Internalizing Indistinguishability with Dependent Types

YIYUN LIU, University of Pennsylvania, USA
JONATHAN CHAN, University of Pennsylvania, USA
JESSICA SHI, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

In type systems with dependency tracking, programmers can assign an ordered set of levels to computations and prevent information flow from high-level computations to the low-level ones. The key notion in such systems is *indistinguishability*: a definition of program equivalence that takes into account the parts of the program that an observer may depend on. In this paper, we investigate the use of dependency tracking in the context of dependently-typed languages. We present the Dependent Calculus of Indistinguishability (DCOI), a system that adopts indistinguishability as the definition of equality used by the type checker. DCOI also internalizes that relation as an observer-indexed propositional equality type, so that programmers may reason about indistinguishability within the language. Our design generalizes and extends prior systems that combine dependency tracking with dependent types and is the first to support conversion and propositional equality at arbitrary observer levels. We have proven type soundness and noninterference theorems for DCOI and have developed a prototype implementation of its type checker.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Modes, Dependent Types, Coq, Formalization

ACM Reference Format:
Yiyun Liu, Jonathan Chan, Jessica Shi, and Stephanie Weirich. 2024. Internalizing Indistinguishability with Dependent Types. *Proc. ACM Program. Lang.* 8, POPL, Article 44 (January 2024), 28 pages. https://doi.org/10.1145/3632886

1 INTRODUCTION

Dependency tracking is a static analysis that determines how computations depend on their inputs. Type systems that support dependency tracking assign levels drawn from some partial order to computations, and prevent the flow of information from higher level computations to lower ones.

For example, consider the type signature of a function f as follows, where the levels L (for low) and H (for high) are ordered $\mathsf{L} < \mathsf{H}$.

$$\mathsf{f} :^{\mathsf{L}} \mathsf{Nat}^{\mathsf{H}} \to \mathsf{Nat}$$

The type signature reads that f is a function that takes a natural number typed at level H and returns a natural that is typed at level L. Because $\mathsf{L} < \mathsf{H}$, the function f may not use its argument and must behave like a constant function.

Dependency tracking is a general feature of type systems and has a variety of applications [Abadi et al. 1999], most frequently information flow analysis [Denning and Denning 1977; Sabelfeld and Myers 2003], but also partial evaluation [Hatcliff and Danvy 1997] and staged programming [Sheard

Authors' addresses: Yiyun Liu, University of Pennsylvania, USA, liuyiyun@seas.upenn.edu; Jonathan Chan, University of Pennsylvania, USA, jcxz@seas.upenn.edu; Jessica Shi, University of Pennsylvania, USA, jwshi@seas.upenn.edu; Stephanie Weirich, University of Pennsylvania, USA, sweirich@seas.upenn.edu.

ACM 2475-1421/2024/1-ART44
https://doi.org/10.1145/3632886

and Nelson 1995; Taha and Sheard 2000]. In the context of dependent type systems, it is related to relevance tracking [Choudhury et al. 2022; de Bruijn 1994; Miquel 2001; Mishra-Linger and Sheard 2008], two-level type theory [Annenkov et al. 2023], and the phase distinction in the ML module system [Sterling and Harper 2021].

Levels used for dependency tracking take on different meanings depending on the specific application. For example, in relevance tracking, the dependency levels L and H can represent relevant and irrelevant computations, respectively, where irrelevant computations can be erased prior to run time. For information flow, the dependency levels correspond to clearance levels and only computations typed at level H can access classified information. For each application, dependency levels are drawn from a predetermined partially ordered set or a richer structure such as a lattice.

When reasoning about program equivalence in a system that supports dependency tracking, the key idea is *indistinguishability*: an equivalence relation that is indexed by an observer level [Sabelfeld and Myers 2003]. This relation defines program equivalence from the point of view of an observer—all parts of the program that are unobservable at that level can be safely equated because the system prevents the observer from depending on those values.

Dependently-typed languages support intrinsic reasoning about program equivalence. Calculi such as PTS [Barendregt 1993], CIC [Coquand and Paulin 1990], and MLTT [Martin-Löf 1975] characterize what it means for programs to be equal and use this definition of equality for type conversion. In these systems, two terms are typically definitionally equal when they are β-equivalent or $\beta\eta$-equivalent. Furthermore, the propositional equality type internalizes the equality judgment as a mechanism for intrinsic reasoning about program equivalence. In CIC and MLTT, the type $a \sim b \in A$ asserts that the terms a and b (of type A) are equal. If this proposition is provable, we say that a and b are propositionally equal. In dependent type systems, more terms are propositionally equal than definitionally equal: for example, one can use induction on naturals to prove $x + 0 \sim x \in \mathsf{Nat}$ even when $x + 0$ and x are not definitionally equal. By internalizing definitional equality as the equality type, programmers gain a powerful tool for reasoning about program equivalence.

However, the power of this tool depends on the definition of equivalence in the first place. In the context of dependency tracking, it is sound to equate programs that are not necessarily $\beta\eta$-equivalent as long as their differences are not observable. To explore this idea, we present the Dependent Calculus of Indistinguishability (DCOI)[1], a dependently-typed calculus that uses *indistinguishability* as its definition of equality.

DCOI can type check programs that would be rejected by CIC or MLTT. For example, given a function P with the signature $\mathsf{P} :^{\mathsf{L}} \mathsf{Bool}^{\mathsf{H}} \to \mathsf{Type}$, we should be able to safely convert between the types $\mathsf{P}\ \mathsf{True}$ and $\mathsf{P}\ \mathsf{False}$. During conversion, the type checker can skip the comparison of True and False because the signature of P tells us that P cannot use its argument in a way that is observable at level L.

In exploring the design of DCOI, this paper makes the following contributions.

- When using indistinguishability as definitional equality, there is the question of what observer level should be used. While it is tempting to use a fixed observer level for type checking [Choudhury et al. 2022], we show here that it is sound to use *any* observer level that is compatible with the terms being compared. Furthermore, this relation can be composed step-wise: if a is equal to b at level ℓ_1 and b is equal to c at level ℓ_2, then we can also equate a and c, even though those terms may not be equal at $\ell_1 \wedge \ell_2$. We define how dependency tracking works in DCOI in Section 3 and its role in definitional equality in Section 3.3.

[1] pronounced "decoy"

- We internalize the indistinguishability judgment in DCOI as an indexed equality type (Section 3.4). Programmers can use this type to reason about program equivalences at a specified observer level. For security type systems, the observer-indexed equality type can be used to show that two programs appear identical to observers with lower clearance levels, ensuring that no secret information is leaked after, say, a significant refactoring. In a system with relevance tracking, programmers can reason about programs without concerning themselves with the irrelevant type-level computations that are meant to be erased after compilation. This design crucially depends on the flexible definition of equality above: the elimination rule for this type must be able to reason about convertibility at any observer level.
- To ensure that our design makes sense, we have mechanically checked its metatheoretic properties, including type soundness and noninterference. Our type soundness result requires showing the consistency of definitional equality, or a proof that equal types have the same head form. We provide an overview of these proofs in Section 4.
- We identify constraints on dependency tracking that are imposed by our use of indistinguishability for type conversion and discuss these constraints in Section 5. Our design contrasts with prior work [Abadi et al. 1999; Choudhury et al. 2022; Shikuma and Igarashi 2008] that we have found to be incompatible with our approach. The insights gained in exploring this design space also explain why several existing mechanisms for tracking run-time irrelevance in dependently typed languages cannot be extended to compile-time irrelevance.
- To understand how DCOI works in practice, we have developed a prototype implementation that extends our core language with level-indexed data types. We have used this implementation to experiment with the development of sample programs. In the next section, we use these examples to provide an introduction to DCOI. We discuss our implementation in more detail in Section 6.

Our Coq proof scripts and prototype implementation are available as supplementary materials.

2 MOTIVATING EXAMPLES

In this section, we present three applications of dependency analysis as motivating examples: run-time irrelevance,[2] compile-time irrelevance,[3] and secure information flow.[4]

All examples in this section have been type checked by our implementation, with footnotes pointing to the corresponding source file. For clarity, we omit some type and level annotations and use the concrete syntax of DCOI as presented in the next section.

To make these examples more realistic, we use data definitions that are not in the DCOI core calculus but are supported by our prototype implementation (Section 6). We also use constructs desugared from DCOI's dependent pairs: nondependent pairs; dependent pairs where one component is labeled with a different level; and box types $T^{\ell}\ A$, similar to the graded modal type of Abadi et al. [1999], that encapsulate terms at a higher level.

2.1 Run-Time Irrelevance

Parts of programs that are not computationally meaningful when run can be considered *run-time irrelevant*. An optimizing compiler can erase the run-time irrelevant parts to yield a more space- and time-efficient program, because those arguments do not need to be kept around or reduced needlessly [Brady 2005; Paulin-Mohring 1989]. Run-time irrelevance annotations tell the compiler what can be erased, and in a well typed program, irrelevant terms are never used relevantly.

[2] `pi/RunTime.pi` [3] `pi/CompileTime.pi` [4] `pi/InfoFlow.pi`

For instance, consider the (simplified) polymorphic constant function, whose type argument and unused input argument we mark irrelevant with label H. Unlabeled arguments and definitions are treated as relevant by default, sometimes labeled with L for clarity, where L < H.

```
const :L (A :H Type) → A → A^H →A
const = λA :H Type. λx : A. λy :H A. x
```

The L annotation next to const says that const is a relevant computation. Since const is a function, the relevance of const is determined by the relevance of its output. That is, the output of a relevant function can only use an irrelevant argument in irrelevant positions, as part of a type annotation (as is the case for the type argument A) or as an argument to another function that takes an irrelevant argument. If we try to return the irrelevant argument y instead as the output for const, the program will no longer type check. Since well-typedness enforces that the first and third arguments of const cannot be used relevantly in its body, both arguments can be safely erased at run-time.

Run-time irrelevance is also useful for extracting only the relevant parts out of proof-carrying code. Consider the following inductive type representing the proposition that a natural is even. Its constructors are labeled as irrelevant because we intend to erase such proofs.

```
data Even : Nat → Type where
  ZEven :H Even 0
  SSEven :H (n : Nat) → (Even n)^H →Even (Succ (Succ n))
```

To represent the type of even naturals, we use the subset type {x :L Nat | Even x}. Its elements are relevant dependent pairs whose first component is a relevant natural, and whose second component is an irrelevant proof of evenness. Using a lemma stating that the sum of two even naturals is even (definition elided),

```
evenEven :H (n : Nat) → (m : Nat) → (Even n)^H →(Even m)^H → Even (add n m)
```

we can define addition on even naturals. If all its run-time irrelevant parts—the proofs of evenness—are erased, this function corresponds exactly to the usual addition on naturals.

```
addEven : {x : Nat | Even x} → {x : Nat | Even x} → {x : Nat | Even x}
addEven (n, en) (m, em) = (add n m, evenEven n m en em)
```

Consider next the following inductive type for length-indexed vectors, whose length index is labeled as irrelevant. These vectors can be erased to ordinary inductive lists.

```
data Vector (A :H Type) :H Nat^H →Type where
  Nil  : Vector A 0
  Cons : (n :H Nat) → A → Vector A n → Vector A (Succ n)
```

If we were to implement a map function over vectors, the index of the vector ensures that it returns a vector with the same length, and by marking the index as irrelevant, the function can be erased at run time to one that looks just like a map function on lists.

```
map : (A :H Type) → (B :H Type) → (n :H Nat) →
      (A → B) → Vector A n → Vector B n
map A B n f v = case v of
  Nil ⇒ Nil
  Cons n′ y ys ⇒ Cons n′ (f y) (map A B n′ f ys)
```

It is worth pointing out that the irrelevant arguments A,B, and n do appear syntactically in the body of map. However, this is allowed since they all appear as irrelevant arguments to a function, which, in this case, is map itself.

2.2 Compile-Time Irrelevance

Relevance tracking is not only useful for run-time erasure, but also for ignoring terms during type conversion, known as *compile-time irrelevance* [de Bruijn 1994]. It allows us to type check the following program where the second argument is a type constructor that ignores its input.

$$\begin{aligned}&\mathsf{substNat}\ :\ (\mathsf{n}\ :^{\mathsf{H}}\ \mathsf{Nat}) \to (\mathsf{P}\ :^{\mathsf{L}}\ \mathsf{Nat}^{\mathsf{H}} \to \mathsf{Type})\ \to\ \mathsf{P}\ \mathsf{n} \to \mathsf{P}\ 0\\ &\mathsf{substNat}\ \mathsf{n}\ \mathsf{P}\ \mathsf{e}\ =\ \mathsf{e}\end{aligned}$$

In substNat, the type checker converts the type of the argument e from P n to the return type P 0. This is valid because P is typed at level L with an input typed at level H, so these terms are indistinguishable to an L observer.

However, it is not always possible to type a type constructor at level L. Recall Vector, which takes in a type parameter and length index that we regard as run-time irrelevant, labeled H. If we tried to type the overall function Vector at level L, then Vector Bool 1 and Vector Nat 2 would be indistinguishable, which would violate type safety since we can now extract an element out of a boolean vector and treat it as a natural number. To ensure we do not convert between these two types, Vector must be typed at level at least H.

