Consistency of a Dependent Calculus of Indistinguishability

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

The Dependent Calculus of Indistinguishability (DCOI) uses dependency tracking to identify irrelevant arguments and uses indistinguishability during type conversion to enable proof irrelevance, supporting run-time and compile-time irrelevance with the same uniform mechanism. DCOI also internalizes reasoning about indistinguishability through the use of a propositional equality type indexed by an observer level.

As DCOI is a pure type system, prior work establishes only its syntactic type safety, justifying its use as the basis for a programming language with dependent types. However, it was not clear whether any instance of this system would be suitable for use as a type theory for theorem proving. Here, we identify a suitable instance DCOI^{ω}, which has an infinite predicative universe hierarchy. We show that DCOI^{ω} is logically consistent, normalizing, and that type conversion is decidable. We have mechanized all results using the Coq proof assistant.

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

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

ACM Reference Format:
Yiyun Liu, Jonathan Chan, and Stephanie Weirich. 2025. Consistency of a Dependent Calculus of Indistinguishability. *Proc. ACM Program. Lang.* 9, POPL, Article 7 (January 2025), 27 pages. https://doi.org/10.1145/3704843

1 Introduction

Dependency-tracking type systems govern when functions are allowed to depend on their inputs. More specifically, they classify inputs using various dependency levels (ℓ) and assign a level to the result of a computation. The key idea is that the type system prevents the flow of information from high-level sources to low-level results. In other words, low-level results may not *depend* on high-level input.

A key ingredient of dependency-tracking systems is the *indistinguishability* relation: a definition of program equivalence that takes an observer level into account. This relation identifies more terms than standard definitions of program equivalence because a particular observer may be prevented from making a distinction between program values classified above a certain level. When used for information flow control [Abadi et al. 1999], indistinguishability at low observer levels means that high-level secrets are not revealed.

The *Dependent Calculus of Indistinguishability* (DCOI) [Liu et al. 2024b] adapts dependency tracking to the context of dependently typed programming languages. The primary goal of dependency tracking in DCOI is the identification of *irrelevant arguments*. Such arguments can be erased during compilation to produce faster execution and ignored during equivalence checks at compile time.

Authors' Contact Information: Yiyun Liu, University of Pennsylvania, Philadelphia, USA, liuyiyun@seas.upenn.edu; Jonathan Chan, University of Pennsylvania, Philadelphia, USA, jcxz@seas.upenn.edu; Stephanie Weirich, University of Pennsylvania, Philadelphia, USA, sweirich@seas.upenn.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/1-ART7
https://doi.org/10.1145/3704843

In DCOI, the indistinguishability relation verifies that irrelevant arguments can be erased when executed. If a program produces the same value for all irrelevant arguments at run time, then such arguments may be replaced by trivial values.

Furthermore, the DCOI type system also uses indistinguishability in its conversion rule. If two types cannot be distinguished by some observer, then it is sound to replace one with another at compile time. Thus, DCOI supports compile-time irrelevance, and may ignore irrelevant arguments when checking for type equality. DCOI also internalizes indistinguishability as a type, with a level-indexed elimination form.

Liu et al. [2024b] use syntactic methods to show the soundness of DCOI's type system and that its definition of indistinguishability supports noninterference. However, while syntactic methods are sufficient for these purposes, they cannot be used to reason about semantic properties of type systems such as logical consistency, normalization, and decidable type conversion.

For flexibility, DCOI is designed around a *Pure Type System* (PTS) [Barendregt 1991], where various instances of sorts, rules, and axioms determine the versions of quantification that is allowed in the language. All instances of DCOI are type safe, but only some can be used as the basis of a consistent logic. For example, including the $\mathcal{U} : \mathcal{U}$ axiom produces a language that allows nontermination while others (such as those that include a predicative universe hierarchy, or a single impredicative sort) are conjectured to only include normalizing terms.

We would like to explore the use of dependency tracking in the context of a logically consistent dependent type theory. Therefore, in this paper, we define the language DCOI$^\omega$: a dependent type theory with dependency tracking featuring a predicative universe hierarchy and large eliminations. This calculus is based on an instance of DCOI and includes many of its features, such as level-indexed dependent function types ($\Pi x :^\ell A.\, B$), level-indexed dependent pairs ($\Sigma x :^\ell A.\, B$), and level-indexed propositional equality ($a =^\ell b$), the internalized indistinguishability relation. However, the needs of logical reasoning in a dependent type theory means that we must refine these features and enhance their expressiveness.

Logical consistency means that programmers can use DCOI$^\omega$ to reason internally about their code. Like Coq [Coq Development Team 2019], Agda [Agda Development Team 2023], and Lean [de Moura et al. 2015], DCOI$^\omega$ supports programming and program verification in the same framework. Unlike these systems, programmers may also internally reason about irrelevance, information flow, or any other application of dependency tracking.

More generally, the contributions of this paper are twofold: new extensions that increase the expressiveness of the system, and stronger semantics-based results about this well-behaved instance of the system.

- We extend the language with an empty type $\bot$ with an elimination form that allows programmers to apply the principle of explosion. Importantly, eliminating high-level proofs of the empty type can refute impossible execution paths at any execution level. This feature means that proofs can exist at an erasable dependency level while being useful for programming. The consistency of DCOI$^\omega$ means that there is no closed term of the empty type.
- We develop the theory of the observer-indexed propositional equality type in DCOI$^\omega$ by enhancing the expressiveness of its elimination form, a label-aware variant of the **J** operator. We demonstrate that this form is more expressive than prior work by using it to derive the reasoning principle which states that equalities indistinguishable by a high observer are also indistinguishable by a low observer.
- We provide a weak elimination form for level-indexed dependent pairs, from which we recover strong projections. We show that the weak elimination form for level-indexed pairs is more general than Liu et al.'s treatment of strong projections as primitives; our admissible second

projection rule does not require a well-formedness check when a high-level observer projects the second component of a low-level pair.
- We show that DCOI$^\omega$ is logically consistent (*i.e.* there is no inhabitant of the empty type) and weakly normalizing (*i.e.* every term can be reduced to a normal form). Due to its expressive conversion rule, support for large eliminations, and infinite universe hierarchy, we cannot reuse proofs by Abel and Scherer [2012]; Geuvers [1994] by embedding DCOI$^\omega$ into their systems. Instead, we define a syntactic logical predicate to directly show the metatheoretic results as corollaries of the fundamental theorem.
- Following Takahashi [1995] and Accattoli et al. [2019], we prove the standardization theorem, which states that every weakly normalizing term can be reduced to normal form by the deterministic leftmost-outermost reduction strategy. From the standardization theorem and the weak normalization property, we show that type conversion is decidable. Furthermore, we can extract from our Coq proof an OCaml function that can be used as a decision procedure.

All of our results have been developed using the Coq proof assistant and our proofs are available in the supplementary materials under the `proofs` directory. Furthermore, we have extended the prototype type checker by Liu et al. [2024b], suitable for experimenting with the newly introduced features, which is also available under the `impl` directory. Finally, for reasons of space, not all rules specifying DCOI$^\omega$ can be included in this document. The remaining rules can be found in the reference appendix bundled in our supplementary material.

2 Examples

We start with examples that demonstrate DCOI$^\omega$ in action, with emphasis on its support for run-time and compile-time irrelevance. To make these examples more realistic, our presentation uses the syntax of our prototype implementation, which is based on a bidirectional type system and relies on a constraint solver to infer level annotations.

In these examples, we use the naturals as concrete dependency levels. For example, we use level `L` for run-time relevant code and level `H` for irrelevant terms. Some level annotations are omitted when they can be inferred by the type checker, though many are retained for clarity.

All examples in this section are type checked by our prototype type checker and can be found in `impl/pi/Paper.pi`. Our implementation includes a few convenience features, in particular inductive definitions, not found in the core DCOI$^\omega$ system that we describe in this paper.

2.1 Run- and Compile-Time Irrelevance

Parts of programs that are not meaningfully used during computation can be considered *run-time irrelevant*. An optimizing compiler can then erase run-time irrelevant parts, and irrelevance annotations tell the compiler what can be erased. Well-typedness of a program ensures that irrelevant arguments are never used in a relevant function.

Consider the following polymorphic identity function, with an irrelevant type argument (`@H`) and relevant term argument (`@L`). This function is defined at level `@L`, which is the level of its output.

```
id :@L (A :@H Type) → (@L A) → A
id = λA x. x
```

Although the type argument informs the type checker about how the function should be typed, it does not appear in the function body and is therefore erasable before execution.

This is not to say that irrelevant arguments may never appear in function bodies, because they can be applied in other irrelevant positions. Consider the following function, which takes an irrelevant argument and a constant function.

```
app :@L (@H Bool) → (@L (@H Bool) → Bool) → Bool
```

```
app = λx f. f x
```

The irrelevant parameter `x` is permitted to be used in the body because it is only ever passed as an argument to the constant function `f`, itself a relevant function.

We can talk about the idea of constantness within the language itself using a level-aware propositional equality type, as in the following proof.

```
cong :@L (f :@L (@H Bool) → Bool) → (x y :@H Bool) → (f x =@L f y)
cong = λf x y. refl
```

This proof asserts that a relevant function taking an irrelevant argument is indeed constant in that argument. The equality type `f x =@L f y` is annotated with `L` to indicate that it holds only for relevant terms. It holds by reflexivity, since `f x` and `f y` are considered definitionally equal by the type checker. Definitional equality will check that the two terms are *indistinguishable* at level `L`, which ignores the higher-level arguments `x` and `y`, making them *compile-time irrelevant*. Finally, the `refl` proof in the body of `cong` does not refer to any of the function arguments, allowing us to type `cong` at level `L` to indicate that it is a constant function itself.

2.2 Irrelevance Is Relative

In the polymorphic identity function, the type parameter `A` is irrelevant in the function body but makes a relevant appearance in the type signature. Our system allows `A` to appear as both relevant and irrelevant by generalizing irrelevance as a notion that is relative to an observer level. The level `H` variable `A` is irrelevant with respect to the level `L` observer, the body of the identity function. The well-formedness of the `id`'s type signature can be independently checked at observer level `H`.

