambda x : A.M)\, N &\longrightarrow M[N/x], \\ \mu f : A.M &\longrightarrow M[\mu f : A.M/f], \\ \mathsf{seq}\ x \leftarrow \mathsf{lift}\, M\ \mathsf{in}\ N &\longrightarrow N[M/x]. \end{aligned}$$

Static protected bind also substitutes its suspended value. The concrete artifact implements call-by-name evaluation with syntactic closures and environments:

application extends the captured environment with a suspended argument, and a fixed point unfolds by binding its own suspension. Suspensions retain their creation-policy interpretation. Strict sequencing forces the outer lift; pairs, protected values, and lifted payloads remain suspended until eliminated. The machine has no execution fuel. The effectful outer let executes its finite policy spine in order and subjects every retained value to the concrete transport before continuing.

Ground adequacy proof. Forcing preserves denotation, including fixed-point unfolding and closure replacement. Conversely, a computational logical relation connects each finite denotational approximation to the suspended syntax; its arrow clause tests application and its lifting clause requires a returned outer lift. Induction on approximation depth and syntax proves this relation, with the recursion case using the preceding approximation. The increasing approximations have the suspension's denotation as their supremum$^{\mathsf{L}^{\exists\forall}\mathsf{N}}$. At a protected lifted Boolean, a returned denotation occurs at some finite approximation. The logical relation then supplies an operational observation; soundness gives the converse. The dynamic machine returns a transported environment and a suspension whose denotations agree with `DynamicTerm.run`. Thus ground observation and denotation agree for the complete finite policy spine, with unrestricted static recursion$^{\mathsf{L}^{\exists\forall}\mathsf{N}}$.

The term $\Omega_A = \mu f : A_\perp.f$ is well typed and diverges; its denotation is bottom. A genuine recursive function is

$$\mu f : \mathsf{Bool} \to (T_{\ell_o}\ \mathsf{Bool})_\perp.\lambda u : \mathsf{Bool}.\mathsf{if}\ u\ \mathsf{thenlift}\,[\mathsf{true}]_{\ell_o}\ \mathsf{else}\ f\,u.$$

It returns on true and diverges on false. Its codomain is pointed, and its control is public. Lean checks this syntax and its actual interpretation: the true application returns after one unfolding, whereas the false application has no returning finite approximation$^{\mathsf{L}^{\exists\forall}\mathsf{N}}$. The normalization argument in Theorem 9 therefore continues to concern only the terminating subfragment.

Extended soundness proof.

Proposition 1 (Extended release-relative soundness$^{\mathsf{L}^{\exists\forall}\mathsf{N}}$). *Assume the continuous admissible DCC interpretation above. Assume each primitive supplies a total continuous context transport satisfying Eq.* (1)*, and the outer dynamic syntax remains finite, with static guards and common branch endpoints. Then related initial environments with equal accumulated releases yield related transported contexts and related result denotations, including at function and pointed types.*

Proof (Proposition 1). The fixed-policy fundamental argument uses the function and least-fixed-point cases just given. Induct on the finite dynamic syntax using the transported context-and-result invariant. Change uses its transport certificate, let pairs releases and composes transports, and a Boolean guard selects the same branch in related environments. This is the same release-refined composition argument as Theorem 8, with arbitrary carrier maps. Lean checks the full invariant and its categorical release-refined form$^{\mathsf{L}^{\exists\forall}\mathsf{N}}$.

Operational corollary. If, additionally, these primitive transports and the resulting ground domain interpretation are computationally adequate, the strong relation yields *termination-sensitive* ground NI. For $x : T_{Hi}\ A; p_i \vdash M : T_{Lo}\ (\mathsf{Bool}_\perp); p_o$, $Hi \not\sqsubseteq_{p_i} Lo$, and equal accumulated releases, related closed inputs a_L, a_R satisfy, for each Boolean b,

$$M[a_L/x] \Downarrow [\mathsf{lift}\, b]_{Lo} \quad \Longleftrightarrow \quad M[a_R/x] \Downarrow [\mathsf{lift}\, b]_{Lo}.$$

Here $\Downarrow$ means forcing the receiving observer's lifted value to a Boolean. Related results have equal denotations at this visible flat lifted type: bottom is related only to bottom, and returned Booleans must agree. Adequacy transfers this equality to agreement on convergence and its value; it does not assert convergence. The original DCC fixed-policy adequacy does not discharge the new dynamic closure-transport adequacy premise. Lean checks equality of the partial denotations and the transfer to the stated convergence equivalence$^{\mathsf{L}\exists\forall\mathsf{N}}$. The generic theorem takes computational adequacy as an explicit premise. For the concrete syntactic call-by-name machine, the adequacy proof above discharges it, yielding operational NI with no adequacy premise$^{\mathsf{L}\exists\forall\mathsf{N}}$. The machine checks an ignored divergent argument, a nonreturning loop, and a recursive result observed after revocation$^{\mathsf{L}\exists\forall\mathsf{N}}$.

C Mechanization and Proof Coverage

We mechanize all claims developed in the main paper in Lean 4, using Mathlib [27] for categorical and order-theoretic infrastructure, as detailed in Section C.1. The proofs cover the categorical construction, the complete terminating first-order language, and concrete higher-order soundness and call-by-name ground adequacy, together with the marked extension instances, under the fragments and hypotheses stated in the paper. The $^{\mathsf{L}\exists\forall\mathsf{N}}$ annotations identify the corresponding Lean declarations. Lean 4.29.1 is pinned in `lean-toolchain`; `lakefile.toml` and `lake-manifest.json` pin Mathlib [27] and its transitive dependencies. Run `make check-sync` from the paper directory to build the project, check the source inventory, and resolve every declaration named by a paper checkmark. The generated inventory is `lean/coverage_audit/report.txt`; it records result statements, scope, and source fingerprints. Name resolution does not mechanically establish correspondence between LaTeX statements and Lean types.