With only two levels L and H, we cannot describe what is irrelevant to a type already typed at level H. Being able to talk about irrelevance at level H is useful in practice: we may want to treat proofs of some propositions as compile-time irrelevant or erase irrelevant components to speed up type-level reduction. We can add a label S (for super-high) where $\mathsf{L} < \mathsf{H} < \mathsf{S}$. This allows us to define the following program where the evenness proof, typed at level S, is ignored by a type constructor P typed at level H.

$$\begin{aligned}&\mathsf{substEven}\ :\ (\mathsf{P}\ :^{\mathsf{H}}\ \mathsf{Nat}^{\mathsf{H}} \to (\mathsf{Even\ n})^{\mathsf{S}} \to\ \mathsf{Type})\ \to\ (\mathsf{n}\ :^{\mathsf{H}}\ \mathsf{Nat}) \to\\ &\qquad\qquad (\mathsf{en}_1 :^{\mathsf{S}} \mathsf{Even\ n}) \to (\mathsf{en}_2 :^{\mathsf{S}} \mathsf{Even\ n}) \to \mathsf{P\ n\ en}_1 \to\ \mathsf{P\ n\ en}_2\\ &\mathsf{substEven\ n\ P\ en}_1\ \mathsf{en}_2\ \mathsf{p} = \mathsf{p}\end{aligned}$$

Intuitively, with the ordering of relevance labels $\mathsf{L} < \mathsf{H} < \mathsf{S}$, terms at higher relevance labels are indistinguishable from the perspective of observers at lower relevance labels. Since the function substEven is typed at level L, all of its argument except for p are erasable at run time. The type constructor P, on the other hand, is typed at level H. In $\mathsf{P\ n\ en}_1$ and $\mathsf{P\ n\ en}_2$, the natural n is typed at level H and used relevantly at compile time, whereas the evenness proofs en_1 and en_2 are both typed at level S and used irrelevantly at compile time. As a result, we can convert between evenness proofs, but the natural must remain the same.

DCOI also includes a propositional equality type $a \sim^{\ell} b$ that internalizes indistinguishability observed at level ℓ. The **refl** term introduces elements of this type. For example, $\mathsf{P\ en}_1 \sim^{\mathsf{H}} \mathsf{P\ en}_2$ is inhabited by **refl**, since the two arguments are indistinguishable to H observers.

This level-indexed equality type enables reasoning about equalities at that level. For example, suppose we have a lemma showing commutativity of natural numbers (definition elided).

$$\mathsf{addComm} : (\mathsf{n} :\ \mathsf{Nat}) \to (\mathsf{m} :\ \mathsf{Nat}) \to \mathsf{add\ n\ m} \sim^{\mathsf{L}} \mathsf{add\ m\ n}$$

We can use this lemma with **transp**, the elimination term for propositional equality, to prove the commutativity of adding two even naturals.

$$\begin{aligned}&\mathsf{addEvenComm} : (\mathsf{en} : \{\mathsf{n}\ :\ \mathsf{Nat}\ \mid\ \mathsf{Even\ n}\}) \to (\mathsf{em} :\ \{\mathsf{m} :\ \mathsf{Nat}\ \mid\ \mathsf{Even\ m}\}) \to\\ &\qquad\qquad \mathsf{addEven\ en\ em} \sim^{\mathsf{L}} \mathsf{addEven\ em\ en}\end{aligned}$$

```
addEvenComm (n, en) (m, em) = transp (addComm n m) refl
```

Given (n, en) and (m, em), our goal is to show that (add n m, evenEven n m en em) and (add m n, evenEven m n em en) are propositionally equal. In addition to showing that (add n m) and (add m n) are equal, we would normally need to manually prove that (evenEven n m en em) and (evenEven m n em en) are equal, further complicated by their definitionally unequal types Even (add n m) and Even (add m n). However, since these proofs are compile-time irrelevant, and we wish to prove a relevantly-indexed equality, the type checker ignores the irrelevant second components, and it suffices to prove the relevant first components equal.

2.3 Information Flow

Dependency analysis can be used to model *secure information flow* [Denning and Denning 1977], where information can never flow from a higher security level H to a lower security level L. To an observer at L, values of a given type at H are indistinguishable from one another, and higher-level information therefore cannot be used meaningfully except to be passed around. On the other hand, observers with higher-level clearance are able to use lower-level information. In the following examples, unlabeled arguments and definitions are treated as low-security by default.

Consider the construct $\mathbf{box}^{\mathsf{H}}\ a$ that hides a secret value a of type A in a high-security box of type $T^{\mathsf{H}}\ A$. One way a low-clearance function can inspect and branch on a secret natural, for instance, is to always box up its output.

```
secretPred :^L (n :^H Nat) → T^H Nat
secretPred n = box^H (case n of
  Zero ⇒ Zero
  Succ m ⇒ m)
```

Unboxing a boxed secret value yields a secret value, which a low-clearance function can't do much with except to pass it to a function that takes a high security value, such as in the ap function below. All of ap's arguments are low-security and it returns a low-security value, but it can still run a secret function on a secret value without ever knowing what was done and to which value.

```
ap :^L (A : Type) → (B : Type) → T^H (A^H → B) → T^H A → T^H B
ap A B f a = box^H (unbox^H f (unbox^H a))
```

We can show metatheoretically that low-clearance functions cannot distinguish between high-level arguments, using the noninterference property of DCOI (Section 4.4). Furthermore, with internalized indistinguishability, programmers can also reason about noninterference within the language.

3 DEPENDENT CALCULUS OF INDISTINGUISHABILITY

In this section, we present the syntax, judgments, and derivation rules of DCOI. The syntax of DCOI (Figure 1) is parameterized over a lattice $\mathcal{L}$ of dependency levels (following Abadi et al. [1999]) and a set of sorts $\mathcal{S}$ (following Barendregt [1993]). We use the metavariable ℓ for dependency levels and s for sorts. For lattices, we use $\ell_1 \wedge \ell_2$ and $\ell_1 \vee \ell_2$ for the meet and join operations. For the examples in this section, we instantiate the lattice to $\{\mathsf{L}, \mathsf{H}\}$, where $\mathsf{L} < \mathsf{H}$, and the set of sorts to include the constant $\star$, which represents the type of types.

The typing judgment of DCOI takes the form $\Gamma \vdash a :^{\ell} A$. Ignoring the labels ℓ in the judgment form and its rules, the typing rules in Figure 2 are a variant of Barendregt's Pure Type Systems (PTS) [Barendregt 1993]. These rules are parameterized over a set of sorts $\mathcal{S}$, a set of axioms $\mathcal{A} \subseteq \mathcal{S}^2$, and sets of rules $\mathcal{R}_\Pi \subseteq \mathcal{S}^3$, $\mathcal{R}_U \subseteq \mathcal{S}$, $\mathcal{R}_B \subseteq \mathcal{S}$. The axioms dictate how sorts are typed

Contexts

Γ	::=	$\cdot \mid \Gamma, x{:}^{\ell} A$	variables annotated by level and type
Φ	::=	$\cdot \mid \Phi, x : \ell$	variables annotated by level

Terms

a, b, A, B	::=	$s \mid x \mid \mathbf{Unit} \mid \mathbf{unit}$	sort, variables, unit type, unit
	$\mid$	$\Pi x{:}^{\ell} A.\, B \mid \lambda x{:}^{\ell} A.\, a \mid a\ b^{\ell}$	function types, abstractions, applications
	$\mid$	$\Sigma x{:}^{\ell} A.\, B \mid (a^{\ell}, b) \mid \pi_1^{\ell}\ a \mid \pi_2^{\ell}\ a$	dependent pair types, pairs, projections
	$\mid$	$a \sim^{\ell} b \in^{\ell_0} A \mid \mathbf{refl} \mid \mathbf{transp}\ a\ b$	equality types, reflexivity proof, transport
	$\mid$	$\mathbb{B} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ a\ b_0\ b_1$	boolean type, true, false, if

Fig. 1. Syntax of DCOI

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Typing)*

$$\textsc{T-Type}\ \frac{\vdash \Gamma \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 :^{\ell} s_2} \qquad \textsc{T-Conv}\ \frac{\Gamma \vdash a :^{\ell} A \qquad \Gamma \vdash B :^{\ell_0} s \qquad |\Gamma| \vdash A \equiv B}{\Gamma \vdash a :^{\ell} B} \qquad \textsc{T-Var}\ \frac{\vdash \Gamma \qquad \ell_0 \le \ell \qquad x{:}^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^{\ell} A}$$

$$\textsc{T-Pi}\ \frac{\Gamma \vdash A :^{\ell} s_1 \qquad \Gamma, x{:}^{\ell_0} A \vdash B :^{\ell} s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}_{\Pi}}{\Gamma \vdash \Pi x{:}^{\ell_0} A.\, B :^{\ell} s_3} \qquad \textsc{T-Abs}\ \frac{\Gamma, x{:}^{\ell_0} A \vdash b :^{\ell} B \qquad \Gamma \vdash (\Pi x{:}^{\ell_0} A.\, B) :^{\ell_1} s}{\Gamma \vdash \lambda x{:}^{\ell_0} A.\, b :^{\ell} \Pi x{:}^{\ell_0} A.\, B}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash b :^{\ell} \Pi x{:}^{\ell_0} A.\, B \qquad \Gamma \vdash a :^{\ell_0} A}{\Gamma \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}}$$

$$\textsc{T-TyUnit}\ \frac{\vdash \Gamma \qquad s \in \mathcal{R}_U}{\Gamma \vdash \mathbf{Unit} :^{\ell} s} \qquad \textsc{T-TmUnit}\ \frac{\Gamma \vdash \mathbf{Unit} :^{\ell_0} s}{\Gamma \vdash \mathbf{unit} :^{\ell} \mathbf{Unit}} \qquad \textsc{T-Bool}\ \frac{\vdash \Gamma \qquad s \in \mathcal{R}_B}{\Gamma \vdash \mathbb{B} :^{\ell} s} \qquad \textsc{T-True}\ \frac{\Gamma \vdash \mathbb{B} :^{\ell_0} s}{\Gamma \vdash \mathbf{true} :^{\ell} \mathbb{B}}$$

$$\textsc{T-False}\ \frac{\Gamma \vdash \mathbb{B} :^{\ell_0} s}{\Gamma \vdash \mathbf{false} :^{\ell} \mathbb{B}} \qquad \textsc{T-If}\ \frac{\Gamma \vdash a :^{\ell} \mathbb{B} \qquad \Gamma \vdash b_0 :^{\ell} A \qquad \Gamma \vdash b_1 :^{\ell} A}{\Gamma \vdash \mathbf{if}\ a\ b_0\ b_1 :^{\ell} A}$$

$\boxed{\vdash \Gamma}$ *(Context well-formedness)*

$$\textsc{Ctx-Empty}\ \frac{}{\vdash \cdot} \qquad \textsc{Ctx-Cons}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A :^{\ell} s \qquad x \notin \operatorname{dom} \Gamma}{\vdash \Gamma, x{:}^{\ell_0} A}$$

Fig. 2. Typing rules (selected)

in rule T-Type, and the rules dictate the sorts assigned to types. For example, function types are typed with respect to the sorts of their domain and codomain types in rule T-Pi. The sorts, axioms, and rules can determine the expressiveness of the language and its normalization properties. For

example, by instantiating $\mathcal{S}$ to $\star$, $\mathcal{A}$ to $\{(\star, \star)\}$, and $\mathcal{R}_\Pi$ to $\{(\star, \star, \star)\}$, we obtain a system with unrestricted access to type-level computation, but which fails the normalization property due to Girard's paradox [Girard 1972].[5]

DCOI supports dependency tracking, as described in Section 3.1. This leads to the most significant divergence from Barendregt's system, the definitional equality relation used in rule T-Conv. This rule permits converting the type of a term to a definitionally equal type, but whereas Barendregt uses β-equivalence, DCOI makes use of the indistinguishability relation introduced in Section 3.3. We also extend the core system with dependent pairs (Section 3.2) and propositional equality (Section 3.4). The boolean type $\mathbb{B}$ is used as a base type to formulate the noninterference property (Section 4.4). For concision, we use $A^\ell \to B$ as a shorthand for $\Pi x :^\ell A.\, B$ when x does not appear in B.

3.1 Dependency Tracking

The level ℓ from the typing judgment $\Gamma \vdash a :^\ell A$ indicates the *observer level* used to type check the term a. As enforced in the precondition of rule T-Var, to access a variable, the observer level of the computation must be at least as high as the level at which the variable is labeled.

Functions that type check at a lower observer level may still accept a higher-level argument, as long as the argument is only used in higher-level positions. Rule T-Abs constructs a function that expects a computation with observer level ℓ_0 as its argument. Correspondingly, rule T-App checks the function argument at level ℓ_0, a level that is completely independent of the observer level ℓ. Both the abstraction and application forms are annotated with this argument level.

As an example of how rules T-Var, T-Abs, and T-App support relevance tracking, consider the following term with a hole ($\square$).

$$\lambda x_0 :^{\mathsf{H}} \mathbb{B}.\, \lambda x_1 :^{\mathsf{L}} (\mathbb{B}^{\mathsf{H}} \to \mathbb{B}).\, \square$$

If we want to type check the term at level L, we cannot fill $\square$ with x_0 since the observer level required to access x_0 is H. However, if we substitute $x_1\ {x_0}^{\mathsf{H}}$ for $\square$, the term type checks at L despite x_0 appearing in the function body; when checking x_0 as an argument to the function x_1, rule T-App changes the observer level from L to H so x_0 can be legally used. Rule T-Abs ensures that all functions that type check at level L with type $\mathbb{B}^{\mathsf{H}} \to \mathbb{B}$ behave like constant functions. The function $\lambda x :^{\mathsf{H}} \mathbb{B}.\, x$, for example, is ill-typed at level L since x is at level H whereas the observer level is L. In other words, the declaration $x_1 :^{\mathsf{L}} \mathbb{B}^{\mathsf{H}} \to \mathbb{B}$ encodes the contract that x_1 does not use its input to construct its result in a way that can be observed at level L. This gives an intuitive justification for why the term $\lambda x_0 :^{\mathsf{H}} \mathbb{B}.\, \lambda x_1 :^{\mathsf{L}} (\mathbb{B}^{\mathsf{H}} \to \mathbb{B}).\, x_1\ {x_0}^{\mathsf{H}}$, which type checks at L, does not leak the high-level information from x_0.