Therefore, a variable `x` with `@H` annotation is not intrinsically irrelevant. Rather, its relevance is dependent on the observer level in which `x` appears. The equational theory of DCOI$^{\omega}$ is indexed by an observer level and only treats computations at higher levels as irrelevant. Consider the following example, whose well-typedness depends on how we instantiate the level, which we mark as `?`.

```
irr :@L (P :@? (@H Bool) → Type) → (b :@H Bool) → (@L P b) → (P true)
irr = λf b x. x
```

Whether `irr` type checks depends on whether we can convert the type of `x` from `P b` to `P true`. Since there is nothing we know about `b`, the predicate `P` must behave like a constant predicate, and that is only the case if we instantiate `?` with level `L`.

Although `b` is at level `H` and thus is irrelevant to `irr`, it may still be used relevantly by `P` if `?` is instantiated to level `H` or higher; our type system correctly rejects `irr` in those scenarios.

This shows that as the observer level increases, more terms can be distinguished by definitional equality since there are fewer terms that can be ignored, and the equality is *finer*. As equality types internalize the notion of indistinguishability at a level, we can internalize the ability to lower the level. In particular, given two levels $\ell_0 \leq \ell_1$, the following internalizes the idea that a finer equality indexed by a higher observer level can be downcast to a coarser equality indexed by a lower observer level.

```
downcast : (A : Type) → (x y : A) → (x =@ℓ1 y) → (x =@ℓ0 y)
downcast = λA x y p. subst refl by p
```

In DCOI from prior work, the specification of the eliminator for equalities was not general enough for `downcast` above to be provable.

2.3 Irrelevant Constructor Components

We now introduce inductive types that package up irrelevant terms and which can only be unpackaged by other irrelevant terms. The following is a constructor for subset types [Constable et al. 1986], which are refinement types whose elements contain a relevant term and an explicit

irrelevant proof of a predicate satisfied by the term. The type constructor itself and its parameters are marked as irrelevant.

```
data Subset (A :@H Type) (B :@H (@L A) → Type) :@H Type where
  pack of (x :@L A) (@H B x)
mem :@L (A : Type) → (B : A → Type) → (@L Subset A B) → A
mem = λA B p. case p of pack x y ⇒ x
prf :@H (A : Type) → (B : A → Type) → (p :@L Subset A B) → B (mem A B p)
prf = λA B p. case p of pack x y ⇒ y
```

We can project out both the relevant term and the proof, but the projection of the proof must be irrelevant. Irrelevance of the proof also means that we may equate two members of a subset type merely when the relevant terms are equal, proven as follows.

```
pcong :@H (A :@H Type) → (B :@H A → Type) → (x y : A)
          (bx :@H B x) → (by :@H B y) → (@H x =@L y) →
          (pack x bx =@L pack y by : Subset A B)
pcong = λA B x y bx by p. subst refl by p
```

Thus compile-time irrelevance of proofs helps us write fewer proofs. As a concrete example, suppose we define the naturals `Nat`, their addition operator `+`, a predicate `isEven` on naturals asserting their evenness, a proof that addition is commutative, and a proof that the sum of two even numbers is even. The following are the types of the latter two definitions.

```
comm :@H (n m : Nat) → (n + m =@L m + n)
esum :@H (n m : Nat) → (@H isEven n) → (@H isEven m) → isEven (n + m)
```

An even number `Even` can be represented as a subset type of even naturals `Subset Nat isEven`, and correspondingly we can define their addition.

```
eadd :@L Even → Even → Even
eadd = λen em. case en of pack n en' ⇒
               case em of pack m em' ⇒ pack (n + m) (esum n m en' em')
```

We can then prove that addition of even numbers is also commutative using `pcong`.

```
ecomm :@H (en em : Even) → (eadd en em =@L eadd em en)
ecomm = λen em. case en of pack n en' ⇒
                case em of pack m em' ⇒
                  pcong Nat isEven (n + m) (m + n)
                        (esum n m en' em') (esum m n em' en') (comm n m)
```

If the proof in the subset type were not irrelevant, we would have the additional proof obligation of showing that `(esum n m en' em')` and `(esum m n em' en')` are equal.

2.4 Eliminating Empty Types

In general, irrelevant inductives may not be eliminated to relevant terms. The exception in DCOI$^{\omega}$ is the empty type, whose eliminator may be typed at any level, regardless of the level of its scrutinee.

```
data Empty :@H Type where {}
```

This feature, new to DCOI$^{\omega}$, allows us to handle impossible match cases, even when the proof of impossibility is irrelevant. For example, consider taking the head of a nonempty list given an irrelevant proof of its nonemptiness. The definitions of the list type and its length are standard.

```
data List (A :@H Type) :@H Type where
  nil
  cons of (@L A) (@L List A)
length :@L (A :@H Type) → List A → Nat
```

A safe head function requires a proof that the length of the given list is not equal to zero, *i.e.* that `(@H (@H (length Nat l) = zero) → Empty)`. This proof should be irrelevant, so it is marked with `@H`. In the `nil` case, the type of the proof is `(zero = zero) → Empty`, so we can obtain and eliminate a proof of the empty type.

```
head :@L (l : List Nat) → (@H (@H (length Nat l) = zero) → Empty) → Nat
head = λl. case l of
             nil      ⇒ λf. case (f Refl) of {}
             cons x _ ⇒ λ_. x
```

3 DCOI$^\omega$

In this section, we present the grammar and judgments of DCOI$^\omega$, which is an instantiation of the Dependent Calculus of Indistinguishability (DCOI) [Liu et al. 2024b] with the PTS axioms and rules of a predicative universe hierarchy. The grammar is given in Figure 1.

$\Gamma, \Delta ::= \cdot \mid \Gamma, x :^\ell A$		Typing contexts
$\Xi, \Phi ::= \cdot \mid \Xi, x : \ell$		Level contexts
$A, B, C,$		
$a, b, c, p ::=$		Terms
$\mid x \mid \mathcal{U}_i \mid \bot \mid \mathbf{absurd}\ b$		*variables, type universes, empty type, absurd*
$\mid \Pi x :^\ell A.\ B \mid \lambda x^\ell.\ b \mid b\ a^\ell$		*function types, abstractions, applications*
$\mid \Sigma x :^\ell A.\ B \mid (a^\ell, b) \mid \mathbf{let}\ (x^{\ell_0}, y) = a^\ell\ \mathbf{in}\ b$		*pair types, pairs, matching*
$\mid \top \mid \mathbf{tt} \mid \mathbf{let}\ \mathbf{tt} = a^\ell\ \mathbf{in}\ b$		*unit type, unit value, eliminator*
$\mid a =^\ell b \mid \mathbf{refl} \mid \mathbf{J}\ c\ p^\ell$		*equality types, reflexivity, $\mathcal{J}$ eliminator*
$\mid \mathbf{Nat} \mid \mathbf{zero} \mid \mathbf{succ}\ a \mid \mathbf{ind}^\ell\ a\ b_0\ (\lambda x\ y.\ b_1)$		*naturals, zero, successor, induction*

Fig. 1. Grammar of contexts and terms

Following Abadi et al. [1999], DCOI$^\omega$ is parameterized over a meet-semilattice of dependency levels, ℓ. We use $\ell_0 \wedge \ell_1$ for the meet of ℓ_0 and ℓ_1. In the following examples, we use concrete levels low L and high H, where $\mathsf{L} < \mathsf{H}$.

Universe levels (i, j, k) are (meta-level) naturals and $i \vee j$ computes the maximum of i and j.

Function types, abstractions, pair types, matching and induction are all binding forms. Although our mechanization uses de Bruijn indices for representing binding and simultaneous substitutions, for clarity we use named variables and single substitutions here. We use the metavariables x, y, z for variables and $a[b/x]$ for the substitution of a term b in place of the variable x inside the term a. As in Section 2, we use ${}^\ell A \to B$ for $\Pi x :^\ell A.\ B$ when x does not appear in B.

The main judgment is the typing relation, which has the form $\boxed{\Gamma \vdash a :^\ell A}$ for a term a that is well typed under context Γ at observer level ℓ with type A. Its rules are given in Figure 2. To reduce clutter, we also use the judgment $\boxed{\Gamma \vdash a : A}$ to mean that there exists some level ℓ such that $\Gamma \vdash a :^\ell A$ holds. The typing judgment is defined mutually with the context well-formedness judgment $\boxed{\vdash \Gamma}$, which holds if the context's types are well typed by some universe $\mathcal{U}_i$. Premises which may be omitted in admissible variants of rules without them are highlighted in grey.

Our syntax is designed to highlight the parts of terms which play a role in evaluation. Therefore, type annotations are omitted, and consequently the typing judgment is not decidable. This minimalist design lends flexibility in which annotations a type checker implementation may choose to add to the syntax.