– **Categorical model.** `Classified.lean` defines the relation-level release condition and checks release refinement, composition, and constancy for observer-indexed reflexive relations. `Dynamic.lean` supplies the preservation lemmas used by the category. `ModalCategory.lean` constructs the category represented by `Modal.World` and `Modal.Arrow`, including its category laws (Theorem 5), and the redaction endofunctor, natural unit and multiplication, monad laws, and invertible multiplication (Theorem 6). It also proves the paired-release kernel identity and the arrow-level visible-result statement of Theorem 7; `Classified.lean` supplies its release-map specializations. `RefinementPullback.lean` checks all four maps, commutativity, factorization, and uniqueness for Figure 2 in the ordinary classified-set category, including empty observer types.

- **Static first-order language.** `CoreDCC.lean` checks intrinsically typed first-order syntax, protection, preservation, and protected-Boolean NI. `Adequacy.lean` gives canonical values and independent big-step evaluation, proving normalization, soundness, determinism, and adequacy. `StaticDCC.lean` checks the abstract constancy/adequacy interface. These are first-order counterparts of Section 2, rather than a formalization of full DCC's equational theory.
- **Changing-flow language.** `DynamicDCC.lean` checks generic composition, including retention of the context and computed values. `IntegratedDCC.lean` instantiates it with typed CoreDCC terms and a separate flow preorder at each policy. Its proofs establish Theorems 8, 9, and 10, and also the conditional cases in Appendix A.4. `CertifiedDCC.lean` packages terms with their local security proofs: its change constructor requires the `Secure` premise of Figure 4. Thus the certified NI theorem needs no separate update-security assumption$^{\mathsf{L}\exists\forall\mathsf{N}}$. Executable examples equivalently use raw terms together with their certificates.
- **Worked examples.** The five examples in `Examples/` follow the main paper's programs. `Hospital.lean` defines `hospitalExample`, the complete copy/discharge/transfer program of Section 4.2, with the introductory transfer fragment inlined and marked in its body. It checks authorized and rejected transfers, including rejection when another newly visible hospital value is omitted from the release. `Conditionals.lean`, `Functions.lean`, `Recursion.lean`, and `Store.lean` define `extC`, `extF`, `extR`, and `extAll`, respectively, matching Sections 5.1–5.4. Their comments identify the paper's variables, labels, and policies. They check the selected-secret release, closure revocation, public recursive branching and protected-bind restriction, and store-passing countdown and divergence. Auxiliary operational checks exercise the appendix's call-by-name evaluation and closure transport.
- **Functions and recursion.** `RecursiveDCC.lean` checks typed static function/lifting/recursion syntax, admissible continuous arrow objects, abstraction and application, strict lifting relations, and relational least-fixed-point induction, unfolding, and leastness. `RecursiveDCC/HigherOrder.lean` specifies the assumed policy-indexed interpretation through `Model` and `StaticSemantics`, derives the full static fundamental theorem, and defines a finite dynamic interpreter with total continuous context transports. It proves Proposition 1, its categorical form, and semantic NI for protected lifted Booleans. The generic convergence corollary assumes computational adequacy. `RecursiveDCC/Concrete*.lean` construct the continuous model, static semantics, and certified closure-replacement transports. `RecursiveDCC/CallByName*.lean` define a syntactic closure machine, prove ground adequacy for unrestricted static recursion and the finite dynamic policy spine, and derive operational NI without an adequacy premise.
- **Putting it all together.** `Examples/Store.lean` checks the higher-order recursive store-passing loop of Section 5.4, each countdown update, the zero-counter case, a diverging cyclic update, and the final protected bit returned by the syntactic machine. These are existing DDCC terms.

`Audit.lean` inspects every declaration in the retained project modules, including auxiliary and private declarations. The default build rejects transitive axiom dependencies outside `propext`, `Classical.choice`, and `Quot.sound`, and reports dependencies for the paper's marked results. There are no admitted goals or additional axiom declarations. The category and continuous-domain constructions use Mathlib's standard classical infrastructure. No dynamic pre-cohesive

adjoint string, concurrency semantics, or recursive policy-update interpreter is claimed.

C.1 Mathlib Support

Categorical infrastructure. Mathlib supplies the structures for categories, functors, natural transformations, monads, and natural isomorphisms used in `ModalCategory.lean`. Its setoids represent the equivalence release bounds, with intersection and inverse image providing composition of bounds. Order homomorphisms organize the preservation obligations through `reindex`. The refinement square is checked using Mathlib's `PullbackCone`, `IsLimit`, and `IsPullback` interfaces. The artifact defines the classified objects and release-bounded arrows and proves their category laws, the redaction monad laws, and the refinement universal property within these interfaces.

Order-theoretic infrastructure. Mathlib supplies ω-complete partial orders, increasing chains, continuous maps, and fixed-point iteration. The recursion construction uses its iteration-chain, unfolding, and leastness results. On top of this infrastructure, the artifact constructs the admissible relational function objects and the continuous lifted domains based on `WithBot`, and proves the relational fixed-point argument. The static interpretation, policy transports, syntactic call-by-name machine, and ground adequacy proof are supplied by the artifact.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 147–160 (1999). `https://doi.org/10.1145/292540.292555`
22. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Software Security—Theories and Systems. LNCS, vol. 3233, pp. 174–191. Springer (2004). `https://doi.org/10.1007/978-3-540-37621-7_9`
27. The mathlib Community: The Lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 367–381. CPP 2020, Association for Computing Machinery (2020). `https://doi.org/10.1145/3372885.3373824`