The observer level ℓ at the colon does not have to be precise. The following subsumption lemma says if a term type checks at level ℓ, then it also type checks at levels higher than ℓ. It can be viewed as a propagation of variable level subsumption $\ell_0 \le \ell$ from rule T-Var to the rest of the system. In terms of relevance tracking, this lemma embodies the idea that relevant computations can be lifted to become irrelevant.

Lemma 3.1 (Subsumption[6]). *If* $\Gamma \vdash a :^{\ell_0} A$ *and* $\ell_0 \le \ell$, *then* $\Gamma \vdash a :^{\ell} A$.

In rule T-Abs, we check that the function type can be assigned a sort s. The most interesting part of this premise is the choice of the function type level ℓ_1, which is independent of the observer level ℓ and the argument level ℓ_0. To show that a type is well formed, it suffices to show that the type is well formed at some arbitrarily chosen level. The flexible choice of the observer level for the well-formedness check affects how we formulate the regularity property.

[5] In Section 5.4, we discuss our conjectures about the normalization behavior of various instantiations of DCOI.
[6] upgrade.v:typing_subsumption

Lemma 3.2 (Regularity[7]). *If* $\Gamma \vdash a :^{\ell} A$, *then there exists a level* ℓ_0 *and a sort* s *such that* $\Gamma \vdash A :^{\ell_0} s$.

If we instantiate ℓ to a bounded lattice, Lemma 3.2 is equivalent to the statement that the types of terms are always well typed at the topmost level. Intuitively, we are allowed to check well-formedness at our level of choice because given $\Gamma \vdash a :^{\ell} A$, DCOI does not have an operator to allow the retrieval of type A from the term a at run time. As a result, we can safely allow A to rely on secrets that are not accessible to a without worrying about any leakage.

3.2 Dependent Pairs

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Typing: dependent pairs)*

T-SSigma
$$\frac{\Gamma \vdash A :^{\ell} s_1 \qquad \Gamma, x:^{\ell_0} A \vdash B :^{\ell} s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}_\Sigma}{\Gamma \vdash \Sigma x:^{\ell_0} A.\, B :^{\ell} s_3}$$

T-SPair
$$\frac{\Gamma \vdash \Sigma x:^{\ell_0} A.\, B :^{\ell_1} s \qquad \Gamma \vdash a :^{\ell_0} A \qquad \Gamma \vdash b :^{\ell} B\{a/x\}}{\Gamma \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x:^{\ell_0} A.\, B}$$

T-Proj1
$$\frac{\Gamma \vdash a :^{\ell} \Sigma x:^{\ell_0} A.\, B \qquad \ell_0 \le \ell}{\Gamma \vdash \pi_1^{\ell_0}\ a :^{\ell} A}$$

T-Proj2
$$\frac{\Gamma \vdash a :^{\ell} \Sigma x:^{\ell_0} A.\, B \qquad \Gamma \vdash \pi_1^{\ell_0}\ a :^{\ell_0} A}{\Gamma \vdash \pi_2^{\ell_0}\ a :^{\ell} B\{\pi_1^{\ell_0}\ a/x\}}$$

Fig. 3. Typing rules for dependent pairs

The typing rules for dependent pairs can be found in Figure 3. Dependent pair types take the form $\Sigma x:^{\ell_0} A.\, B$. To govern the formation of these types, we extend our rule sets to include $\mathcal{R}_\Sigma$, the set of rules dictating which pair types are valid, analogous to $\mathcal{R}_\Pi$. The label ℓ_0 in a type $\Sigma x:^{\ell_0} A.\, B$ specifies the level of the first component: in rule T-SPair, the first component a is checked at the labeled level ℓ_0 rather than the observer level ℓ. This is analogous to rule T-App where the argument of the function is also checked independently at the labeled level.

Dependent pairs are eliminated by two projection operators $\pi_1^{\ell_0}\ a$ and $\pi_2^{\ell_0}\ a$. Rule T-Proj1 projects out the first component when the observer level ℓ is greater than or equal to the labeled level ℓ_0, while rule T-Proj2 projects out the second component. The precondition $\Gamma \vdash \pi_1^{\ell_0}\ a :^{\ell_0} A$ in rule T-Proj2 ensures the well-formedness of the type $B\{\pi_1^{\ell_0}\ a/x\}$. This condition is admissible when $\ell \le \ell_0$. Given $\Gamma \vdash a :^{\ell} \Sigma x:^{\ell_0} A.\, B$, we can derive $\Gamma \vdash a :^{\ell_0} \Sigma x:^{\ell_0} A.\, B$ by subsumption (Lemma 3.1). We can then apply rule T-Proj1 to obtain $\Gamma \vdash \pi_1^{\ell_0}\ a :^{\ell_0} A$. However, the condition is not admissible in the general case. When $\ell \not\le \ell_0$, the pair a may depend on variables that are accessible at level ℓ but not at ℓ_0, so the term $\pi_1^{\ell_0}\ a$ is not necessarily typable at level ℓ_0 given $\Gamma \vdash a :^{\ell} \Sigma x:^{\ell_0} A.\, B$ and $\ell \not\le \ell_0$.

The box type T^{ℓ_0} from Section 2 can be defined in terms of a dependent pair type whose first component is the type being boxed, and whose second component is unit.

$$T^{\ell_0}\ A := \Sigma x:^{\ell_0} A.\, \mathbf{Unit} \qquad \mathbf{box}^{\ell_0}\ a := (a^{\ell_0}, \mathbf{unit}) \qquad \mathbf{unbox}^{\ell_0}\ a := \pi_1^{\ell_0}\ a$$

Rule T-Proj1 enforces that we can only unbox a term if the observer level ℓ is greater than or equal to the box level ℓ_0. We can derive the following admissible rules for boxes from the rules for dependent pairs, where the relation $\mathcal{R}_T$ is defined in terms of $\mathcal{R}_\Sigma$.

T-T
$$\frac{\Gamma \vdash A :^{\ell} s_1 \qquad (s_1, s_2) \in \mathcal{R}_T}{\Gamma \vdash T^{\ell_0}\ A :^{\ell} s_2}$$

T-Box
$$\frac{\Gamma \vdash a :^{\ell_0} A \qquad \Gamma \vdash T^{\ell_0}\ A :^{\ell_1} s}{\Gamma \vdash \mathbf{box}^{\ell_0}\ a :^{\ell} T^{\ell_0}\ A}$$

T-Unbox
$$\frac{\Gamma \vdash a :^{\ell} T^{\ell_0}\ A \qquad \ell_0 \le \ell}{\Gamma \vdash \mathbf{unbox}^{\ell_0}\ a :^{\ell} A}$$

[7] regularity.v:typing_regularity

$$(s_1, s_2) \in \mathcal{R}_T \iff \exists s_3, (s_1, s_3, s_2) \in \mathcal{R}_\Sigma \wedge s3 \in \mathcal{R}_U$$

Rules T-Box and T-Unbox behave similarly to rules T-SPair and T-Proj1 . Rule T-Box allows us to place the term a inside a box with level ℓ_0 protection provided that a can type check at level ℓ_0. Rule T-Unbox allows us to release the data stored in the box, but requires that the observer level is greater than or equal to the level associated with the box type.

The box type is a graded modal type. It was first used for dependency tracking in DCC [Abadi et al. 1999] as an indexed monad type. Our formulation of the box type is most similar to a later variation of DCC called the Sealing Calculus [Shikuma and Igarashi 2008], which releases the boxed content through an **unbox** operator rather than the monadic **bind** operator from DCC.

3.3 Dependency-Aware Definitional Equality

Rule T-Conv allows us to convert between types that are definitionally equal. The definitional equality judgment in Figure 4 takes the form $\Phi \vdash a \equiv b$ where Φ is an erased context that maps variables to their levels. In rule T-Conv, the erased context $|\Gamma|$ is obtained from the typing context Γ by dropping the type annotations.

Unlike PTS, definitional equality in DCOI is defined in terms of the indistinguishability judgment, which takes the form $\Phi \vdash a \equiv^\ell b$. Indistinguishability is a coarser relation than β-equivalence as it takes advantage of the dependency information to identify terms that are not convertible through β-equivalence alone. For example, one can derive the judgment $\cdot \vdash \mathbf{box}^{\mathsf{H}}\ \mathbf{false} \equiv^{\mathsf{L}} \mathbf{box}^{\mathsf{H}}\ \mathbf{true}$. Even though the boxes store different boolean values, from an L observer's point of view, the two terms cannot be distinguished. Using rule E-One, we can inject indistinguishability into definitional equality and show that $\mathbf{box}^{\mathsf{H}}\ \mathbf{false}$ and $\mathbf{box}^{\mathsf{H}}\ \mathbf{true}$ are definitionally equal.

Rule E-Trans means that two terms are definitionally equal if they can be related by a sequence of indistinguishability judgments, each of which may have a distinct observer level. Note that there may not be a single observer level that relates all terms in the sequence. In other words, for a fixed Φ, definitional equality can be defined as the transitive closure of the relation $R(a, b) = \exists \ell, \Phi \vdash a \equiv^\ell b$.

Indistinguishability $\Phi \vdash a \equiv^\ell b$ and guarded indistinguishability $\Phi \vdash a \equiv^\ell_{\ell_0} b$ are mutually defined. In rule GE-App, we use guarded indistinguishability to skip the comparison of unobservable terms. In particular, when using guarded indistinguishability to compare terms a and b that are labeled at level ℓ_0 but being observed at level ℓ, two cases arise. Either ℓ_0 is observable at ℓ, i.e. $\ell_0 \leq \ell$, in which case we revert to indistinguishability due to rule CGE-Leq, or ℓ_0 is not observable at ℓ, in which case we immediately conclude that the terms are indistinguishable due to rule CGE-NLeq. As a result, the lower the observer level, the coarser, or more general, indistinguishability becomes, since there are more opportunities where rule CGE-NLeq is applicable.

Rule GE-AppAbs is the β-rules for functions, and the remaining rules make indistinguishability reflexive, symmetric, transitive, and congruent. The box type, defined earlier as syntactic sugar for dependent pairs, has the following admissible equality rules. The example $\cdot \vdash \mathbf{box}^{\mathsf{H}}\ \mathbf{false} \equiv^{\mathsf{L}} \mathbf{box}^{\mathsf{H}}\ \mathbf{true}$ from above holds by rules GE-Box and CGE-NLeq since $\mathsf{H} \not\leq \mathsf{L}$.

GE-T
$$\frac{\Phi \vdash A \equiv^\ell B}{\Phi \vdash T^{\ell_0}\ A \equiv^\ell T^{\ell_0}\ B}$$

GE-Box
$$\frac{\Phi \vdash a \equiv^\ell_{\ell_0} b}{\Phi \vdash \mathbf{box}^{\ell_0}\ a \equiv^\ell \mathbf{box}^{\ell_0}\ b}$$

GE-Unbox
$$\frac{\Phi \vdash a \equiv^\ell b \qquad \ell_0 \leq \ell}{\Phi \vdash \mathbf{unbox}^{\ell_0}\ a \equiv^\ell \mathbf{unbox}^{\ell_0}\ b}$$

Given $\Phi \vdash a \equiv^{\mathsf{H}} b$, it is not necessarily the case that $\Phi \vdash a \equiv^{\mathsf{L}} b$. If $\Phi \vdash a \equiv^\ell b$, neither a nor b may use variables that are unobservable at level ℓ. This restriction is imposed by rule GE-Var and rules related to projecting from pairs, which mirror similar constraints in the typing judgments. When lowering ℓ, the typing might become invalid if the observer level becomes too low to access a variable or project out a dependent pair. On the other hand, given $\Phi \vdash a \equiv^{\mathsf{L}} b$, it is not necessarily the

$\boxed{\Phi \vdash a \equiv b}$ *(Definitional equality)*

$$\textsc{E-One}\ \frac{\Phi \vdash a \equiv^{\ell} b}{\Phi \vdash a \equiv b} \qquad \textsc{E-Trans}\ \frac{\Phi \vdash a \equiv b_0 \qquad \Phi \vdash b_0 \equiv b}{\Phi \vdash a \equiv b}$$