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Well-typedness)*

$$\textsc{Wt-Var}\ \frac{\vdash \Gamma \quad x :^{\ell_0} A \in \Gamma \quad \ell_0 \le \ell}{\Gamma \vdash x :^{\ell} A} \qquad \textsc{Wt-Univ}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathcal{U}_i :^{\ell} \mathcal{U}_{i+1}} \qquad \textsc{Wt-Empty}\ \frac{\vdash \Gamma}{\Gamma \vdash \bot :^{\ell} \mathcal{U}_i} \qquad \textsc{Wt-Absurd}\ \frac{\Gamma \vdash b : \bot \quad \Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash \mathbf{absurd}\ b :^{\ell} A}$$

$$\textsc{Wt-Pi}\ \frac{\Gamma \vdash A :^{\ell} \mathcal{U}_i \quad \Gamma, x :^{\ell_0} A \vdash B :^{\ell} \mathcal{U}_j}{\Gamma \vdash \Pi x :^{\ell_0} A.\, B :^{\ell} \mathcal{U}_{i \vee j}} \qquad \textsc{Wt-Abs}\ \frac{\Gamma, x :^{\ell_0} A \vdash b :^{\ell} B \quad \Gamma \vdash \Pi x :^{\ell_0} A.\, B : \mathcal{U}_i}{\Gamma \vdash \lambda x^{\ell_0}.\, b :^{\ell} \Pi x :^{\ell_0} A.\, B} \qquad \textsc{Wt-App}\ \frac{\Gamma \vdash b :^{\ell} \Pi x :^{\ell_0} A.\, B \quad \Gamma \vdash a :^{\ell_0} A}{\Gamma \vdash b\, a^{\ell_0} :^{\ell} B[a/x]}$$

Fig. 2. Typing rules (universes, empty, functions)

The typing rules are defined mutually with the context well-formedness rules, omitted here, which assert the well-sortedness of its types. When viewed as a PTS, the rules Wt-Univ and Wt-Pi correspond to the sort axioms $(\mathcal{U}_i, \mathcal{U}_{i+1})$ for every i and the sort rules $(\mathcal{U}_i, \mathcal{U}_j, \mathcal{U}_{i \vee j})$ for every i, j.

Except for the use of levels for dependency tracking, typing rules of DCOI$^{\omega}$ are standard for dependent type theory. In the rest of this section, we discuss how dependency levels interact with the rules of Figure 2 in Section 3.1. We discuss how the conversion rule, which changes the type of a term to an equivalent one, uses a definitional equality defined in terms of indistinguishability in Section 3.2. We then look at the rules for dependent pair types and the unit type (Section 3.3), and finally for propositional equality types (Section 3.4), which internalize indistinguishability.

DCOI$^{\omega}$ also includes the naturals and their induction principle. Notably, the induction principle permits large eliminations: we can branch on a natural and return different types for each branch. However, as their interactions with dependency levels are similar to those of the other constructs, we omit them from our discussion and only list their rules in the supplementary appendix A.1.

3.1 Dependency Tracking

The observer level ℓ of a typing judgment $\Gamma \vdash a :^{\ell} A$ tracks what variables a may use, as well as the level at which a itself may be used in later computation. The former is enforced by the constraint in rule Wt-Var, which permits a variable to be used only if the observer level is at least as high as that of the variable. Informally, higher-level information cannot leak into a lower-level term.

In rule Wt-Abs, the function parameter is annotated with a fixed level, thus restricting its uses to observers at least that high. Even so, there is no restriction on that level, and a function may be applied to an argument of a higher level, as long as the annotated level of the function parameter matches the annotated level of the application in rule Wt-App. Allowing a higher-level parameter is useful because it may be used in the function's *type*, which is typed at an arbitrary level independent of the function itself in the second premise of rule Wt-Abs. By rule Wt-Pi, there are no restrictions on the levels of the domain and codomain types either (as long as they match). As an example, consider the following typing judgment.

$$\cdot \vdash (\lambda X^{\mathsf{H}}.\, \lambda x^{\mathsf{H}}.\, \lambda f^{\mathsf{L}}.\, f\, x^{\mathsf{H}}) :^{\mathsf{L}} (\Pi X :^{\mathsf{H}} \mathcal{U}_i.\, {}^{\mathsf{H}}X \to {}^{\mathsf{L}}({}^{\mathsf{H}}X \to X) \to X)$$

The entire function is typed at level L, while the parameters X and x are annotated at level H, so the inner function body cannot be x, for instance, even if it has the correct type. On the other hand, the parameter f is a function that takes a high-level X and returns a low-level X, so $f\, x^{\mathsf{H}}$ is a valid function body. Meanwhile, the type of the function need only be well-typed at *some* level, and in particular is well-typed at H, which allows using the high-level X in the codomain of the type.

In general, we can prove a regularity lemma stating that the type of a well-typed term is itself well typed at a level not necessarily related to that of the term, and may even be higher, as is the case for the example above. Intuitively, this is permitted because there is no way for a term to depend on its type at run time, so the type of a lower-level term can safely use higher-level information without it leaking into the term.

Lemma 3.1 (Regularity). *If* $\Gamma \vdash a :^{\ell} A$, *then* $\vdash \Gamma$ *holds, and there exist* ℓ_0, i *such that* $\Gamma \vdash A :^{\ell_0} \mathcal{U}_i$.

While generally a lower-level term may not meaningfully use higher-level information, a proof of the empty type $\bot$ at *any* level may be eliminated to *any other* level via rule Wt-Absurd. By logical consistency, which we prove in Section 5, there is no closed term of type $\bot$, and so it cannot contain any meaningful information, and furthermore represents a dead execution path. From a type-theoretic perspective, the unconstrained elimination level allows us to handle impossible cases of a lower-level term using a higher-level proof of its impossibility.

3.2 Definitional Equality and Indistinguishability

$$\textsc{Wt-Conv}\ \frac{\Gamma \vdash a :^{\ell} A \qquad \Gamma \vdash B : \mathcal{U}_i \qquad |\Gamma| \vdash A \equiv B}{\Gamma \vdash a :^{\ell} B} \qquad \textsc{E-Conv}\ \frac{a \Rightarrow^* c_0 \qquad b \Rightarrow^* c_1 \qquad \Xi \vdash c_0 \equiv^{\ell} c_1}{\Xi \vdash a \equiv b} \qquad \textsc{P-AppAbs}\ \frac{a_0 \Rightarrow a_1 \qquad b_0 \Rightarrow b_1}{(\lambda x^{\ell_0}.\, b_0)\ a_0^{\ell_0} \Rightarrow b_1[a_1/x]}$$

Fig. 3. Conversion, definitional equality, and parallel reduction (β only) rules

The typing rules also include rule Wt-Conv, shown in Figure 3, which allows converting the type of a well-typed term to a definitionally equal type. Our definitional equality is untyped, but eventually relies on the levels of variables, so its judgment form $\boxed{\Xi \vdash a \equiv b}$ requires a level context Ξ that maps variables to levels (only). In rule Wt-Conv, this level context is produced by erasing the typing context, denoted $\boxed{|\Gamma|}$, and dropping all type annotations.

Definitional equality is defined by rule E-Conv in Figure 3, which states that terms are equal when they reduce to terms that are *indistinguishable* by some observer ℓ, defined in Figure 4 and described below. To make this level visible in the judgment, we also define $\boxed{\Xi \vdash a \Leftrightarrow^{\ell} b}$ as a level-annotated form of definitional equality: it holds iff $a \Rightarrow^* c_0$, $b \Rightarrow^* c_1$, and $\Xi \vdash c_0 \equiv^{\ell} c_1$. Consequently, $\Xi \vdash a \equiv b$ holds iff there exists some level ℓ such that $\Xi \vdash a \Leftrightarrow^{\ell} b$ holds.

Parallel reduction $\boxed{a \Rightarrow b}$ consists of the β-reduction rules for our elimination forms, listed in Figure 3 and in following subsections, along with rules for its reflexive, congruent closure, which are omitted here and can be found in the supplementary appendix A.3. Multi-step parallel reduction $\boxed{a \Rightarrow^* b}$ is the transitive closure of parallel reduction.

Two terms are indistinguishable at observer level ℓ if they have the same shape and their subterms are also indistinguishable at ℓ. The only exception is rule I-App that uses *guarded indistinguishability*, $\boxed{\Xi \vdash a \equiv^{\ell} b}$, which is mutually defined with indistinguishability in Figure 4.

Indistinguishability is *coarser* than mere α-equivalence (and hence $\Xi \vdash a \equiv b$ is coarser than mere β-equivalence of a and b) because guarded indistinguishability sometimes identifies terms that are not α-equivalent. In rule I-App, if the arguments are labeled at ℓ_0, but observed at ℓ, guarded indistinguishability compares the arguments based on the relationship between ℓ_0 and ℓ. If $\ell_0 \le \ell$, meaning that the observer can fully see the arguments, then rule GI-Dist applies, and the arguments must themselves be indistinguishable at ℓ. Otherwise, if $\ell_0 \not\le \ell$, then rule GI-Indist applies, and the arguments are too high to be observed at ℓ, so indistinguishability of the applications does not consider the arguments. As a simple example, $f : \mathsf{L}, x : \mathsf{H}, y : \mathsf{H} \vdash f\ x^{\mathsf{H}} \equiv^{\mathsf{L}} f\ y^{\mathsf{H}}$ holds even though x and y could be anything, because the observer level is L, while the arguments' levels are H and

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

$$\textsc{GI-Dist}\ \frac{\Xi \vdash a \equiv^{\ell} b \quad \ell_0 \le \ell}{\Xi \vdash a \equiv^{\ell}_{\ell_0} b} \qquad \textsc{GI-Indist}\ \frac{\ell_0 \not\le \ell}{\Xi \vdash a \equiv^{\ell}_{\ell_0} b} \qquad \textsc{I-Var}\ \frac{x : \ell_0 \in \Xi \quad \ell_0 \le \ell}{\Xi \vdash x \equiv^{\ell} x}$$

$$\textsc{I-Empty}\ \frac{}{\Xi \vdash \bot \equiv^{\ell} \bot} \qquad \textsc{I-Absurd}\ \frac{}{\Xi \vdash \mathbf{absurd}\ b_0 \equiv^{\ell} \mathbf{absurd}\ b_1}$$

$$\textsc{I-Pi}\ \frac{\Xi \vdash A_0 \equiv^{\ell} A_1 \quad \Xi, x : \ell_0 \vdash B_0 \equiv^{\ell} B_1}{\Xi \vdash \Pi x :^{\ell_0} A.\, B \equiv^{\ell} \Pi x :^{\ell_0} A_1.\, B_1} \qquad \textsc{I-Abs}\ \frac{\Xi, x : \ell_0 \vdash b_0 \equiv^{\ell} b_1}{\Xi \vdash \lambda x^{\ell_0}.\, b_0 \equiv^{\ell} \lambda x^{\ell_0}.\, b_1}$$

$$\textsc{I-App}\ \frac{\Xi \vdash b_0 \equiv^{\ell} b_1 \quad \Xi \vdash a_0 \equiv^{\ell}_{\ell_0} a_1}{\Xi \vdash b_0\, a_0^{\ell_0} \equiv^{\ell} b_1\, a_1^{\ell_0}}$$

Fig. 4. (Guarded) indistinguishability rules (empty, functions)

therefore unobservable. Indistinguishability of unobservable arguments articulates the intuition that functions may safely ignore these arguments that they cannot meaningfully use.