$\boxed{\Phi \vdash a \equiv^{\ell} b}$ *(Indistinguishability)*

$$\textsc{GE-Var}\ \frac{\ell_0 \le \ell \qquad x : \ell_0\ \mathbf{in}\ \Phi}{\Phi \vdash x \equiv^{\ell} x} \qquad \textsc{GE-Type}\ \frac{}{\Phi \vdash s \equiv^{\ell} s} \qquad \textsc{GE-Sym}\ \frac{\Phi \vdash a \equiv^{\ell} b}{\Phi \vdash b \equiv^{\ell} a} \qquad \textsc{GE-Trans}\ \frac{\Phi \vdash a \equiv^{\ell} b_0 \quad \Phi \vdash b_0 \equiv^{\ell} b}{\Phi \vdash a \equiv^{\ell} b}$$

$$\textsc{GE-App}\ \frac{\Phi \vdash b_0 \equiv^{\ell} b_1 \quad \Phi \vdash a_0 \equiv^{\ell}_{\ell_0} a_1}{\Phi \vdash b_0\ a_0{}^{\ell_0} \equiv^{\ell} b_1\ a_1{}^{\ell_0}} \qquad \textsc{GE-Abs}\ \frac{\Phi, x\colon \ell_0 \vdash b_0 \equiv^{\ell} b_1}{\Phi \vdash \lambda x{:}^{\ell_0} A_0.\ b_0 \equiv^{\ell} \lambda x{:}^{\ell_0} A_1.\ b_1} \qquad \textsc{GE-AppAbs}\ \frac{\Phi \vdash a \equiv^{\ell} \lambda x{:}^{\ell_0} A.\ a_0 \quad \Phi \vdash b_0 \equiv^{\ell}_{\ell_0} b_1}{\Phi \vdash a\ b_0{}^{\ell_0} \equiv^{\ell} a_0\{b_1/x\}}$$

$$\textsc{GE-Pi}\ \frac{\Phi \vdash A_0 \equiv^{\ell} A_1 \quad \Phi, x\colon \ell_0 \vdash B_0 \equiv^{\ell} B_1}{\Phi \vdash \Pi x{:}^{\ell_0} A_0.\ B_0 \equiv^{\ell} \Pi x{:}^{\ell_0} A_1.\ B_1} \qquad \textsc{GE-TyUnit}\ \frac{}{\Phi \vdash \mathbf{Unit} \equiv^{\ell} \mathbf{Unit}} \qquad \textsc{GE-TmUnit}\ \frac{}{\Phi \vdash \mathbf{unit} \equiv^{\ell} \mathbf{unit}}$$

$$\textsc{GE-Proj1Cong}\ \frac{\Phi \vdash a_0 \equiv^{\ell} a_1 \qquad \ell_0 \le \ell}{\Phi \vdash \pi_1^{\ell_0}\ a_0 \equiv^{\ell} \pi_1^{\ell_0}\ a_1} \qquad \textsc{GE-Proj2Cong}\ \frac{\Phi \vdash a_0 \equiv^{\ell} a_1}{\Phi \vdash \pi_2^{\ell_0}\ a_0 \equiv^{\ell} \pi_2^{\ell_0}\ a_1} \qquad \textsc{GE-Proj1Beta}\ \frac{\Phi \vdash b \equiv^{\ell} (a_1{}^{\ell_0}, a_2) \quad \ell_0 \le \ell}{\Phi \vdash \pi_1^{\ell_0}\ b \equiv^{\ell} a_1}$$

$$\textsc{GE-Proj2Beta}\ \frac{\Phi \vdash b \equiv^{\ell} (a_1{}^{\ell_0}, a_2)}{\Phi \vdash \pi_2^{\ell_0}\ b \equiv^{\ell} a_2} \qquad \textsc{GE-SPair}\ \frac{\Phi \vdash a_0 \equiv^{\ell}_{\ell_0} a_1 \quad \Phi \vdash b_0 \equiv^{\ell} b_1}{\Phi \vdash (a_0{}^{\ell_0}, b_0) \equiv^{\ell} (a_1{}^{\ell_0}, b_1)} \qquad \textsc{GE-SSigma}\ \frac{\Phi \vdash A_0 \equiv^{\ell} A_1 \quad \Phi, x\colon \ell_0 \vdash B_0 \equiv^{\ell} B_1}{\Phi \vdash \Sigma x{:}^{\ell_0} A_0.\ B_0 \equiv^{\ell} \Sigma x{:}^{\ell_0} A_1.\ B_1}$$

$\boxed{\Phi \vdash a \equiv^{\ell}_{\ell_0} b}$ *(Guarded indistinguishability)*

$$\textsc{CGE-Leq}\ \frac{\ell_0 \le \ell \qquad \Phi \vdash a \equiv^{\ell} b}{\Phi \vdash a \equiv^{\ell}_{\ell_0} b} \qquad \textsc{CGE-NLeq}\ \frac{\ell_0 \not\le \ell}{\Phi \vdash a \equiv^{\ell}_{\ell_0} b}$$

Fig. 4. Equality rules (selected)

case that $\Phi \vdash a \equiv^{\mathsf{H}} b$. By raising ℓ, the equivalence becomes finer, as more subterms are observable. As a result, given an arbitrary derivation of $\Phi \vdash a \equiv^{\ell} b$, we can neither raise nor lower ℓ.

Hence the label ℓ in indistinguishability not only controls how fine-grained the equality is but also records the usage of variables in the two terms being equated, enforcing an untyped version of dependency tracking. The coupling of the two purposes within the same label may appear restrictive as we cannot easily raise or lower the labels, but it is necessary to ensure that DCOI is type sound. If we remove the restriction imposed by rule GE-Var, we could end up converting

between two arbitrary types A and B through this chain of equalities:

$$A \equiv^{\mathsf{L}} (\lambda x{:}^{\mathsf{H}} \star.\, x)\ A^{\mathsf{H}} \equiv^{\mathsf{L}} (\lambda x{:}^{\mathsf{H}} \star.\, x)\ B^{\mathsf{H}} \equiv^{\mathsf{H}} B$$

Without the inequality constraint in rule GE-Var, the lambda term $\lambda x{:}^{\mathsf{H}} \star.\, x$ would be allowed to appear in the first equality labeled at L. With the observer level set to L, we can derive the second equality without having to show that A and B are equal. By instantiating A and B to two types with distinct head forms, we could easily derive a program that crashes in DCOI.

3.4 Indexed Equality Type

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Typing: equalities)*

T-CEq

$$\frac{\Gamma \vdash a :^{\ell_1} A \qquad \Gamma \vdash b :^{\ell_1} A \qquad \Gamma \vdash A :^{\ell_2} s_1 \qquad (s_1, s_2) \in \mathcal{R}_{\sim} \qquad \ell_1 \le \ell_0 \qquad \ell_0 \le \ell}{\Gamma \vdash (a \sim^{\ell_0} b \in^{\ell_1} A) :^{\ell} s_2}$$

T-Refl

$$\frac{\Gamma \vdash (a \sim^{\ell_0} a \in^{\ell_1} A) :^{\ell_2} s}{\Gamma \vdash \mathbf{refl} :^{\ell} (a \sim^{\ell_0} a \in^{\ell_1} A)}$$

T-Transp

$$\frac{\Gamma, x{:}^{\ell_1} A \vdash B :^{\ell_0} s \qquad \Gamma \vdash a :^{\ell} (a_0 \sim^{\ell_0} a_1 \in^{\ell_1} A) \qquad \Gamma \vdash b :^{\ell} B\{a_0/x\}}{\Gamma \vdash \mathbf{transp}\ a\ b :^{\ell} B\{a_1/x\}}$$

Fig. 5. Typing rules for indexed equalities

The indexed equality type internalizes the indistinguishability judgment so programmers can reason about indistinguishability within DCOI itself. The typing and equality rules related to the equality type can be found in Figure 5 and 6. Ignoring the levels, the rules for the indexed equality rules in DCOI closely correspond to the rules for identity types from MLTT [Martin-Löf 1975].

An indexed equality type takes the form $a \sim^{\ell_0} b \in^{\ell_1} A$ and is well formed when it satisfies the conditions specified in rule T-CEq. The binary relation $\mathcal{R}_{\sim}$ is analogous to $\mathcal{R}_{\Pi}$ for function types and determines the sort we can assign to an equality type. Both a and b must have type A with ℓ_1 as their observer level. The level ℓ_0 indicates the observer level at which a and b are compared and can be distinct from ℓ_1, though it must satisfy the constraint that $\ell_1 \le \ell_0$. The $\ell_1 \le \ell_0$ constraint will be explained when we introduce the elimination form for the equality type. We defer the discussion of the second constraint $\ell_0 \le \ell$ to Section 5 since its explanation requires some background on the metatheoretic properties of DCOI. Similar to the introduction rule of the identity type from MLTT, a trivial **refl** proof witnesses the reflexive indistinguishability through rule T-Refl, provided that the equality type is well formed.

Rule T-Transp allows us to eliminate an equality proof based on the (specialized) congruence property of indistinguishability: for $\ell_0 \le \ell$, if $\Phi, x :\ \ell_0 \vdash B_0 \equiv^{\ell} B_1$ and $\Phi \vdash a_0 \equiv^{\ell} a_1$, then $\Phi \vdash B_0\{a_0/x\} \equiv^{\ell} B_1\{a_1/x\}$. Given a motive B and an equality proof a between the terms a_0 and a_1, rule T-Transp can convert a term b of type $B\{a_0/x\}$ to a term of type $B\{a_1/x\}$. The observer level ℓ_0 from the equality type matches the observer level of the motive B, whereas the level ℓ_1 matches the level associated with the variable x.

The congruence property of indistinguishability requires us carefully to constrain the observer levels so we do not accidentally convert between incompatible types. If B_0 and B_1 are indistinguishable at level ℓ, then we cannot substitute in a_0 and a_1 if they are indistinguishable at level ℓ_0 such that $\ell_0 < \ell$. To see why this is problematic, consider the following two judgments.

$$x{:}\,\mathsf{L} \vdash \mathbf{unbox}^{\mathsf{H}}\ x \equiv^{\mathsf{H}} \mathbf{unbox}^{\mathsf{H}}\ x \qquad \cdot \vdash \mathbf{box}^{\mathsf{H}}\ \mathbf{false} \equiv^{\mathsf{L}} \mathbf{box}^{\mathsf{H}}\ \mathbf{true}$$

The first judgment is true because both sides are identical. The second judgment is true since **false** and **true** both appear in high-level boxes and are thus indistinguishable for a low-level observer. If we were allowed to substitute the second equation for x in the first one, we would be able to derive $\cdot \vdash \mathbf{unbox}^{\mathsf{H}}\ (\mathbf{box}^{\mathsf{H}}\ \mathbf{false}) \equiv^{\mathsf{H}} \mathbf{unbox}^{\mathsf{H}}\ (\mathbf{box}^{\mathsf{H}}\ \mathbf{true})$. By reducing each side, we end up proving that $\cdot \vdash \mathbf{false} \equiv^{\mathsf{H}} \mathbf{true}$. In general, if B_0 and B_1 are indistinguishable at level ℓ, then we want to only substitute in an indistinguishability judgment that is not too coarse for the level ℓ.

Finally, given an equality type $a \sim^{\ell_0} b \in^{\ell_1} A$, due to the constraint $\ell_1 \le \ell_0$ in rule T-CEq, rule T-Transp can only be applied when $\ell_1 \le \ell_0$, so we are only allowed to eliminate an indexed equality if the level ℓ_1 of the term we are substituting is bounded above by the level of the motive. This constraint does not limit the expressiveness of DCOI but is convenient for our proofs. When $\ell_1 \not\le \ell_0$, the variable x can only appear in B in contexts, such as the argument to a function, where the level is high enough to use x. In that case, $B\{a_0/x\}$ and $B\{a_1/x\}$ are indistinguishable at level ℓ_0 by equating the subterms containing a_0 and a_1 through rule CGE-NLeq from guarded indistinguishability. We can then assign type $B\{a_1/x\}$ to a by rule T-Conv directly without using rule T-Transp. We make this intuition formal in Section 4 through the full congruence property of indistinguishability (Lemma 4.5).

$\boxed{\Phi \vdash a \equiv^{\ell} b}$ *(Indistinguishability: equalities)*

GE-CEq
$$\frac{\Phi \vdash a_0 \equiv^{\ell} a_1 \qquad \Phi \vdash b_0 \equiv^{\ell} b_1 \qquad \ell_0 \le \ell}{\Phi \vdash a_0 \sim^{\ell_0} b_0 \in^{\ell_1} A_0 \equiv^{\ell} a_1 \sim^{\ell_0} b_1 \in^{\ell_1} A_1}$$

GE-TranspRefl
$$\frac{\Phi \vdash a_0 \equiv^{\ell} \mathbf{refl} \qquad \Phi \vdash b_0 \equiv^{\ell} b_1}{\Phi \vdash \mathbf{transp}\ a_0\ b_0 \equiv^{\ell} b_1}$$

GE-Transp
$$\frac{\Phi \vdash a_0 \equiv^{\ell} a_1 \qquad \Phi \vdash b_0 \equiv^{\ell} b_1}{\Phi \vdash \mathbf{transp}\ a_0\ b_0 \equiv^{\ell} \mathbf{transp}\ a_1\ b_1}$$

GE-Refl
$$\frac{}{\Phi \vdash \mathbf{refl} \equiv^{\ell} \mathbf{refl}}$$

Fig. 6. Equality rules for indexed equalities

Figure 6 shows the indistinguishability rules for indexed equality. When used in conjunction with rule GE-CEq, rule T-Refl allows us to use the **refl** constructor to witness the equality between the two terms a and b at observer level ℓ if we can show that $\Phi \vdash a \equiv^{\ell} b$. This is analogous to how the reflexivity proof term can witness the equality between two β-equivalent terms in Martin-Löf type theory. In rule GE-CEq, the precondition requires that the observer level ℓ_0 of the equality type be lower than the observer level ℓ of the judgment. Moreover, the observer level used for the equalities in the preconditions is ℓ rather than ℓ_0 even though ℓ_0 is the one labeled on the equality types. We will explain the subtleties behind the choice of labels in rule GE-CEq in Section 5.2.

4 METATHEORY

In this section, we present our main results, including type soundness (Theorem 4.24 and 4.29) and noninterference (Theorem 4.31), with pointers to the corresponding Coq proof file and name in the footnote. To formulate these properties, we first define the operational semantics and the syntax of values in Figure 7. This figure includes the β-rules only—the compatibility rules are standard for a call-by-name language. The full reduction rules can be found in the supplementary materials[8].

[8] spec.pdf

$\boxed{a \rightsquigarrow b}$ *(Reduction)*

R-AppAbs
$$\frac{}{(\lambda x{:}^{\ell_0} A.\ a)\ b^{\ell_0} \rightsquigarrow a\{b/x\}}$$

R-TranspRefl
$$\frac{}{\textbf{transp refl}\ a \rightsquigarrow a}$$

R-Proj1Beta
$$\frac{}{\pi_1^{\ell}\ ({a_1}^{\ell}, a_2) \rightsquigarrow a_1}$$

R-Proj2Beta
$$\frac{}{\pi_2^{\ell}\ ({a_1}^{\ell}, a_2) \rightsquigarrow a_2}$$

R-IfTrue
$$\frac{}{\textbf{if true}\ b_0\ b_1 \rightsquigarrow b_0}$$

R-IfFalse
$$\frac{}{\textbf{if false}\ b_0\ b_1 \rightsquigarrow b_1}$$

$$v ::= s \mid \textbf{Unit} \mid \textbf{unit} \mid \Pi x{:}^{\ell} A.\ B \mid \lambda x{:}^{\ell} A.\ a \mid \Sigma x{:}^{\ell} A.\ B \mid (a^{\ell}, b) \mid a \sim^{\ell} b \in^{\ell_0} A \mid \textbf{refl} \mid \mathbb{B} \mid \textbf{true} \mid \textbf{false}$$

Fig. 7. Reduction (β-rules only) and values

4.1 Structural Lemmas

The system supports the standard structural rules such as weakening and narrowing. Since rule T-Conv depends on the definitional equality judgment, we need to first establish the structural lemmas for definitional equality and indistinguishability before we can derive the structural lemmas for the typing judgment.

Lemma 4.1 (Indistinguishability Weakening[9]). *If* $\Phi \vdash a \equiv^{\ell} b$, *then* $\Phi, \Phi_0 \vdash a \equiv^{\ell} b$.

Lemma 4.2 (DefEq weakening[10]). *If* $\Phi \vdash a \equiv b$, *then* $\Phi, \Phi_0 \vdash a \equiv b$.

In both weakening lemmas, we use Φ, Φ_0 to indicate the concatenation of two erased contexts with disjoint and unique domains.

The narrowing property says that equalities still hold after lowering the levels of the variables in the context.

Lemma 4.3 (Indistinguishability Narrowing[11]). *If* $\Phi \vdash A \equiv^{\ell} B$ *and* $\Phi_0 \leq \Phi$, *then* $\Phi_0 \vdash A \equiv^{\ell} B$.

Lemma 4.4 (DefEq Narrowing[12]). *If* $\Phi \vdash A \equiv B$ *and* $\Phi_0 \leq \Phi$, *then* $\Phi_0 \vdash A \equiv B$.

The relation $\Phi_0 \leq \Phi$ compares pointwise the levels of two erased contexts with the same domain.

Unlike for the weakening and narrowing properties, indistinguishability and definitional equality behave differently when it comes to congruence. The congruence lemma for indistinguishability is defined as follows.

Lemma 4.5 (GDefEq Congruence[13]). *If* $\Phi, x : \ell_0 \vdash a_0 \equiv^{\ell} a_1$ *and* $\Phi \vdash b_0 \equiv^{\ell}_{\ell_0} b_1$, *then* $\Phi \vdash a_0\{b_0/x\} \equiv^{\ell} a_1\{b_1/x\}$.

It is crucial that the label ℓ from the guarded indistinguishability judgment matches the observer level from the indistinguishability judgment. The guarded indistinguishability we are substituting in must be as fine as the level of the indistinguishability judgment or we may equate two terms with differing head forms. Given $\Phi, x : \ell_0 \vdash a_0 \equiv a_1$, its derivation may consist of a sequence of indistinguishability judgments with different observer levels. As a result, we cannot formulate a congruence property for definitional equality in a manner similar to Lemma 4.5.

Lemma 4.6 (DefEq Congruence[14]). *If* $\Phi, x : \ell_0 \vdash a_0 \equiv a_1, \Phi \vdash b_0 \equiv^{\ell_0} b_1$, *and* b_0 *is* β*-equivalent to* b_1, *then* $\Phi \vdash a_0\{b_0/x\} \equiv a_1\{b_1/x\}$.

[9] defeq_weak.v:gdefeq_weak_nil [10] defeq_weak.v:defeq_weak_nil [11] narrow.v:gdefeq_narrow
[12] narrow.v:defeq_narrow [13] defeq_subst.v:gdefeq_cong [14] defeq_subst.v:defeq_cong

Parallel reductions

$\Phi \vdash a \Rightarrow_\ell b$	*Indexed parallel reduction*	$\mapsto$	$\Phi \vdash a \equiv^\ell b$
$\Phi \vdash^\ell_{\ell_0} a \Rightarrow b$	*Guarded parallel reduction*	$\mapsto$	$\Phi \vdash a \equiv^\ell_{\ell_0} b$
$\Phi \vdash a \Rightarrow b$	*Parallel reduction*	$\mapsto$	$\Phi \vdash a \equiv b$

Fig. 8. Summary of the parallel reduction judgments

Since we do not know which observer levels are used underneath the derivation of $\Phi, x\colon \ell_0 \vdash a_0 \equiv a_1$, we can only make the worst-case assumption that we cannot skip any comparison between the subterms in b_0 and b_1. That is why Lemma 4.6 additionally requires that b_0 and b_1 are β-equivalent.

Indistinguishability does not include reflexivity as an axiom, but we can prove that reflexivity holds for well-typed terms.

Lemma 4.7 (Typing Indistinguishability[15]). *If* $\Gamma \vdash a :^\ell A$, *then* $|\Gamma| \vdash a \equiv^\ell a$.

Now we are ready to prove some properties about the typing judgments. The proofs follow the same structure as their equality counterparts by induction over the derivation, though they rely on the structural rules about equalities proven earlier.

Lemma 4.8 (Narrowing[16]). *If* $\Gamma \vdash a :^\ell A$ *and* $\Gamma_0 \leq \Gamma$, *then* $\Gamma_0 \vdash a :^\ell A$.

Subsumption (Lemma 3.1) and weakening both follow from Lemma 4.8.

Lemma 4.9 (Weakening[17]). *If* $\Gamma_1 \vdash a :^\ell A$ *and* $\vdash \Gamma_1, \Gamma_2$, *then* $\Gamma_1, \Gamma_2 \vdash a :^\ell A$.

We can now prove the substitution property for the typing judgment through structural induction over derivations.

Lemma 4.10 (Substitution[18]). *If* $\Gamma, x{:}^{\ell_0} A \vdash b :^\ell B$ *and* $\Gamma \vdash a :^{\ell_0} A$, *then* $\Gamma \vdash b\{a/x\} :^\ell B\{a/x\}$.

The rule T-Var case of Lemma 4.10 relies on Lemma 3.1 since we may substitute in a term with a lower level when $\ell_0 \leq \ell$. The rule T-Conv case requires us to show that $|\Gamma| \vdash A\{a/x\} \equiv B\{a/x\}$ given $|\Gamma|, x\colon \ell_0 \vdash A \equiv B$ and $\Gamma \vdash a :^{\ell_0} A$. This can be done by composing Lemma 4.6 and Lemma 4.7.

4.2 Preservation

Similar to the preservation proof for PTS, preservation in DCOI requires inversion lemmas about the typing judgment and injectivity lemmas about the definitional equality judgment.

The inversion lemmas witness the fact that given $\Gamma \vdash v :^\ell A$, the derivation must consist of the introduction rule for v nested under zero or more instances of the conversion rule. The proofs for those lemmas, with the exception of the **refl** form, can be carried out by induction over the derivation. For now, we only show the inversion lemma for abstractions and defer the explanation of the **refl** case until we discuss the injectivity properties; the rest of the inversion lemmas follow the same pattern and are thus omitted.

Lemma 4.11 (Inversion (Abs)[19]). *If* $\Gamma \vdash (\lambda x{:}^{\ell_0} A.\, b) :^\ell A_0$, *then there exists a term* B *such that the following conditions hold:*

- $\Gamma, x{:}^{\ell_0} A \vdash b :^\ell B$
- $|\Gamma| \vdash \Pi x{:}^{\ell_0} A.\, B \equiv A_0$
- $\Gamma \vdash A_0 :^{\ell_1} s$ *for some level* ℓ_1 *and some sort* s

[15] typing_defeq.v:typing_gdefeq [16] narrow.v:typing_narrow [17] weak.v:typing_weak_nil
[18] subst.v:typing_subst_nil [19] inv.v:typing_abs_inv

To prove preservation for β-rules such as rules R-AppAbs and R-TranspRefl (Figure 7), we need to prove some injectivity lemmas about definitional equality. For example, to prove that rule R-AppAbs is type preserving, we need to show that the two definitionally equal function types must also have definitionally equal argument types and return types. A similar proof can be found in the preservation proof for PTS, where definitional equality is defined as β-equivalence. To prove the injection lemma for function types in a PTS, one needs to first prove the Church-Rosser property, which states that two terms are β-equivalent if and only if they can strongly reduce to the same term within a finite number of steps. The injection lemma then immediately follows since a reduction sequence starting from a function type can only consist of repeated reductions of its argument type or return type.

We adapt the proof technique and prove the equivalence between the equality judgments and their corresponding parallel reduction relations, summarized in Figure 8. Each reduction relation corresponds to the equivalence relation in the following sense: given two terms a_0 and a_1, the terms a_0 and a_1 are equivalent if and only if there exists some term b such that a_0 and a_1 both reduce to b after a finite number of steps. We omit the rules for the parallel reduction relations since they closely correspond to the equality rules in Figure 4. The rules for each parallel reduction relation can be obtained from the corresponding equivalence rule by removing the transitivity and symmetry rules and replacing $\equiv$ with $\Rightarrow$. These parallel reduction rules can be found in the supplementary materials[20]. We also omit the structural lemmas for the reduction relations since their specifications and proofs can be obtained with a similar replacement.

Similar to the indistinguishability judgment, given $\Phi \vdash a \Rightarrow_\ell b$, neither a nor b is allowed to use variables from Φ that are not observable at level ℓ. The following shows that if $\Phi \vdash a \Rightarrow_\ell b$ or $\Phi \vdash a \equiv^\ell b$, then indexed parallel reduction is reflexive on both a and b.

Lemma 4.12 (Par Reflexive[21]). *If $\Phi \vdash a \Rightarrow_\ell b$, then $\Phi \vdash a \Rightarrow_\ell a$ and $\Phi \vdash b \Rightarrow_\ell b$.*

Lemma 4.13 (Indistinguishability Par Reflexive[22]). *If $\Phi \vdash a \equiv^\ell b$, then $\Phi \vdash a \Rightarrow_\ell a$ and $\Phi \vdash b \Rightarrow_\ell b$.*

Both lemmas follow from induction on the derivation of the premise.

Indexed parallel reduction generalizes β-reduction in the same way that indistinguishability generalizes β-equivalence. It is possible to embed the β-reduction relation into the parallel reduction relation.

Lemma 4.14 (Red Embed Par[23]). *If $a \rightsquigarrow b$ and there exists some b_0 such that $\Phi \vdash a \Rightarrow_\ell b_0$, then $\Phi \vdash a \Rightarrow_\ell b$.*

In Lemma 4.14, the precondition $\Phi \vdash a \Rightarrow_\ell b_0$ tells us that a does not use variables with labels not observable at ℓ. It provides us a context and a label at which we can reduce from a to b through indexed parallel reduction. The precondition $\Phi \vdash a \Rightarrow_\ell b_0$ can be supplied through the well-typedness of a by composing Lemma 4.7 and Lemma 4.13.

From the structural lemmas, we can show that indexed parallel reduction satisfies the following confluence property.

Lemma 4.15 (Indexed Par Confluence[24]). *If $\Phi \vdash a \Rightarrow_\ell b_0$ and $\Phi \vdash a \Rightarrow_\ell b_1$, then there exists some b_2 such that $\Phi \vdash b_0 \Rightarrow_\ell b_2$ and $\Phi \vdash b_1 \Rightarrow_\ell b_2$.*

Before we can use Lemma 4.15 to derive the confluence property of parallel reduction, we need to first show the following downgrade property about indexed parallel reduction.

[20] spec.pdf [21] par.v:Par_grade_mutual [22] defeq_proj.v:gdefeq_grade [23] preservation.v:red_embed
[24] par.v:Par_confluence

Lemma 4.16 (Indexed Par Downgrade[25]). *If* $\Phi \vdash a \Rightarrow_{\ell} b_0$ *and* $\Phi \vdash a \Rightarrow_{\ell_0} b_1$, *then* $\Phi \vdash a \Rightarrow_{\ell \wedge \ell_0} b_0$.

The term b_1 does not appear in the conclusion. The only information we need from the premise $\Phi \vdash a \Rightarrow_{\ell_0} b_1$ is that a has ℓ_0 as its observer level. The proof proceeds by induction over the derivation and leverages the fact that $\ell \wedge \ell_0$ is the greatest lower bound.

Confluence of parallel reduction then follows as a corollary of Lemmas 4.15 and 4.16.

Lemma 4.17 (Par Confluent[26]). *If* $\Phi \vdash a \Rightarrow b_0$ *and* $\Phi \vdash a \Rightarrow b_1$, *then there exists some* b_2 *such that* $\Phi \vdash b_0 \Rightarrow b_2$ *and* $\Phi \vdash b_1 \Rightarrow b_2$.

The proof proceeds by unfolding the definition of parallel reduction. This gives us $\Phi \vdash a \Rightarrow_{\ell} b_0$ and $\Phi \vdash a \Rightarrow_{\ell_0} b_1$ for some ℓ and ℓ_0. By applying Lemma 4.16 twice, we can conclude that $\Phi \vdash a \Rightarrow_{\ell \wedge \ell_0} b_0$ and $\Phi \vdash a \Rightarrow_{\ell \wedge \ell_0} b_1$. The conclusion then follows immediately by applying Lemma 4.15.

Once Lemma 4.17 is proven, we can easily show that two terms are definitionally equal if and only if they parallel reduce to the same term.

Lemma 4.18 (DefEq Join[27]). $\Phi \vdash a_0 \equiv a_1$ *if and only if there exists some* b *such that* $\Phi \vdash a_0 \Rightarrow^* b$ *and* $\Phi \vdash a_1 \Rightarrow^* b$.

The backward direction is trivial since it is possible to embed parallel reduction into definitional equality by simply unfolding definitions. The forward direction uses Lemma 4.17 to finish the rule E-Trans case and the rest of the cases are straightforward.

The injectivity lemmas for the function types and dependent sum types follow as corollaries of Lemma 4.18.

Lemma 4.19 (DefEq Π-Inj1[28]). *If* $\Phi \vdash \Pi x{:}^{\ell_0} A_0.\, B_0 \equiv \Pi x{:}^{\ell_0} A_1.\, B_1$, *then* $\Phi \vdash A_0 \equiv A_1$.