Indistinguishability is a partial equivalence relation. The source of partiality is due to the level comparison that occurs in rule I-Var, where a variable may only be indistinguishable from itself if the observer level is high enough to observe it. This prevents equating arbitrary terms via function applications, such as $x : \mathsf{H}, y : \mathsf{H} \vdash (\lambda z^{\mathsf{H}}.\, z)\, x^{\mathsf{H}} \equiv (\lambda z^{\mathsf{H}}.\, z)\, y^{\mathsf{H}}$, since the bodies of the functions are not indistinguishable. Without this level comparison, we have via rule E-Conv the equalities $\cdot \vdash \top \equiv (\lambda z^{\mathsf{H}}.\, z)\, \top^{\mathsf{H}}$, $\cdot \vdash (\lambda z^{\mathsf{H}}.\, z)\, \top^{\mathsf{H}} \equiv (\lambda z^{\mathsf{H}}.\, z)\, \bot^{\mathsf{H}}$, and $\cdot \vdash (\lambda z^{\mathsf{H}}.\, z)\, \bot^{\mathsf{H}} \equiv \bot$, which would permit a closed term of $\bot$ via repeated applications of rule Wt-Conv.

An important property of indistinguishability is the simulation property, which enforces that indistinguishable terms must reduce in lockstep. This is crucial for proving further syntactic and semantic lemmas in Sections 4 and 5, such as the preservation of definitional equality between two terms as they reduce and the preservation of meaning for indistinguishable types.

Lemma 3.2 (Simulation). *If $\Xi \vdash a_0 \equiv^{\ell} b_0$ and $a_0 \Rightarrow^{*} a_1$, then there exists some b_1 such that $b_0 \Rightarrow^{*} b_1$ and $\Xi \vdash a_1 \equiv^{\ell} b_1$.*

Compared to prior work by Liu et al. [2024b], where indistinguishability is defined as an undecidable relation that incorporates both β-reduction and conversion of irrelevant components, our formulation of type conversion is more structured and breaks down type conversion into two separate steps: reduction and checking indistinguishability. This algorithmic version of indistinguishability allows us to reduce the problem of proving decidability of type conversion to the problem of finding a normalization procedure for well-typed terms (Section 5.3). Finally, we note that if we instantiate DCOI$^{\omega}$ with a singleton lattice, then indistinguishability degenerates into α-equivalence, and the definitional equality degenerates into β-equivalence.

3.3 Dependent Pairs and Unit

Dependent pairs are annotated with a fixed level in their first component, given by ℓ_0 in rules Wt-Sig and Wt-Pair of Figure 5. Like for function types, the levels of pair types are not necessarily related to that annotated level. While the first component of the pair constructor has a fixed level, the level of the second component is the level of the overall pair.

In contrast to prior work by Liu et al. [2024b], pairs are eliminated using a pattern-matching expression instead of projections. The eliminator is typed according to rule Wt-Let, and its reduction behavior is given by rule P-LetPair. It scrutinizes a pair a annotated at level ℓ_1, binding its first and second components as variables x at level ℓ_0 and y at level ℓ_1 respectively in the body b, which must be typed at the same level ℓ as the overall eliminator. The overall type of the eliminator

$$\textsc{Wt-Sig}\ \frac{\Gamma \vdash A :^{\ell} \mathcal{U}_i \qquad \Gamma, x :^{\ell_0} A \vdash B :^{\ell} \mathcal{U}_j}{\Gamma \vdash \Sigma x :^{\ell_0} A.\, B :^{\ell} \mathcal{U}_{i \vee j}} \qquad \textsc{Wt-Pair}\ \frac{\Gamma \vdash a :^{\ell_0} A \qquad \Gamma \vdash b :^{\ell} B[a/x] \qquad \Gamma \vdash \Sigma x :^{\ell_0} A.\, B : \mathcal{U}_i}{\Gamma \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x :^{\ell_0} A.\, B}$$

$$\textsc{Wt-Let}\ \frac{\begin{array}{c}\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x :^{\ell_0} A \vdash B : \mathcal{U}_j \\ \Gamma, z :^{\ell_1} \Sigma x :^{\ell_0} A.\, B \vdash C : \mathcal{U}_k \\ \Gamma \vdash a :^{\ell_1} \Sigma x :^{\ell_0} A.\, B \\ \Gamma, x :^{\ell_0} A, y :^{\ell_1} B \vdash b :^{\ell} C[(x^{\ell_0}, y)/z] \qquad \ell_1 \le \ell\end{array}}{\Gamma \vdash \mathbf{let}\,(x^{\ell_0}, y) = a^{\ell_1}\ \mathbf{in}\ b :^{\ell} C[a/z]} \qquad \textsc{P-LetPair}\ \frac{\begin{array}{c} a_0 \Rightarrow a_1 \\ b_0 \Rightarrow b_1 \qquad c_0 \Rightarrow c_1\end{array}}{\mathbf{let}\,(x^{\ell_0}, y) = (a_0{}^{\ell_0}, b_0)^{\ell}\ \mathbf{in}\ c_0 \Rightarrow c_1[a_1/x, b_1/y]}$$

$$\textsc{Wt-Unit}\ \frac{\vdash \Gamma}{\Gamma \vdash \top :^{\ell} \mathcal{U}_i} \qquad \textsc{Wt-TT}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{tt} :^{\ell} \top} \qquad \textsc{Wt-Seq}\ \frac{\Gamma \vdash a :^{\ell_1} \top \qquad \Gamma \vdash b :^{\ell} C[\mathbf{tt}/x] \qquad \Gamma, x :^{\ell_1} \top \vdash C : \mathcal{U}_i \qquad \ell_1 \le \ell}{\Gamma \vdash \mathbf{let}\,\mathbf{tt} = a^{\ell_1}\ \mathbf{in}\ b :^{\ell} C[a/x]} \qquad \textsc{P-SeqTT}\ \frac{b_0 \Rightarrow b_1}{\mathbf{let}\,\mathbf{tt} = \mathbf{tt}^{\ell}\ \mathbf{in}\ b_0 \Rightarrow b_1}$$

Fig. 5. Typing and parallel reduction rules (pairs, unit)

is specified by the motive C, which is abstracted over the scrutinee z. The level of the scrutinee ℓ_1 is not exactly ℓ but bounded above by it; this is a subtle technical issue, which we defer to Section 6.1.

The first and second pair projections of a pair a at level ℓ_1 can be recovered as follows.

$$\pi_1^{\ell_0}\ a := \mathbf{let}\,(x^{\ell_0}, y) = a^{\ell_1}\ \mathbf{in}\ x \qquad\qquad \pi_2^{\ell_0}\ a := \mathbf{let}\,(x^{\ell_0}, y) = a^{\ell_1}\ \mathbf{in}\ y$$

The typing rules for the projections can be stated as admissible rules with the same premises as the rules in DCOI, and are given in Figure 6 as Wt-proj1 and Wt-proj2. The first projection is well typed only if the first component's level is bounded by the level of the pair: we can only project an observable component. The second projection is well typed only if the first projection is well typed at the first component's level ℓ_0. This is because the type of the second component is dependent on the first component at level ℓ_0. Rule Wt-Proj1 tells us that the $\pi_1^{\ell_0}\ a$ is at level ℓ, but we need the extra side condition that $\pi_1^{\ell_0}\ a$ is typed at level ℓ_0 for $B[\pi_1^{\ell_0}\ a/x]$ to be well-formed.

$$\textsc{Wt-proj1}\ \frac{\Gamma \vdash a :^{\ell} \Sigma x :^{\ell_0} A.\, B \qquad \ell_0 \le \ell}{\Gamma \vdash \pi_1^{\ell_0}\ a :^{\ell} A} \qquad \textsc{Wt-proj2}\ \frac{\Gamma \vdash a :^{\ell} \Sigma x :^{\ell_0} A.\, B \qquad \Gamma \vdash \pi_1^{\ell_0}\ a :^{\ell_0} A}{\Gamma \vdash \pi_2^{\ell_0}\ a :^{\ell} B[\pi_1^{\ell_0}\ a/x]} \qquad \textsc{Wt-proj2alt}\ \frac{\Gamma \vdash a :^{\ell} \Sigma x :^{\ell_0} A.\, B}{\Gamma \vdash \pi_2^{\ell_0}\ a :^{\ell} \mathbf{let}\,(y^{\ell_0}, z) = a^{\ell}\ \mathbf{in}\ B[y/x]}$$

Fig. 6. Admissible typing rules for pair projections

However, this is not the only way we can ensure the well-typedness of the second projection. The alternate admissible rule Wt-proj2alt only requires well-typedness of the pair and not of the first projection. Here, the type of the second projection matches on the pair and then substitutes in the first component inside of B, rather than matching on the pair within the substitution in B. The second projection is therefore well typed regardless of the well-typedness of the first projection. In particular, we can take the second projection of a higher-level pair even if it contains a lower-level first component. This was not possible when pair projections were the primitive operators on a pair and not the eliminator we have, which is strictly more expressive.

The typing rules for the unit type (rule Wt-Unit), its sole inhabitant (rule Wt-TT), and the dependent eliminator (rule Wt-Seq) are standard, with reduction given by rule P-SeqTT. In rule Wt-Seq, the level constraint on the scrutinee exists for the same reason as in the pair eliminator.