Lemma 4.20 (DefEq Π-Inj2[29]). *If* $\Phi \vdash \Pi x{:}^{\ell_0} A_0.\, B_0 \equiv \Pi x{:}^{\ell_0} A_1.\, B_1$, *then* $\Phi, x : \ell_0 \vdash B_0 \equiv B_1$.

Lemma 4.21 (DefEq Σ-Inj1[30]). *If* $\Phi \vdash \Sigma x{:}^{\ell_0} A_0.\, B_0 \equiv \Sigma x{:}^{\ell_0} A_1.\, B_1$, *then* $\Phi \vdash A_0 \equiv A_1$.

Lemma 4.22 (DefEq Σ-Inj2[31]). *If* $\Phi \vdash \Sigma x{:}^{\ell_0} A_0.\, B_0 \equiv \Sigma x{:}^{\ell_0} A_1.\, B_1$, *then* $\Phi, x : \ell_0 \vdash B_0 \equiv B_1$.

Furthermore, Lemma 4.18 allows us to show the following strengthened inversion lemma about the indexed equality type.

Lemma 4.23 (Refl GDefEq[32]). *If* $\Gamma \vdash \mathbf{refl} :^{\ell} a \sim^{\ell_0} b \in^{\ell_1} A$, *then* $|\Gamma| \vdash a \equiv^{\ell_0} b$.

In the proof of Lemma 4.23, Lemma 4.18 is used to show that given $|\Gamma| \vdash a_0 \sim^{\ell_0} b_0 \in^{\ell_1} A \equiv a \sim^{\ell_0} b \in^{\ell_1} A$, we can conclude that $|\Gamma| \vdash a_0 \equiv^{\ell_0} a$ and $|\Gamma| \vdash b_0 \equiv^{\ell_0} b$.

Now, we can finally show that the reduction relation is type preserving.

Theorem 4.24 (Preservation[33]). *If* $\Gamma \vdash a :^{\ell} A$ *and* $a \rightsquigarrow b$, *then* $\Gamma \vdash b :^{\ell} A$.

The proof of Theorem 4.24 is carried out by structural induction over the reduction relation. The inversion lemmas are applied to the premise $\Gamma \vdash a :^{\ell} A$ to gain information about the derivation. The injectivity lemmas are needed for the cases that involve β-rules. The case for the second projection needs some special treatment. As part of the proof, given $\Gamma \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.\, B$ and $a \rightsquigarrow a_0$, we need to prove that $\pi_2^{\ell_0}\ a_0$ can be assigned the type $B\{\pi_1^{\ell_0}\ a/x\}$. From rule T-Proj2 and the inductive hypothesis, we can show that $\Gamma \vdash \pi_2^{\ell_0}\ a_0 :^{\ell} B\{\pi_1^{\ell_0}\ a_0/x\}$. To finish off the proof, we use Lemma 4.14 to inject the reduction into the indistinguishability relation to show that $B\{\pi_1^{\ell_0}\ a/x\}$ is definitionally equal to $B\{\pi_1^{\ell_0}\ a_0/x\}$.

[25] par.v:Par_downgrade_mutual
[26] par.v:EPar_confluence
[27] defeq_proj.v:defeq_ejoins_iff
[28] defeq_proj.v:defeq_pi_proj1
[29] defeq_proj.v:defeq_pi_proj2
[30] defeq_proj.v:defeq_sigma_proj
[31] defeq_proj.v:defeq_sigma_proj
[32] preservation.v:typing_refl_gdefeq
[33] preservation.v:preservation

4.3 Progress

Before we prove progress, it is useful to derive the following simple corollary of Lemma 4.18.

Lemma 4.25 (DefEq Consistency[34]). *If* $\Phi \vdash v_0 \equiv v_1$, *then* v_0 *and* v_1 *have the same head form.*

By Lemma 4.18, v_0 and v_1 can reduce to the same term through parallel reduction. However, since parallel reduction preserves head forms, v_0 and v_1 cannot have conflicting head forms or it would be impossible to reduce them to the same term.

Lemma 4.25, when combined with the inversion lemmas from Section 4.2, can be used to prove the following canonical form lemmas.

Lemma 4.26 (Π-Canon[35]). *If* $\Gamma \vdash v :^{\ell} \Pi x :^{\ell_0} A.\, B$, *then there exists some* A_0 *and* a_0 *such that* $v = \lambda x :^{\ell_0} A_0.\, a_0$.

Lemma 4.27 (Σ-Canon[36]). *If* $\Gamma \vdash v :^{\ell} \Sigma x :^{\ell_0} A_0.\, B_0$, *then there exists some* a_0 *and* a_1 *such that* $v = (a_0 {}^{\ell_0}, a_1)$.

Lemma 4.28 (Eq-Canon[37]). *If* $\Gamma \vdash v :^{\ell} b_0 \sim^{\ell_0} b_1 \in^{\ell_1} A$, *then* $v = \mathbf{refl}$.

We can now conclude our type soundness proof by showing the progress theorem.

Theorem 4.29 (Progress[38]). *If* $\Gamma \vdash a :^{\ell} A$, *then* a *is a value or there exists some* b *such that* $a \rightsquigarrow b$.

The proof proceeds by induction over the derivation $\Gamma \vdash a :^{\ell} A$. Each non-value case can be discharged with the help of its corresponding canonical form lemma.

Theorem 4.24 and 4.29 complete our type soundness result. A simple corollary of the type soundness result is the erasability of the levels that appear in the term syntax. For example, the level ℓ_0 in the application form $a\ b^{\ell_0}$ can block rule R-AppAbs if it mismatches the level of the lambda term. Preservation and progress imply that a well-typed term's execution will never be blocked by mismatching levels. Since blocking β-rules is the only way the levels can affect reduction, we can safely erase the levels before running a well-typed term. However, levels are still necessary for definitional equality and indistinguishability because those relations are untyped. Refactoring to use a typed indistinguishability relation might allow us to remove the levels from the term syntax completely, and we leave that as part of our future work.

4.4 Noninterference

In DCOI, one can formulate the following simulation property in terms of indistinguishability. It states that programs that are indistugishable remain indistinguishable during evaluation.

Lemma 4.30 (Simulation[39]). *If* $\Phi \vdash a_0 \equiv^{\ell} b_0$, $a_0 \rightsquigarrow^* a_1$, *and* $b_0 \rightsquigarrow^* b_1$, *then* $\Phi \vdash a_1 \equiv^{\ell} b_1$.

Theorem 4.30 is easily derivable from Lemma 4.14 and Lemma 4.12. The conclusion of Lemma 4.30 does not immediately tell us much about the relation between a_1 and b_1 since indistinguishability incorporates both reduction and irrelevance.

However, when combined with Lemma 4.18, Lemma 4.30 imposes very strong constraints on how two indistinguishable programs may behave after evaluation. With the boolean type, we formulate the following noninterference theorem, which states that two indistinguishable boolean computations reduce to the same boolean value.

Theorem 4.31 (Noninterference[40]). *If* $\cdot \vdash a \equiv^{\ell} b$, $\cdot \vdash a :^{\ell} \mathbb{B}$, $\cdot \vdash b :^{\ell} \mathbb{B}$, $a \rightsquigarrow^* v_0$, *and* $b \rightsquigarrow^* v_1$, *then* $v_0 = v_1$.

[34] defeq_par.v:defeq_consist [35] progress.v:typing_pi_canon [36] progress.v:typing_ssigma_canon
[37] progress.v:typing_ceq_canon [38] progress.v:progress [39] preservation.v:simulation
[40] progress.v:non_interference

5 DISCUSSION

5.1 Incompatibility between Indistinguishability and the Upgrade Property

Consider these variants of rules T-App and T-Abs, with argument level $\ell \vee \ell_0$ instead of ℓ.

T-AppJoin
$$\frac{\Gamma \vdash b :^{\ell} \Pi x :^{\ell_0} A.\, B \qquad \Gamma \vdash a :^{\ell \vee \ell_0} A}{\Gamma \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}}$$

T-AbsJoin
$$\frac{\Gamma, x :^{\ell \vee \ell_0} A \vdash b :^{\ell} B \qquad \Gamma \vdash (\Pi x :^{\ell \vee \ell_0} A.\, B) :^{\ell_1} s}{\Gamma \vdash \lambda x :^{\ell_0} A.\, b :^{\ell} \Pi x :^{\ell_0} A.\, B}$$

Compared to the corresponding rules from DCOI, the alternative rules are more permissive. The following judgment is derivable with the alternative rules but not the existing ones.

$$y :^{\mathsf{H}} \mathbb{B} \vdash (\lambda x :^{\mathsf{L}} \mathbb{B}.\, x)\ y^{\mathsf{L}} :^{\mathsf{H}} \mathbb{B}$$

With rule T-App, the application is illegal because the argument y is type checked at L but y can only be used when the observer level is H. However, with rule T-AppJoin, the level for type checking the argument is the join of the observer level H and the annotation L, so the example holds.

The variant of rule T-App and rule T-Abs that raises the argument level to be at least high as the observer level can be found in systems that track either dependency or usage, such as QTT [Atkey 2018] (after instantiating to the boolean semiring) and extensions of DCC [Abadi et al. 1999], including DCC$^{\text{pc}}$ [Tse and Zdancewic 2004], the sealing calculus [Shikuma and Igarashi 2008], and DDC [Choudhury et al. 2022]. Notably, the alternative rules still satisfy noninterference.

In earlier iterations of DCOI, we opted for rule T-AppJoin and rule T-AbsJoin. However, we found that these rules were incompatible with the use of indistinguishability for conversion, since the system no longer admitted the subsumption property. Consider the following function f that type checks with both the alternative and existing rules.

$$\cdot \vdash \lambda x :^{\mathsf{L}} (\mathbb{B}^{\mathsf{H}} \to \star).\, \lambda y :^{\mathsf{L}} ((x\ \mathbf{true}^{\mathsf{H}})^{\mathsf{L}} \to \mathbb{B}).\, \lambda z :^{\mathsf{L}} (x\ \mathbf{false}^{\mathsf{H}}).\, y\ z^{\mathsf{L}} :^{\mathsf{L}} \mathbb{B}$$

In the example above, since $x\ \mathbf{false}^{\mathsf{H}}$ and $x\ \mathbf{true}^{\mathsf{H}}$ are indistinguishable at level L, we are allowed to pass z, a term with type $x\ \mathbf{false}^{\mathsf{H}}$, as an argument to y, a function which expects something with type $x\ \mathbf{true}^{\mathsf{H}}$.

If subsumption holds, we can type check f at level H. However, we run into a problem when we try to apply f using rule T-AppJoin: the first argument for f is type checked at $\mathsf{H} \vee \mathsf{L}$, where H is the observer level of f after applying subsumption and L is the annotation from the lambda. That is, the alternative type system would accept any term a as an argument for f as long as $\cdot \vdash a :^{\mathsf{H}} \mathbb{B}^{\mathsf{H}} \to \star$. But since a is typed at H, it can produce different types by pattern matching against its input. By instantiating a properly, f would allow us to convert between two arbitrary types! On the other hand, with our existing rules, we enforce that a is well typed at L regardless of the observer level, so a will always behave like a constant function even after applying subsumption.

The above contradiction can also be phrased in terms of the incompatibility between indistinguishability and the following, which we informally refer to as the upgrade property.

Proposition 5.1 (Upgrade property, does not hold for DCOI). *If $\Gamma \vdash b :^{\ell} A$, then $\ell \vee \Gamma \vdash b :^{\ell} A$, where $\ell \vee \Gamma$ is an operation that joins the levels in Γ pointwise against the level ℓ.*

The proposition states that raising the levels in the context to the observer level does not make the term b more difficult to type check. Suppose we have $x :^{\mathsf{L}} \mathbb{B}^{\mathsf{H}} \to \star$ in the typing context Γ. Then, the derivation of b could convert between $x\ \mathbf{false}^{\mathsf{H}}$ and $x\ \mathbf{true}^{\mathsf{H}}$ via indistinguishability. By applying the upgrade property, however, we end up with $x :^{\mathsf{H}} \mathbb{B}^{\mathsf{H}} \to \star$ and x can no longer be used at observer level L in that derivation. As a result, the upgrade property does not hold in DCOI.

In both QTT and DDC, we can find lemmas that capture the same idea as the upgrade property. In QTT, the upgrade property holds when its semiring structure is instantiated to $\{0, 1\}$, the boolean

semiring. In DDC, the upgrade property is more interesting. DDC is parameterized by some lattice structure $\mathcal{L}$ and additionally adds a two-point lattice $\{C, \top\}$ on top of $\mathcal{L}$ such that $\ell < C < \top$ for all $\ell \in \mathcal{L}$. Like DCOI, DDC uses indistinguishability as part of its conversion rule; however, the observer level for indistinguishability in the conversion rule is always C. In fact, DDC treats the level $\{C, \top\}$ differently compared to the levels from the lattice $\mathcal{L}$.

The upgrade property in DDC only holds when the level ℓ is less than or equal to C. The disjoint treatment between $\{C, \top\}$ and $\mathcal{L}$ in DDC is consistent with our finding: since the lower levels satisfy the upgrade property, indistinguishability can no longer be used for conversion at those levels. In contrast, by abandoning rules T-AppJoin and T-AbsJoin and therefore the upgrade property, DCOI can use indistinguishability for conversion at any level in the lattice.

5.2 Order Constraints for Indexed Equality Types

We explain the necessity of the ordering constraint $\ell_0 \le \ell$ that appears in rule GE-CEq.

$$\text{GE-CEq}\quad \frac{\Phi \vdash a_0 \equiv^{\ell} a_1 \qquad \Phi \vdash b_0 \equiv^{\ell} b_1 \qquad \ell_0 \le \ell}{\Phi \vdash a_0 \sim^{\ell_0} b_0 \in^{\ell_1} A_0 \equiv^{\ell} a_1 \sim^{\ell_0} b_1 \in^{\ell_1} A_1}$$

In rule GE-CEq, to show that the two equality types $a_0 \sim^{\ell_0} b_0 \in^{\ell_1} A_0$ and $a_1 \sim^{\ell_0} b_1 \in^{\ell_1} A_1$ are equal, the endpoints a_0, a_1 and b_0, b_1 are compared pointwise at the observer level ℓ, where ℓ is constrained to be greater than or equal to the observer level ℓ_0 of the equality type. The premise of rule GE-CEq does not require any relation between A_0 and A_1 since they are type annotations.

It is tempting to simplify rule GE-CEq into the following.

$$\text{GE-CEqWrong}\quad \frac{\Phi \vdash a_0 \equiv^{\ell_0} a_1 \qquad \Phi \vdash b_0 \equiv^{\ell_0} b_1}{\Phi \vdash a_0 \sim^{\ell_0} b_0 \in^{\ell_1} A_0 \equiv^{\ell} a_1 \sim^{\ell_0} b_1 \in^{\ell_1} A_1}$$

Compared to rule GE-CEq, this simplified version completely ignores the observer level ℓ in the premises and instead uses the level ℓ_0 to compare the subterms. The issue with this design is that it breaks the congruence property of indistinguishability (Lemma 4.5). Consider the following derivable judgment with rule GE-CEqWrong.

$$x : \mathsf{H} \vdash (x \sim^{\mathsf{H}} \mathbf{true} \in^{\mathsf{H}} \mathbb{B}) \equiv^{\mathsf{L}} (x \sim^{\mathsf{H}} \mathbf{true} \in^{\mathsf{H}} \mathbb{B})$$

Since $\mathsf{H} \not\le \mathsf{L}$, we can derive the vacuously true guarded indistinguishability $\cdot \vdash \mathbf{false} \equiv^{\mathsf{L}}_{\mathsf{H}} \mathbf{true}$. If Lemma 4.5 holds, then we can apply substitution and derive this indistinguishability judgment.

$$\cdot \vdash (\mathbf{false} \sim^{\mathsf{H}} \mathbf{true} \in^{\mathsf{H}} \mathbb{B}) \equiv^{\mathsf{L}} (\mathbf{true} \sim^{\mathsf{H}} \mathbf{true} \in^{\mathsf{H}} \mathbb{B})$$

The equality on the left is bogus, but the equality on the right is true by reflexivity. We could thus inhabit the equality type $\mathbf{false} \sim^{\mathsf{H}} \mathbf{true} \in^{\mathsf{H}} \mathbb{B}$ with **refl** by composing rule T-Refl and rule T-Conv with the indistinguishability judgment from above. By eliminating the bogus proof term, we could easily derive a crashing program.

The problem with rule GE-CEqWrong is that the two equality types have their own notion of indistinguishability based on the level ℓ_0, which is now independent of the observer level ℓ. The congruence lemma allows us to substitute in an equality that is sensible for an observer at level ℓ. However, if $\ell < \ell_0$, then the equality being substituted in is too coarse for an observer at level ℓ_0. In the counterexample above, the variable x is labeled with level H. From an L observer's perspective, **false** and **true** are indistinguishable when guarded by the level H. However, from the H observer's perspective, **false** and **true** are distinct when guarded by the level H. The mismatch between the

notion of equivalence allows us to break type soundness in a way similar to the **unbox** example from Section 3.4.

To address this problem, rule GE-CEQ restricts the observer level of the equality type to be at most the observer level of the judgment. The same constraint $\ell_0 \leq \ell$ found in rule T-CEQ is a direct consequence of the constraint in rule GE-CEQ since we need the order constraint in the typing rule to derive Lemma 4.7.

5.3 Irrelevance, Relatively

To talk about whether two terms are indistinguishable, we must choose an observer level. Terms that are indistinguishable at L might be distinguishable at H. So, irrelevance in DCOI is relative. For example, if $\cdot \vdash a :^{\mathsf{L}} \mathbb{B}^{\mathsf{H}} \to T^{\mathsf{H}}\ \mathbb{B}$, then these terms are both valid candidates for a.

$$\lambda x{:}^{\mathsf{H}}\mathbb{B}.\ \mathbf{box}^{\mathsf{H}}\ x \qquad \lambda x{:}^{\mathsf{H}}\mathbb{B}.\ \mathbf{box}^{\mathsf{H}}\ \mathbf{false}$$

The second function behaves like a constant function in the traditional sense—the input x does not appear in the body. The first function is only a constant function if we observe its results at level L.

Consider the filter function on vectors as another example, where unmarked arguments and the function itself are at level L. Because we do not know the final length of the filtered vector, we package the vector and its length together in a dependent pair where the first component is at level H, which we write as $(\mathsf{m} :^{\mathsf{H}} \mathsf{Nat}) \times \mathsf{Vec\ A\ m}$.

```
filter : (A :ᴴ Type) → (n :ᴴ Nat) → (A → Bool) → Vector A n →
         (m :ᴴ Nat) × Vector A m
filter A n f v = case v of
  Nil ⇒ (0, Nil)
  Cons n′ y ys ⇒ let zs = filter A n′ f ys in
                 if f y then (Succ (π₁ zs), Cons (π₁ zs) y (π₂ zs)) else zs
```

From the perspective of a run-time observer, the arguments A and n are both irrelevant. However, for an observer at level H, these arguments are both relevant. In the body of filter, the variable zs is a pair containing a vector of unknown length. The length index π_1 zs in the pair is irrelevant at level L; after erasure at level L, filter should have the run-time performance of the filter function for lists. On the other hand, the length index in the pair is used relevantly in the type of the second component, Vector A (π_1 zs). Because the notion of relevance is relative, we can still make sense of the filter function even though the length index is used both relevantly and irrelevantly.

5.4 Normalization Proof

In PTS, particular instantiations of the sorts, axioms, and rules lead to systems that are known to satisfy the strong normalization property. We have not proven normalization for any instance of DCOI; however, since it is based on PTS, we believe that there is a close correspondence between the normalization behavior of PTS and of DCOI. If a specific instantiation of the sorts results in a normalizing PTS, we would expect the same instantiation to result in a normalizing DCOI. In particular, we expect all instances from the lambda cube [Barendregt 1991] to satisfy the strong normalization property when reformulated as their corresponding DCOI instance.

Having a normalization proof for such instantiations of sorts is desirable to allow for relevance tracking with proof erasure. In rule T-TRANSP, the level of the equality proof must be the same as the level of the term being cast, so we run into the awkward situation where an irrelevant proof cannot be used to cast a relevant term. In a system where equality types can be inhabited by a diverging computation, the proof being eliminated must be treated as relevant since the erasure of a bogus proof can cause the resulting program to crash. However, if we know that a particular instance of

DCOI is normalizing, then we are able to justify the elimination of an irrelevant equality proof in a relevant context through the canonicity of closed equality proofs. Because of its importance, we will investigate the normalization behavior of DCOI as part of our future work.

6 IMPLEMENTATION

We have implemented a type checker for an instantiation of DCOI with the naturals and its usual total order as our lattice of labels, and a single type universe Type with type-in-type.[41] The biggest difference between DCOI and the implementation is that the implementation contains data types, which subsumes the unit type and dependent pair types. Below is a data definition for dependent pairs where the first component is fixed to level 1; we use it to discuss some design decisions for data definitions.

```
data Pair (A :⁰ Type) (B :⁰ A¹ → Type) :⁰ Type where
  MkPair :ᵏ (x :¹ A) → (B x)ᵏ → Pair A B
```

The level of type parameters A and B and of the Pair A B type are fixed to level 0, in contrast to the primitive dependent pair type of DCOI, which are implicitly level-polymorphic in the sense that they can be type checked at any level as long as the parameters can be type checked at the same level. Instead of fixing to level 0, we could assign A and B different levels, as long as the level of Pair A B is greater than or equal to the larger of these levels. This condition ensures that the parameters are always relevant: we should not to equate Pair A_1 B_1 and Pair A_2 B_2 unless A_1, A_2 and B_1, B_2 are themselves equal. To retain the subsumption property, data types can be used at any level greater than or equal to the level they are declared at, similar to how variables are checked by rule T-Var.

On the other hand, constructors are explicitly level-polymorphic: the second component of the pair is type checked at the same level variable k as the overall pair, so that when given a pair declared at a given level, we can use the second component at the same level. This minimal amount of polymorphism appears to be sufficient for practical examples. As with rule T-SPair, there is no relation between the level of pair type and k.

Pairs are eliminated by case expressions rather than projections, but projection functions can still be defined.

```
proj1 :¹ (A :⁰ Type) → (B :⁰ A¹ → Type) → (Pair A B)⁰ → A
proj1 = λA B p. case p of
  MkPair x y ⇒ x

proj2 :⁰ (A :⁰ Type) → (B :⁰ A¹ → Type) → (p :⁰ Pair A B) → B (proj1 A B p)
proj2 = λA B p. case p of
  MkPair x y ⇒ y
```

The branches of a case expression must be checked at a level greater or equal to that of the scrutinee. Intuitively, this prevents extracting relevant information out of something irrelevant. As a minor detail, in the type of the branch for a given constructor, the scrutinee is replaced by the constructor pattern, so the branch of proj2 requires a term of type B (proj1 A B (MkPair x y)), reducing to B x, which is exactly the type of y.

If we have pairs at level 0, then as expected, two pairs are indistinguishable at level 0 merely if their second components are, since their first components (at level 1) are automatically indistinguishable at this level. With internalized indistinguishability, this can be proven as a lemma.

[41] The code in this section has been stylized to match existing examples; the actual concrete syntax is introduced in `README.pi`.

```
pairEq :0 (A :0 Type) → (B :0 A1 →Type) → (p1 :0 Pair A B) → (p2 :0 Pair A B) →
          (proj2 A B p1 ∼0 proj2 A B p2) → p1 ∼0 p2
pairEq = λA B p1 p2 q. case p1 of
  MkPair x1 y1 ⇒ case p2 of
    MkPair x2 y2 ⇒ transp q refl
```

In DCOI, definitional equality is the transitive closure of indistinguishability at existentially-quantified levels; in the implementation, when converting between two types, we simply use indistinguishability at an arbitrary generated level metavariable at which both types are well typed.

The conversion checker checks guarded indistinguishability $\Phi \vdash a \equiv^{\ell}_{\ell_0} b$ by first asking the constraints whether $\ell < \ell_0$ is implied. If so, this corresponds to the rule CGE-NLeq case; otherwise, it adds $\ell_0 \leq \ell$ as a new constraint and continues on as in the rule CGE-Leq case. Conversion checking is therefore incomplete with respect to the rules, since it may turn out that $\ell < \ell_0$ was the appropriate constraint to add. Adding backtracking might make conversion complete, but it seems that with explicit level annotations, enough of the guesswork is eliminated.

To accommodate level metavariables, type checking always takes a level as an input, collecting constraints for generated level metavariables, and solves the level constraints with minimal level instantiations at the end, or fails with a type error.

7 FUTURE WORK

Level polymorphism. There are multiple forms of level polymorphism that could be added to DCOI. The simplest is prenex polymorphism which, in combination with global or let-bound definitions, can reduce code duplication. By quantifying a definition over the levels it uses, it can be instantiated at different levels as needed. We might however wish to also enforce constraints among the levels quantified, so that we have guarantees about their relative relevancies. This can be done by bounded quantification, which can only be instantiated by levels bounded above by the one or more other levels given. Adding such forms of polymorphism likely would not violate any of our metatheoretical properties, since each instantiation of a polymorphic definition with particular levels could be inlined as an ordinary DCOI term with those levels substituted in.

Inductive types. While our prototype implementation contains data types with levels, adding inductive types to DCOI with proper type indices, strict positivity checking, and eliminators, and updating the Coq development to ensure none of the metatheoretical properties are broken, remain future work. Aside from increasing expressivity, another goal of adding inductive types is to subsume the existing unit, dependent pair, and propositional equality primitive types. Although we won't know what rules for inductive types are correct until the proofs are done, we can use the implementation's type checker and the rules for the existing types to guide the design of expressive yet well-behaved inductive types. Lastly, inductive definitions should be prenex-polymorphic in all levels mentioned to be as flexible as the primitives, which are not fixed in their levels.

Staged programming[42]. A potential application of DCOI is in *staged programming* [Sheard and Nelson 1995], where different pieces of code are marked with different levels or *stages*, which can be thought of as different stages of metaprogramming. Our reduction rules could be extended by indexing with a stage and modified to proceed stage by stage, corresponding to expanding metaprograms at each stage in order.

Although the dynamic behavior does not yet correspond to staged evaluation, we can use the existing static types for staged programs. In particular, the box construct represents a way to

[42] pi/Staged.pi

manipulate code at different stages; for example, a value of type $T^1\ ((\mathsf{Nat}^1) \to \mathsf{Nat})$ is a function over naturals available at stage 1 but is manipulable as an opaque piece of object code at stage 0.

Consider the following implementation of exponentiation of naturals at stage 0 in terms of multiplication at stage 1. In other words, exponentiation resembles a macro that expands to applications of multiplication.

```
mul :¹ Nat¹ →Nat¹ → Nat
exp :⁰ Nat¹ → Nat⁰ → T¹ Nat
exp b e = case e of
  Zero ⇒ box¹ 1
  Succ e′ ⇒ box¹ (mul b (unbox¹ (exp b e′)))
```

With proper staged evaluation, when we reach stage 1, expressions at stage 0 will have been evaluated, and applications of exponentiation to a concrete exponent will have been expanded into iterated multiplication. For instance, $(\mathbf{unbox}^1\ (\mathsf{exp\ b\ 3}))$ evaluates to $(\mathsf{mul\ b\ (mul\ b\ (mul\ b\ \ 1)))}$ at stage 1.