The box type $\mathbf{T}^{\ell_0}\, A$ [Abadi et al. 1999] can be seen as a pair with information only in its first component, of which we only ever take the first projection, and can be encoded as a unary tuple.

$$\mathbf{T}^{\ell_0}\, A \coloneqq \Sigma x :^{\ell_0} A.\, \top \qquad \mathbf{box}^{\ell_0}\, a \coloneqq (a^{\ell_0}, \mathbf{tt}) \qquad \mathbf{unbox}^{\ell_0} a \coloneqq \pi_1^{\ell_0}\, a$$

By these encodings and the typing rules for pairs, we obtain the admissible typing rules for box types (Wt-T), boxes (Wt-Box), and unboxing (Wt-Unbox) in Figure 7. The level constraint in Wt-Unbox comes from the corresponding constraint in rule Wt-Let, and intuitively says that the only values that can be unboxed are observable ones.

$$\textsc{Wt-T}\ \frac{\Gamma \vdash A :^{\ell} \mathcal{U}_i}{\Gamma \vdash \mathbf{T}^{\ell_0}\, A :^{\ell} \mathcal{U}_i} \qquad \textsc{Wt-Box}\ \frac{\Gamma \vdash a :^{\ell_0} A}{\Gamma \vdash \mathbf{box}^{\ell_0}\, a :^{\ell} \mathbf{T}^{\ell_0}\, A} \qquad \textsc{Wt-Unbox}\ \frac{\Gamma \vdash a :^{\ell} \mathbf{T}^{\ell_0}\, A \qquad \ell_0 \le \ell}{\Gamma \vdash \mathbf{unbox}^{\ell_0} a :^{\ell} A}$$

Fig. 7. Admissible typing rules for box types

The indistinguishability rules for pair types, the unit type, and their eliminators are given in Figure 8. The primary use case of the box type is to be able to enclose higher-level terms and to treat them as indistinguishable from one another by a lower-level observer. That is, $\Xi \vdash \mathbf{box}^{\mathsf{H}}\, a \equiv^{\mathsf{L}} \mathbf{box}^{\mathsf{H}}\, b$ ought to hold regardless of a and b. This idea generalizes to pairs by using guarded indistinguishability when comparing first components in rule I-Pair, similar to how function arguments are compared. In rules I-Let and I-Seq, the constraints $\ell_1 \le \ell$ are required due to their presence in rules Wt-Let and Wt-Seq.

$$\textsc{I-Sig}\ \frac{\Xi \vdash A_0 \equiv^{\ell} A_1 \qquad \Xi, x : \ell_0 \vdash B_0 \equiv^{\ell} B_1}{\Xi \vdash \Sigma x :^{\ell_0} A_0.\, B_0 \equiv^{\ell} \Sigma x :^{\ell_0} A_1.\, B_1} \qquad \textsc{I-Pair}\ \frac{\Xi \vdash a_0 \equiv^{\ell}_{\ell_0} a_1 \qquad \Xi \vdash b_0 \equiv^{\ell} b_1}{\Xi \vdash (a_0{}^{\ell_0}, b_0) \equiv^{\ell} (a_1{}^{\ell_0}, b_1)}$$

$$\textsc{I-Unit}\ \frac{}{\Xi \vdash \top \equiv^{\ell} \top} \qquad \textsc{I-TT}\ \frac{}{\Xi \vdash \mathbf{tt} \equiv^{\ell} \mathbf{tt}}$$

$$\textsc{I-Let}\ \frac{\Xi \vdash a_0 \equiv^{\ell} a_1 \qquad \Xi, x : \ell_0, y : \ell_1 \vdash b_0 \equiv^{\ell} b_1 \qquad \ell_1 \le \ell}{\Xi \vdash \mathbf{let}\, (x^{\ell_0}, y) = a_0{}^{\ell_1}\, \mathbf{in}\, b_0 \equiv^{\ell} \mathbf{let}\, (x^{\ell_0}, y) = a_1{}^{\ell_1}\, \mathbf{in}\, b_1} \qquad \textsc{I-Seq}\ \frac{\Xi \vdash a_0 \equiv^{\ell} a_1 \qquad \Xi \vdash b_0 \equiv^{\ell} b_1 \qquad \ell_1 \le \ell}{\Xi \vdash \mathbf{let}\, \mathbf{tt} = a_0{}^{\ell_1}\, \mathbf{in}\, b_0 \equiv^{\ell} \mathbf{let}\, \mathbf{tt} = a_1{}^{\ell_1}\, \mathbf{in}\, b_1}$$

Fig. 8. Indistinguishability rules (pairs, unit)

3.4 Propositional Equality

$$\textsc{Wt-Eq}\ \frac{\Gamma \vdash a :^{\ell_0} A \qquad \Gamma \vdash b :^{\ell_0} A \qquad \ell_0 \le \ell}{\Gamma \vdash a =^{\ell_0} b :^{\ell} \mathcal{U}_j} \qquad \textsc{Wt-Refl}\ \frac{\Gamma \vdash a :^{\ell_0} A}{\Gamma \vdash \mathbf{refl} :^{\ell} a =^{\ell_0} a} \qquad \textsc{P-JRefl}\ \frac{c_0 \Rightarrow c_1}{\mathbf{J}\, c_0\, \mathbf{refl}^{\ell} \Rightarrow c_1}$$

$$\textsc{Wt-J}\ \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a :^{\ell_1} A \qquad \Gamma \vdash b :^{\ell_1} A \qquad \Gamma \vdash p :^{\ell_2} a =^{\ell_0} b \qquad \Gamma, x :^{\ell_1} A, y :^{\ell_2} a =^{\ell_0} x \vdash C :^{\ell_0} \mathcal{U}_j \qquad \Gamma \vdash c :^{\ell} C[a/x, \mathbf{refl}/y] \qquad \ell_1 \le \ell_0 \qquad \ell_2 \le \ell}{\Gamma \vdash \mathbf{J}\, c\, p^{\ell_2} :^{\ell} C[b/x, p/y]}$$

Fig. 9. Typing and parallel reduction rules (equality)

The propositional equality type internalizes the indistinguishability judgment, enabling reasoning about indistinguishability within DCOI$^{\omega}$ itself. The typing rules for constructs related to the

equality type are given in Figure 9. The type $a =^{\ell_0} b$ represents the proposition that a and b are indistinguishable by observers at level ℓ_0. By rule Wt-Eq, the endpoints must themselves be well typed at level ℓ_0, since it only makes sense to talk about the indistinguishability of observable terms, just as rule I-Var only holds for observable variables. The observer level ℓ_0 must also be bounded by the overall level ℓ to ensure the consistency of the equational theory, as explained by Liu et al. [2024b]. Its constructor is the usual reflexivity proof **refl**, given by rule Wt-Refl.

In rule Wt-J, the eliminator **J** takes as scrutinee an equality proof p annotated at level ℓ_2, along with a body c. The scrutinee must be well typed as an equality between a and b of type A, observed at level ℓ_0. The observer, so to speak, is the motive C, abstracted over the right-hand side of the equality x (the left-hand side being fixed at a) and the equality proof itself y. As the observer, the motive must be well typed at level ℓ_0. The type of the body is the motive instantiated with a and **refl**, while the type of the fully-applied eliminator is the motive instantiated with b and p. The levels of x and y must therefore match the levels of b and p, respectively. Because by rule P-JRefl the eliminator reduces on **refl** to the body, the level of the body must match the level ℓ of the overall eliminator. The level of the scrutinee ℓ_2 is bounded above by the observer level ℓ, and thus eliminating an equality proof always counts as an observation even though no branching happens. We treat the equality proof as observable to retain decidable type conversion (Section 6.3). The non-exact match between ℓ_2 and ℓ makes the rule more flexible than a rule that simply requires them to be equal. The necessity of the flexibility is discussed in Section 6.1.

The other level constraint in rule Wt-J, which allows the equality endpoints to be typed at a level ℓ_1 possibly lower than the equality observer level ℓ_0, increases the expressivity of the eliminator, since the motive C is able to abstract over x at the lower level ℓ_1. In particular, it allows us to show the following lemma.

Lemma 3.3 (Downgrade). *Suppose* $\Gamma \vdash a :^{\ell_0} A$ *and* $\Gamma \vdash b :^{\ell_0} A$. *If* $\Gamma \vdash p :^{\ell} a =^{\ell_1} b$ *and* $\ell_0 \le \ell_1$, *then* $\Gamma \vdash \mathbf{J}\ \mathbf{refl}\ p^{\ell} :^{\ell} a =^{\ell_0} b$.

Downgrade permits lowering the observer level of an equality proof. Here, the motive is $a =^{\ell_0} x$, which would not have been well typed if the level of x had to match the observer level ℓ_1 of the equality p being eliminated. Intuitively, this rule internalizes the idea that if two terms are indistinguishable at some higher level, then they also are at a lower level, since a lower-level indistinguishability is coarser than a higher-level one, and treats more terms as indistinguishable.

$$\text{I-Eq}\ \frac{\Xi \vdash a_0 \equiv^{\ell} a_1 \quad \Xi \vdash b_0 \equiv^{\ell} b_1 \quad \ell_0 \le \ell}{\Xi \vdash a_0 =^{\ell_0} b_0 \equiv^{\ell} a_1 =^{\ell_0} b_1} \qquad \text{I-Refl}\ \frac{}{\Xi \vdash \mathbf{refl} \equiv^{\ell} \mathbf{refl}} \qquad \text{I-J}\ \frac{\Xi \vdash p_0 \equiv^{\ell} p_1 \quad \Xi \vdash c_0 \equiv^{\ell} c_1 \quad \ell_2 \le \ell}{\Xi \vdash \mathbf{J}\ c_0\ p_0{}^{\ell_2} \equiv^{\ell} \mathbf{J}\ c_1\ p_1{}^{\ell_2}}$$

Fig. 10. Indistinguishability rules (equality)

The shape of the indistinguishability rules for the propositional equality constructs in Figure 10 are unsurprising: rules I-Eq, I-Refl, and I-J assert indistinguishability of the terms given indistinguishability of each pair of subterms. The constraints $\ell_0 \le \ell$ in rule I-Eq and $\ell_2 \le \ell$ in rule I-J are required due to their presence in rules Wt-Eq and Wt-J respectively.