Analogously to previous applications of dependency analysis, pieces of code from later stages are indistinguishable from code from earlier stages. Intuitively, this means that metaprograms can't tell what programs are doing, since programs can't run before the metaprograms expand.

Aside from separating programs into different stages of evaluation, levels could also separate proofs into different type theories, as is done in two-level type theory [Annenkov et al. 2023], which also has applications in staged compilation [Kovács 2022].

8 RELATED WORK

8.1 Systems with Dependency Tracking

The Dependency Core Calculus (DCC) [Abadi et al. 1999] introduces the graded model type $T^\ell\ A$ for dependency tracking. Similar to the box type from DCOI, given a term a of type $T^\ell\ A$, the content stored inside a can only be used in a context that can observe ℓ. DCC enforces information flow by restricting the type of the bind operator that inspects the content of a. For example, supposing a has type $T^{\ell_0}\ \mathbb{B}$, the return type of the bind must then be protected at level ℓ_0. Intuitively, a type is protected at ℓ_0 when every branch node of the type's syntax tree is guarded by some T^ℓ constructor where $\ell_0 \le \ell$, excluding domain types.

Variations of DCC, such as DCC$^{\mathrm{pc}}$ from Tse and Zdancewic [2004] and the sealing calculus from Shikuma and Igarashi [2008], introduce a program counter or observer level to the typing judgment. Accessing the data stored in a box of type $T^{\ell_0}\ A$ is done by comparing the observer level ℓ against the level of the box ℓ_0, similar to the admissible rule T-Unbox in DCOI.

The Dependent Dependency Calculus (DDC) [Choudhury et al. 2022] extends PTS [Barendregt 1993] with dependency tracking. Compared to previous systems, DDC's design was influenced by systems with coeffects, and the variables in the typing context are labeled with levels. The lattice $\mathcal{L}$ in DDC is augmented with two elements $\{C, T\}$ such that $\ell \le C \le T$ for every $\ell \in \mathcal{L}$. The dependency tracking mechanism for levels below C is similar to previous systems, whereas the levels C and T are handled differently through the resurrection mechanism [Pfenning 2001] in order to support compile-time irrelevance. The type conversion rule of DDC is based on indistinguishability, though the observer level of the indistinguishability judgment must be fixed to C, due to the limitation discussed in Section 5.1. The design of DDC directly inspired the use of indistinguishability in DCOI's conversion rule. However, this work handles dependency tracking and compile-time irrelevance through a uniform mechanism, generalizes the conversion rule to arbitrary observer levels, and includes propositional equality.

8.2 Compile-Time Irrelevance

The ability to convert between types based on indistinguishability can be viewed as a generalization of a form of compile-time irrelevance. This feature plays an important role in reasoning about data structures containing embedded proofs: the type checker should treat different witnesses of the same proposition as equal and only compare relevant components.

In Coq and Agda, the respective sorts SProp and Prop [Gilbert et al. 2019] classify types whose inhabitants are treated as definitionally equal. The system then ensures that terms whose types are in this sort can never be pattern matched to produce terms of other sorts, preventing the flow of information from irrelevant types to computationally relevant types. The Squash constructor allows one to embed data types of the relevant sort into SProp and is analogous to the graded modal type $T^{\ell}\ A$ from DCOI. In Gilbert et al. [2019], it is impossible to construct the choice function with the signature Squash (Squash A → A). Instead, the choice function most be postulated as an axiom. In DCOI, the choice function with the analogous type $T^{\mathsf{H}}\ (\mathbb{B}^{\mathsf{H}} \to \mathbb{B})$ can be defined as the term $\mathbf{box}^{\mathsf{H}}\ (\lambda x{:}^{\mathsf{H}}\mathbb{B}.\ x)$.

Resurrection, introduced by Pfenning [2001], is used by DDC, EPTS [Mishra-Linger and Sheard 2008], and Agda [Abel and Scherer 2012]. Similar to DCOI and unlike Gilbert et al. [2019], proof irrelevance in Pfenning [2001] is not determined by sorts. As in DCOI, the type system can label variables as either relevant or irrelevant. However, a type system based on resurrection does not require an observer level in its typing judgment. Instead, when checking irrelevant parts of the term, the type system uses the resurrection operation on the typing context to make irrelevant variables accessible.

To see how resurrection behaves differently from the dependency tracking mechanism of DCOI, consider the terms $\lambda x{:}^{\mathsf{H}}\mathbb{B}.\ \mathbf{box}^{\mathsf{H}}\ x$ and $\mathbf{box}^{\mathsf{H}}\ (\lambda x{:}^{\mathsf{H}}\mathbb{B}.\ x)$, the latter of which is the choice function we have seen earlier. In DCOI, both terms type check at level L. However, with resurrection, only the first term type checks. In the first term, by the abstraction rule, it suffices to show that $\mathbf{box}^{\mathsf{H}}\ x$ is well typed given the typing context $x{:}^{\mathsf{H}}\mathbb{B}$. Before checking the subterm wrapped inside an H box, the typing context is resurrected and the level of the variable x is lowered to L. As a result, the x can legally appear inside the box. In the second term, the H box wraps around the entire lambda expression. Before checking $(\lambda x{:}^{\mathsf{H}}\mathbb{B}.\ x)$, the empty context is resurrected. In this case, resurrection has no effect on the variable x, which has yet to be introduced to the typing context, so its usage in the body of the lambda is rejected.

Resurrection interacts poorly with strong dependent pairs because projections, unlike pattern matching, do not introduce variables that can be resurrected later on. Consider a term xs of type $(\mathsf{m} :^{\mathsf{H}} \mathsf{Nat}) \times \mathsf{Vector\ A\ m}$. We cannot check the second projection proj2 xs because its type, Vector A (proj1 xs), contains the ill-typed term proj1 xs. If we interpret the level H as describing irrelevance at runtime, rejecting proj1 xs in the term but accepting proj1 xs in the type is the desired behavior. However, without an observer level on the typing judgment, there is no way to distinguish between the illegal use of proj1 xs in the term and the legal use of proj1 xs as an argument to Vector. Both must be rejected.

As pointed out by Choudhury et al. [2022], a direct consequence of the above limitation is that the filter function from Section 5.3 must pattern match the returned existential from the recursive call. Eisenberg et al. [2021] observe that filter defined using pattern matching is needlessly strict and loses the streaming behavior of an ordinary list filter function in a call-by-name language.

To overcome the limitations of the resurrection mechanism, DDC restricts labels for run-time irrelevance from being used as the observer in definitional equality. In Agda, there is a disjoint mechanism for run-time irrelevance and function arguments are considered independently for

run-time and compile-time irrelevance. DCOI provides a uniform mechanism that works the same for reasoning about run-time erasure and compile-time equality.

8.3 Quantitative Type Systems

Unlike systems that track only what depends on what, quantitative type systems track usage more precisely. In these systems, inputs are annotated with abstract usage information drawn from an arbitrary semiring. This information specifies how many times a variable may be used in the computation. These systems are related to dependency tracking because computation may not depend on a variable marked with 0 usage.

The run-time irrelevance mechanism in Agda [Abel and Bernardy 2020] and Idris [Brady 2021] is based on Quantitative Type Theory (QTT) [Atkey 2018; McBride 2016]. The typing judgment in QTT takes the form $\Gamma \vdash M :^{\sigma} S$ where $\sigma \in \{0, 1\}$. The σ at the colon indicates the relevance of the term being constructed, similar to the observer level in DCOI. When $\sigma = 1$, the term M is computationally relevant. When $\sigma = 0$, the term M is irrelevant. A variable with 0 usage can only be used when constructing irrelevant terms or in types.

While QTT judgments resemble those of DCOI, QTT only tracks usage at the term level: in types, all tracking is effectively disabled. As a result, QTT's definitional equality cannot rely on usage information. Furthermore, as mentioned in Section 5.1, when instantiated with the boolean semiring, QTT satisfies the upgrade property (Proposition 5.1) using the $1 \leq 0$ ordering. As a result, QTT cannot replace β-equivalence with a usage-aware equality without breaking type soundness.

The more recent systems, GraD [Choudhury et al. 2021] and GRTT [Moon et al. 2021], track usage in both terms and types. However, both systems require a fixed point of reference; the typing judgment is not annotated by a level. (One can view the typing judgment as implicitly indexed by usage 1, meaning that the term being constructed is always computationally relevant or "public".)

In GraD, the typing judgment takes the form $\Delta; \Gamma \vdash a : A$ where Γ is the typing context with usage information and Δ is the same as Γ but without usage information. GraD's regularity property (shown below) is reminiscent of the regularity property (Lemma 3.2) of DCOI.

Lemma 8.1 (GraD: Regularity). *If* $\Delta; \Gamma \vdash a : A$, *then there exists some* Γ' *such that* $\Delta; \Gamma' \vdash A : \mathit{type}$.

As in DCOI, there is independence between term-level and type-level usage information. However, in GraD, the independence appears in the context (Γ' vs. Γ), whereas in DCOI, the independence appears in the observer level of the judgment.

GRTT takes a more fine-grained approach to tracking type-level resource usage. Each variable in GRTT is associated with two usage levels, one for term-level usage and one for type-level usage. A variable may be restricted to the 0 usage at the term level, but still have a nonzero usage in the type, and vice versa. In DCOI, we allow a function's argument to be used relevantly in its type but irrelevantly in its body. However, there is no way to enforce the invariant that a function's argument is never used in its type since well-formedness checks in DCOI can be done at an arbitrarily chosen level. Despite GRTT's fine-grained usage tracking, its conversion rule is β-equivalence.

9 CONCLUSION

In this work, we present DCOI, an extension of Pure Type Systems with dependency tracking that uses indistinguishability for type conversion and internalizes indistinguishability as an indexed equality type. These new features allow reasoning about information flow within the type system, extending the expressiveness of the language. Our design also handles run-time and compile-time irrelevance in a uniform way, using the same mechanism for generating more efficient code and reducing the type-checking effort for programs that rely heavily on type-level computations.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work was supported by the National Science Foundation under Grant Nos. 2006535 and 2327738.

DATA AVAILABILITY STATEMENT

The Coq proofs and the Haskell prototype implementation are available on the ACM Digital Library [Liu et al. 2023].

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (aug 2020), 28 pages. https://doi.org/10.1145/3408972

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012), 1:29. https://doi.org/10.2168/LMCS-8(1:29)2012

Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. 2023. Two-level type theory and applications. *Mathematical Structures in Computer Science* 33, 8 (2023), 688–743. https://doi.org/10.1017/S0960129523000130

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 462–490. https://doi.org/10.1017/S0956796800020025

Henk P. Barendregt. 1993. *Lambda Calculi with Types.* Oxford University Press, Inc., USA, 117–309.

Edwin C. Brady. 2005. *Practical implementation of a dependently typed functional programming language.* Ph. D. Dissertation. Durham University, UK. http://etheses.dur.ac.uk/2800/

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. A Dependent Dependency Calculus. In *Programming Languages and Systems, ESOP 2022 (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 403–430. https://doi.org/10.1007/978-3-030-99336-8_15 Artifact available.

Pritam Choudhury, Harley D. Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021). https://doi.org/10.1145/3434331 Artifact available.

Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88*, Per Martin-Löf and Grigori Mints (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–66. https://doi.org/10.1007/3-540-52335-9_47

Nicolaas Govert de Bruijn. 1994. Some extensions of Automath: the AUT-4 family. In *Studies in Logic and the Foundations of Mathematics*. Vol. 133. Elsevier, 283–288.

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (jul 1977), 504–513. https://doi.org/10.1145/359636.359712

Richard A. Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel Lee. 2021. An Existential Crisis Resolved: Type inference for first-class existential types. *Proc. ACM Program. Lang.* 5, ICFP (Aug. 2021). https://doi.org/10.1145/3473569 Distinguished Paper Award.

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–28. https://doi.org/10.1145/3290316

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph. D. Dissertation. Université Paris 7.

John Hatcliff and Olivier Danvy. 1997. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science* 7, 5 (1997), 507–541. https://doi.org/10.1017/S0960129597002405

András Kovács. 2022. Staged compilation with two-level type theory. *Proceedings of the ACM on Programming Languages* 6, ICFP (aug 2022), 540–569. https://doi.org/10.1145/3547641

Yiyun Liu, Jonathan Chan, Jessica Shi, and Stephanie Weirich. 2023. *Artifact associated with Internalizing Indistinguishability with Dependent Types.* https://doi.org/10.1145/3580424

Per Martin-Löf. 1975. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. North-Holland, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

Conor McBride. 2016. I got plenty o'nuttin'. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Alexandre Miquel. 2001. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359. https://doi.org/10.1007/3-540-45413-6_27

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–364. https://doi.org/10.1007/978-3-540-78499-9_25

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 462–490. https://doi.org/10.1007/978-3-030-72019-3_17

Christine Paulin-Mohring. 1989. Extracting F_ω's Programs from Proofs in the Calculus of Constructions. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 89–104. https://doi.org/10.1145/75277.75285

Frank Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 221–230. https://doi.org/10.1109/LICS.2001.932499

A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Time Sheard and Neal Nelson. 1995. *Type safe abstractions using program generators.* Technical Report 95-013. Oregon Graduate Institute of Science and Technology. https://doi.org/10.6083/w95050724

Naokata Shikuma and Atsushi Igarashi. 2008. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. *Logical Methods in Computer Science* 4 (2008). https://doi.org/10.1007/978-3-540-77505-8_24

Jonathan Sterling and Robert Harper. 2021. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 68, 6, Article 41 (oct 2021), 47 pages. https://doi.org/10.1145/3474834

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science* 248, 1-2 (2000), 211–242. https://doi.org/10.1145/258993.259019

Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. *ACM SIGPLAN Notices* 39, 9 (2004), 115–125. https://doi.org/10.1145/1016848.1016868

Received 2023-07-11; accepted 2023-11-07