4 Syntactic Metatheory

In this section, we develop the syntactic metatheory of DCOI$^{\omega}$, ultimately culminating in subjection reduction, *i.e.* type preservation of typing with respect to parallel reduction. Because the syntactic metatheory has already been established for DCOI by Liu et al. [2024b], here we only state without proof the most important lemmas, some of which are used in the next section for the semantic

metatheory. Nevertheless, everything has been mechanized in Coq, and the development can be found in the supplementary materials under the `proofs/theories` directory.

Some lemmas are presented as admissible derivation rules of the corresponding judgment, which are written with a double horizontal bar instead of a single bar. Their corresponding mechanizations are annotated where mentioned in the prose.

4.1 Preliminaries

As indistinguishability and definitional equality involve a level context, we require a level checking judgment $\boxed{\Xi \vdash a : \ell}$ on terms to be able to prove some substitution lemmas. The judgment specifies the dependency level that a term may be assigned. In later sections, we will see that semantic typing and the logical relation are defined over level checked terms. The rules for the judgment correspond exactly to stripping out all typing information from the typing rules, leaving only the levels. We omit them here, but they are listed in the supplementary appendix A.4.

$$\textsc{L-subst}\ \frac{\Xi, x : \ell_0 \vdash b : \ell \qquad \Xi \vdash a : \ell_0}{\Xi \vdash b[a/x] : \ell} \qquad \textsc{L-sub}\ \frac{\Xi \vdash a : \ell_0 \qquad \ell_0 \leq \ell_1}{\Xi \vdash a : \ell_1}$$

$$\textsc{I-subst}\ \frac{\Xi, x : \ell_0 \vdash b_0 \equiv^{\ell} b_1 \qquad \Xi \vdash a : \ell_0}{\Xi \vdash b_0[a/x] \equiv^{\ell} b_1[a/x]} \qquad \textsc{I-cong}\ \frac{\Xi, x : \ell_0 \vdash b : \ell \qquad \Xi \vdash a_0 \equiv^{\ell}_{\ell_0} a_1}{\Xi \vdash b[a_0/x] \equiv^{\ell} b[a_1/x]}$$

$$\textsc{I-down}\ \frac{\Xi \vdash a \equiv^{\ell_0} b \qquad \Xi \vdash a \equiv^{\ell_1} c}{\Xi \vdash a \equiv^{\ell_0 \wedge \ell_1} b} \qquad \textsc{Wt-subst}\ \frac{\Gamma \vdash a :^{\ell_0} A \qquad \Gamma, x :^{\ell_0} A \vdash b :^{\ell} B}{\Gamma \vdash b[a/x] :^{\ell} B[a/x]} \qquad \textsc{Wt-sub}\ \frac{\Gamma \vdash a :^{\ell_0} A \qquad \ell_0 \leq \ell_1}{\Gamma \vdash a :^{\ell_1} A}$$

Fig. 11. Admissible rules for level checking, indistinguishability, and type checking

The level checking and indistinguishability judgments satisfy some standard properties, listed as admissible rules in Figure 11. L-subst and I-subst permit substituting in a level-checked term of the appropriate level in place of a variable of that level in the context. Indistinguishability also satisfies a congruence property I-cong where indistinguishable terms may be substituted into a level-checked term. Level checking is subsumptive (L-sub): the level of a level-checked term may be raised. Indistinguishability satisfies a downgrade property (I-down), where knowing that a term is indistinguishable from another at two different levels permits concluding that it is also indistinguishable at the meet of the levels.

The key lemmas of parallel reduction are the diamond property and confluence, which state that if a term reduces to two different terms, those two terms themselves must converge back to the same term. The proof uses the notion of complete development by Takahashi [1995].

Lemma 4.1 (Diamond). *If $a \Rightarrow b_0$ and $a \Rightarrow b_1$, then there exists c s.t. $b_0 \Rightarrow c$ and $b_1 \Rightarrow c$.*

Lemma 4.2 (Confluence). *If $a \Rightarrow^* b_0$ and $a \Rightarrow^* b_1$, then there exists c s.t. $b_0 \Rightarrow^* c$ and $b_1 \Rightarrow^* c$.*

Injectivity of type constructors are trivial to prove in our system since definitional equality is directly defined in terms of indistinguishability and parallel reduction, both of which are injective by definition. Extra effort is needed to show that the definitional equality is transitive, which requires downgrade (I-down), Confluence, and Simulation.

Lemma 4.3 (Transitivity of equality). *If $\Xi \vdash a \equiv b$ and $\Xi \vdash b \equiv c$, then $\Xi \vdash a \equiv c$.*

4.2 Type Preservation

Like level checking, type checking admits substitution (Wt-subst) and subsumption (Wt-sub), as stated in Figure 11. Due to rule Wt-Conv, deducing the well-typedness of subterms of a well-typed term doesn't follow immediately from inversion and requires separate generation lemmas, which are standard and omitted here. One notable consequence is that if **refl** is a propositional equality between a and b at some observer level ℓ_0, then they are definitionally equal at that level. Finally, we are able to prove type preservation, which proceeds by induction on the typing derivation.

Lemma 4.4. *If* $\Gamma \vdash \mathbf{refl} :^{\ell} a =^{\ell_0} b$, *then* $|\Gamma| \vdash a \Leftrightarrow^{\ell_0} b$.

Theorem 4.5 (Type preservation). *If* $a \Rightarrow b$ *(or* $a \Rightarrow^* b$*) and* $\Gamma \vdash a :^{\ell} A$ *then* $\Gamma \vdash b :^{\ell} A$.

5 Semantic Metatheory

In this section, we use a logical predicate to show weak normalization and consistency for DCOI$^{\omega}$. Due to DCOI$^{\omega}$'s flexible treatment of type conversion and its support for large elimination, we cannot directly reuse the proofs found in Abel and Scherer [2012]; Geuvers [1994] through embedding. We structure the section as follows. In Section 5.1, we present the definition of the logical predicate and its properties. In Section 5.2, we define valid substitutions and semantic typing judgments based on the logical predicate and then prove the fundamental theorem, which states that all syntactically well-typed terms must be semantically well-typed, from which we derive weak normalization and consistency. In Section 5.3, we combine weak normalization and the standardization theorem to recover a decidable type conversion algorithm.

5.1 A Logical Predicate for DCOI$^{\omega}$

To characterize semantically well-behaved terms through the logical predicate, we use the notions of neutral forms $\boxed{\mathbf{ne}\ a}$ and normal forms $\boxed{\mathbf{nf}\ a}$, which characterize sets of terms that are free of redexes. Our definition of neutral forms excludes terms like $(\lambda x^{\mathsf{H}}.\ a)\ b^{\mathsf{L}}$ that are stuck because of mismatching levels. Furthermore, we consider **absurd** a as neutral as long as a is in normal form since there are no constructors of the $\bot$ type. The definitions of neutral and normal forms are otherwise standard and can be found in the supplementary appendix A.5. Weakly normalizing terms are then terms which reduce to a normal form.

Definition 5.1 (Weakly normalizing terms). Given a term a, if there exists some term b such that $a \Rightarrow^* b$ and **nf** b (resp. **ne** b), then we say that a weakly normalizes to a normal (resp. neutral) form, which we denote as $\boxed{\mathbf{wnf}\ a}$ (resp. $\boxed{\mathbf{wne}\ a}$).

The following lemma captures the idea that neutral and normal forms are free of redexes.

Lemma 5.2. *If* **nf** a *or* **ne** a, *then for all* b *such that* $a \Rightarrow b$, *we have* $a = b$.

Figure 12 gives the definition of our logical predicate. The omitted rules for the naturals can be found in appendix A.6. Similar to Abel and Scherer [2012]'s semantic interpretation of types, we define our logical predicate $\boxed{[\![\Xi \vDash_i A]\!] \searrow S}$ as an inductive relation recursively defined over the universe level i, where S is a family of terms indexed by dependency levels. The judgment $\boxed{a \in S(\ell)}$ indicates that a is a term belonging to the set in S indexed by ℓ. When constructing indexed sets, we use $\boxed{\ell.\{a \mid \ldots\}}$ to indicate a family of sets parameterized by ℓ. The judgment $[\![\Xi \vDash_i A]\!] \searrow S$ states that under level context Ξ and universe level i, the type A is interpreted as the family of sets S.

For concision, we introduce *semantic inhabitance* to avoid mentioning the type well-formedness side condition when talking about a term inhabiting the set interpretation of a type.

Definition 5.3 (Semantic inhabitance). Given a context Ξ and a level ℓ, we say that a *semantically inhabits* the type A at level ℓ if there exists some i and S such that $[\![\Xi \vDash_i A]\!] \searrow S$ and $a \in S(\ell)$.

(Auxiliary notation)

$$b \in \hat{\Pi}(S_A^{\ell_0}, R^{\ell}) \triangleq \forall a, S_B, a \in S_A(\ell_0) \to R(a, S_B) \to b\, a^{\ell_0} \in S_B(\ell)$$
$$c \in \hat{\Sigma}(S_A^{\ell_0}, R^{\ell}) \triangleq \exists a, b, c \Rightarrow^* (a^{\ell_0}, b) \wedge a \in S_A(\ell_0) \wedge (\forall S_B, R(a, S_B) \to b \in S_B(\ell))$$
$$p \in (\Xi \vdash a \mathrel{\hat{=}}^{\ell_0} b) \triangleq \Xi \vdash p : \ell \wedge p \Rightarrow^* \mathbf{refl} \wedge \Xi \vdash a \Leftrightarrow^{\ell_0} b$$
$$[\![\Xi \vDash_j A]\!] \triangleq \exists S, [\![\Xi \vDash_j A]\!] \searrow S$$

$\boxed{[\![\Xi \vDash_i A]\!] \searrow S}$ *(Logical predicate)*

SWT-NE
$$\frac{\mathbf{ne}\ A}{[\![\Xi \vDash_i A]\!] \searrow \ell.\{a \mid \Xi \vdash a : \ell \wedge \mathbf{wne}\ a\}}$$

SWT-EQ
$$\frac{\mathbf{nf}\ a \qquad \mathbf{nf}\ b}{[\![\Xi \vDash_i a =^{\ell_0} b]\!] \searrow \ell.\{p \mid p \in (\Xi \vdash a \mathrel{\hat{=}}^{\ell_0} b) \vee \mathbf{wne}\ p\}}$$

SWT-PI
$$\frac{[\![\Xi \vDash_i A]\!] \searrow S_A \qquad \forall a, a \in S_A(\ell_0) \to \exists S_B, R(a, S_B) \qquad \forall a, S_B, a \in S_A(\ell_0) \to R(a, S_B) \to [\![\Xi \vDash_i B[a/x]]\!] \searrow S_B}{[\![\Xi \vDash_i \Pi x :^{\ell_0} A.\, B]\!] \searrow \ell.\{b \mid \Xi \vdash b : \ell \wedge b \in \hat{\Pi}(S_A^{\ell_0}, R^{\ell})\}}$$

SWT-EMPTY
$$\frac{}{[\![\Xi \vDash_i \bot]\!] \searrow \ell.\{a \mid \Xi \vdash a : \ell \wedge \mathbf{wne}\ a\}}$$

SWT-SIGMA
$$\frac{[\![\Xi \vDash_i A]\!] \searrow S_A \qquad \forall a, a \in S_A(\ell_0) \to \exists S_B, R(a, S_B) \qquad \forall a, S_B, a \in S_A(\ell_0) \to R(a, S_B) \to [\![\Xi \vDash_i B[a/x]]\!] \searrow S_B}{[\![\Xi \vDash_i \Sigma x :^{\ell_0} A.\, B]\!] \searrow \ell.\{c \mid \Xi \vdash c : \ell \wedge (c \in \hat{\Sigma}(S_A^{\ell_0}, R^{\ell}) \vee \mathbf{wne}\ c)\}}$$

SWT-STEP
$$\frac{A \Rightarrow B \qquad [\![\Xi \vDash_i B]\!] \searrow S}{[\![\Xi \vDash_i A]\!] \searrow S}$$

SWT-UNIT
$$\frac{}{[\![\Xi \vDash_i \top]\!] \searrow \ell.\{a \mid \Xi \vdash a : \ell \wedge (a \Rightarrow^* \mathbf{tt} \vee \mathbf{wne}\ a)\}}$$

SWT-UNIV
$$\frac{j < i}{[\![\Xi \vDash_i \mathcal{U}_j]\!] \searrow \ell.\{A \mid \Xi \vdash A : \ell \wedge [\![\Xi \vDash_j A]\!]\}}$$

Fig. 12. Logical predicate

In rules SWT-PI and SWT-SIGMA, the metavariable R represents a relation between terms and the indexed family of terms S. Ignoring the conclusion of the rules, SWT-PI and SWT-SIGMA share the same preconditions ensuring their well-formedness: for every term a that semantically inhabits the type A at level ℓ_0, the type $B[a/x]$ is well-formed. In the function case, the definition of $b \in \hat{\Pi}(S_A^{\ell_0}, R^{\ell})$ specifies the semantic inhabitants of a function type as the set of terms that maps well-behaved inputs to well-behaved outputs.

The logical predicate takes in a level context Ξ as a parameter to ensure that S only includes terms that level check at the indexed level. This property is formally stated as follows and can be proven by straightforward induction on the logical predicate.

LEMMA 5.4 (ESCAPE). *If* $[\![\Xi \vDash_i A]\!] \searrow S$ *and* $a \in S(\ell)$, *then* $\Xi \vdash a : \ell$.

We refer to Lemma 5.4 as the escape lemma as it is reminiscent of the escape lemmas by Abel and Scherer [2012] and Adjedj et al. [2024], though instead of escaping from the logical predicate to syntactic typing, we only require the much weaker level checking, which will be necessary to show that indistinguishable types have the same interpretation.

Similar to its syntactic counterpart, the logical predicate satisfies the subsumption property, proven by straightforward induction using L-SUB.

LEMMA 5.5 (SUBSUMPTION FOR THE LOGICAL PREDICATE). *Suppose* $[\![\Xi \vDash_i A]\!] \searrow S$. *If* $\ell_0 \le \ell_1$ *and* $a \in S(\ell_0)$, *then* $a \in S(\ell_1)$.

Next, we show some properties which will be useful for proving the fundamental theorem. We omit some uses of syntactic lemmas about reduction and indistinguishability in the proof sketches.

The logical predicate satisfies the following inversion principles. They are not immediately obtainable by inverting the derivation because we close our interpretation with parallel reduction in rule SWT-STEP. For brevity, we show only the cases for equality, function, and empty types.

LEMMA 5.6 (INVERSION OF THE LOGICAL PREDICATE (SELECTED)).
(1) If $[\![\Xi \vDash_i a =^{\ell_0} b]\!] \searrow S$, *then* $S = \ell.\{p \mid \Xi \vdash p : \ell \wedge (p \in (\Xi \vdash a \doteq^{\ell_0} b) \vee \mathbf{wne}\ p)\}$.
(2) If $[\![\Xi \vDash_i \Pi x :^{\ell_0} A.\, B]\!] \searrow S$, *then there exist* S_A *and* R *such that following hold:*
- $S = \ell.\{b \mid \Xi \vdash b : \ell \wedge b \in \hat{\Pi}(S_A^{\ell_0}, R^{\ell})\}$;
- $[\![\Xi \vDash_i A]\!] \searrow S_A$;
- $\forall a, a \in S_A(\ell_0) \longrightarrow \exists S_B, R(a, S_B)$; *and*
- $\forall a, S_B, a \in S_A(\ell_0) \longrightarrow R(a, S_B) \longrightarrow [\![\Xi \vDash_i B[a/x]]\!] \searrow S_B$.

(3) If $[\![\Xi \vDash_i \bot]\!] \searrow S$, *then* $S = \ell.\{a \mid \Xi \vdash a : \ell \wedge \mathbf{wne}\ a\}$.

PROOF. Immediate by induction on the derivation of the logical predicate. □

The set of terms that semantically inhabit a type is closed by expansion (*i.e.* backward reduction).

LEMMA 5.7 (BACKWARD CLOSURE). *Suppose* $[\![\Xi \vDash_i A]\!] \searrow S$ *and* $\Xi \vdash a : \ell$. *If* $a \Rightarrow b$ *and* $b \in S(\ell)$, *then* $a \in S(\ell)$.

PROOF. By induction on the derivation of the logical predicate. The SWT-UNIV case requires SWT-STEP, the backward closure property baked into the definition of the logical predicate. All other cases are trivial. □

The logical predicate satisfies cumulativity: if a type has an interpretation at a lower level, then it must also have the same interpretation at a higher level.

LEMMA 5.8 (CUMULATIVITY). *If* $[\![\Xi \vDash_i A]\!] \searrow S$ *and* $i \leq j$, *then* $[\![\Xi \vDash_j A]\!] \searrow S$.

PROOF. By induction on the derivation of $[\![\Xi \vDash_i A]\!] \searrow S$ and transitivity of $\leq$. □

Rule SWT-STEP closes the interpretation of types under expansion. We can also show that the interpretation of types is preserved under forward reduction through the following lemma.

LEMMA 5.9 (REDUCTION PRESERVES MEANING). *If* $[\![\Xi \vDash_i A]\!] \searrow S$ *and* $A \Rightarrow B$, *then* $[\![\Xi \vDash_i B]\!] \searrow S$.

PROOF. By induction on the derivation of $[\![\Xi \vDash_i A]\!] \searrow S$. Rule SWT-STEP is the only interesting case, using the Diamond property to reconcile two potentially distinct reductions from A. □

Next, we prove that the logical predicate is functional. That is, given a context Ξ and a universe level i, a type A can correspond to at most one interpretation S.

LEMMA 5.10 (FUNCTIONALITY). *If* $[\![\Xi \vDash_i A]\!] \searrow S_0$ *and* $[\![\Xi \vDash_i A]\!] \searrow S_1$, *then* $S_0 = S_1$.

PROOF. By induction on the derivation of $[\![\Xi \vDash_i A]\!] \searrow S_0$. The SWT-STEP case is immediate by Lemma 5.9. For all other cases, the conclusion follows by applying the inversion properties (Lemma 5.6) on $[\![\Xi \vDash_i A]\!] \searrow S_1$. □

The logical predicate is functional not only for types that are syntactically equal, but also for types that are indistinguishable. This property is needed to fully justify the soundness of the type conversion rule, since the type conversion involves both reduction and indistinguishability.

LEMMA 5.11 (FUNCTIONALITY FOR INDISTINGUISHABLE TYPES). *If* $[\![\Xi \vDash_i A]\!] \searrow S_0$, $[\![\Xi \vDash_i B]\!] \searrow S_1$, *and* $\Xi \vdash A \equiv^{\ell} B$ *for some* ℓ, *then* $S_0 = S_1$.

Proof. The argument is similar to the one for Lemma 5.10. We start by induction on the derivation of $[\![\Xi \vDash_i A]\!] \searrow S_0$. For case SWt-Step, we need both Lemma 5.9 and Simulation. For other cases, we apply the inversion properties on $[\![\Xi \vDash_i B]\!] \searrow S_1$.

The cases for function and pair types both require one extra step to complete the proof. Consider the case where $A = \Pi x :^{\ell_0} A_0.\, B_0$ and $B = \Pi x :^{\ell_0} A_1.\, B_1$. The induction hypothesis says that for every term a that semantically inhabits A_0, if a type C is indistinguishable from $B_0[a/x]$, then C and $B_0[a/x]$ have the same meaning. To complete the proof, we use the induction hypothesis and instantiate the variable C with the type $B_1[a/x]$ and prove that $B_0[a/x]$ is indistinguishable from $B_1[a/x]$. From the premise that $\Pi x :^{\ell_0} A_0.\, B_0$ and $\Pi x :^{\ell_0} A_1.\, B_1$ are indistinguishable, we already know that B_0 and B_1 are indistinguishable. To prove that $B_0[a/x]$ and $B_1[a/x]$ are indistinguishable, we apply I-subst. This requires a to be checked at level ℓ_0, which follows from a semantically inhabiting A_0 via Escape. □

The last main property is adequacy, which establishes the connection between the inhabitants of the logical predicate and terms that weakly normalize to normal or neutral forms, but we first need some useful facts about normal forms. Following Geuvers [1994], we introduce a dummy constant $\mathbf{d}$ to the set of terms for presentation purposes. The constant $\mathbf{d}$ is a neutral term that level checks under any context and parallel reduces only to itself. First, reductions and normal forms are preserved when we undo substitutions involving $\mathbf{d}$.

Lemma 5.12. *If $a[\mathbf{d}/x] \Rightarrow b$, then there exists some term b_0 such that $b = b_0[\mathbf{d}/x]$ and $a \Rightarrow b_0$.*

Lemma 5.13. *If $\mathbf{nf}\ a[\mathbf{d}/x]$ (resp. $\mathbf{ne}\ a[\mathbf{d}/x]$), then $\mathbf{nf}\ a$ (resp. $\mathbf{ne}\ a$).*

From Lemmas 5.12 and 5.13, we derive the following corollary about weak normalization.

Lemma 5.14 (Antirenaming for weak normalization). *If $\mathbf{wnf}\ a[\mathbf{d}/x]$, then $\mathbf{wnf}\ a$.*

In Geuvers [1994], the $\mathbf{d}$ constant can be defined as syntactic sugar for a fresh variable. However, this is not possible in DCOI$^\omega$ because of the scoping constraint imposed by level checking. Instead, we define $\mathbf{d}$ as $\mathbf{absurd}\ \bot$. Note that the term appearing in the body of **absurd** is flexible: all we need is a term in normal form that has no reference to variables, so we do not violate scoping.

Finally, we state and prove the adequacy lemma.

Lemma 5.15 (Adequacy). *If $[\![\Xi \vDash_i A]\!] \searrow S$, then $\mathbf{wnf}\ A$ and the following hold:*

- *If $a \in S(\ell)$, then $\mathbf{wnf}\ a$;*
- *If $\Xi \vdash a : \ell$ and $\mathbf{wne}\ a$, then $a \in S(\ell)$.*

Proof. By induction on the derivation of $[\![\Xi \vDash_i A]\!] \searrow S$. The only interesting part of the proof is showing normalization for well-formed function and pair types. We focus on function types, and the proof for pair types is almost identical.

In the function case, we have $A = \Pi x :^{\ell_0} A_0.\, B_0$. By the induction hypothesis, we already know $\mathbf{wnf}\ A_0$ and for all terms a that semantically inhabit A_0, the term $B_0[a/x]$ is weakly normalizing. To show that $\Pi x :^{\ell_0} A_0.\, B_0$ is weakly normalizing, it suffices to show that B_0 is weakly normalizing.

By the induction hypothesis for A_0, we also know that if $\Xi \vdash a : \ell$ and a is a neutral term, then a semantically inhabits A_0. We pick $\mathbf{d}$ for a and deduce that $B_0[\mathbf{d}/x]$ is weakly normalizing. By Lemma 5.14, we conclude that B_0 is also weakly normalizing. □

5.2 Soundness

To define semantic well-typedness, we need to first define the notion of valid substitutions, also referred to as valuations. While we continue to use named variables rather than de Bruijn indices,

in this section we frame substitution in terms of *substitution maps* (or simply *substitutions* for short) and simultaneous substitutions.

Definition 5.16 (Substitution maps). A substitution map ρ is a mapping from variables to terms. We use $\mathrm{dom}(\rho)$ to denote the variables in the domain of ρ, $\rho(x)$ the term to which ρ maps x, $\cdot$ the identity substitution map, $(\rho, a/x)$ the extension of ρ by a mapping from x to a, and $a[\rho]$ the simultaneous substitution of the variables $x \in \mathrm{dom}(\rho)$ in a by $\rho(x)$. The singleton substitution map a/x is then defined as $(\cdot, a/x)$, the single extension of the identity substitution map.

Definition 5.17 (Valid substitutions). A substitution map is a valid substitution from Γ to Δ, written $\boxed{\Delta \vDash \rho : \Gamma}$, if for every $x :^{\ell} A \in \Gamma$, $|\Delta| \vdash \rho(x) : \ell$ holds, and for all i, S such that $[\![|\Delta| \vDash_i A[\rho]]\!] \searrow S$, we have $\rho(x) \in S(\ell)$.

By definition, every term $\rho(x)$ in a valid substitution map ρ from Γ to Δ level checks with respect to the erased context $|\Delta|$. Applying the substitution then yields a term that level checks.

Lemma 5.18. *If* $|\Gamma| \vdash a : \ell$ *and* $\Delta \vDash \rho : \Gamma$, *then* $|\Delta| \vdash a[\rho] : \ell$.

Proof. Immediate by iterating the substitution principle for level checking (L-subst). □

Valid substitutions satisfy the following admissible structural rules.

Lemma 5.19 (Structural rules for valid substitutions).

(1) For every Γ, $\Gamma \vDash \cdot : \Gamma$ *holds; and*

(2) Given $\Delta \vDash \rho : \Gamma$, *if a semantically inhabits* $A[\rho]$ *at level* ℓ_0 *under level context* $|\Delta|$, *then* $\Delta \vDash \rho, a/x : \Gamma, x :^{\ell_0} A$ *holds.*

Proof. (1) is easily justified by the part of the adequacy lemma that states level-checked neutral terms (variables in Γ in this case) semantically inhabit any well-formed types.

(2) requires Functionality and Cumulativity to convert the fact that $a \in S(\ell_0)$ for the specific S from the semantic inhabitance to the universal statement that $a \in S_0(\ell_0)$ for arbitrary j and S_0 such that $[\![|\Delta| \vDash_j A[\rho]]\!] \searrow S_0$, as needed by the definition of valid substitutions. □

Next, we define semantic typing based on valid substitutions and the logical predicate.

Definition 5.20 (Semantic well-typedness). A term a is semantically well typed with type A at level ℓ under context Γ, written $\boxed{\Gamma \vDash a :^{\ell} A}$, if for every ρ, Δ, given $\Delta \vDash \rho : \Gamma$, the term $a[\rho]$ semantically inhabits the type $A[\rho]$ under the level context $|\Delta|$.

Valid substitutions and semantic well-typedness both satisfy weakening properties, which align with the intuition that a substitution map for a context Γ also closes over any prefixes of Γ.

Lemma 5.21 (Weakening for valid substitutions and semantic well-typedness).

- *If* $\Delta \vDash \rho : \Gamma, x :^{\ell} A$, *then* $\Delta \vDash \rho : \Gamma$.
- *If* $\Gamma \vDash a :^{\ell} A$, *then* $\Gamma, x :^{\ell_0} A \vDash a :^{\ell} A$.

Proof. Immediate by the definitions of valid substitutions and semantic well-typedness. □

Given a valid substitution map ρ from Γ to Δ, the validity of ρ does not necessarily imply the well-formedness of the types in Γ. Instead, the existence of interpretations for types in Γ is separately provided by the semantic context well-formedness judgment $\boxed{\vDash \Gamma}$.

$$\frac{}{\vDash \cdot}\ \textsc{Swf-nil} \qquad \frac{\vDash \Gamma \qquad \Gamma \vDash A :^{\ell} \mathcal{U}_i}{\vDash \Gamma, x :^{\ell} A}\ \textsc{Swf-cons}$$

We now specify the fundamental theorem and provide proof sketches for the interesting cases.

Theorem 5.22 (Fundamental theorem). *If* $\Gamma \vdash a :^{\ell} A$ *then* $\Gamma \vDash a :^{\ell} A$*; if* $\vdash \Gamma$ *then* $\vDash \Gamma$.

Proof. By mutual induction on syntactic typing and well-formedness. Despite the complex setup of the logical predicate, semantic typing, and their properties, the proof of the fundamental theorem is straightforward and is similar to the proofs by logical relations for simply typed languages (*e.g.* Abel et al. [2019]). For example, the Wt-App case proves itself since the idea of valid inputs to valid outputs is baked into the SWt-Pi case of the logical predicate. For other cases involving eliminators such as Wt-J and Wt-Let, the scrutinee either reduces to a normal form (*i.e.* **refl** or (a^{ℓ_0}, b)) or a neutral term. In the former, the conclusion follows by Backward closure and the induction hypothesis. In the latter, we know that the elimination form evaluates to a neutral form and thus semantically inhabits the type by Adequacy. The Wt-Conv case is justified by the preservation of meaning under expansion (SWt-Par), reduction (Lemma 5.9), and indistinguishability (Lemma 5.11). The Wt-Var case requires weakening for semantic typing (Lemma 5.21) to bring the semantically well-formed type to the right scope. □

Corollary 5.23 (Weak normalization for well-typed terms). *If* $\Gamma \vdash a : A$*, then* **wnf** a.

Proof. Immediate by the Fundamental theorem and Lemma 5.19 (1). □

Corollary 5.24 (Logical consistency). *The judgment* $\cdot \vdash a :^{\ell} \bot$ *is not derivable for any* a *or* ℓ.

Proof. Immediate by the Fundamental theorem, Lemma 5.19 (1), and Lemma 5.6 (3). □

5.3 Decidability of Type Conversion

We can use the normalization theorem (Corollary 5.23) to recover a decision procedure for type conversion. Let $\boxed{a \rightsquigarrow^{lo} b}$ be the leftmost-outermost reduction relation for DCOI$^{\omega}$.

Lemma 5.25 (Standardization). *If* $a \Rightarrow^{*} b$ *and* **nf** b*, then* $a \rightsquigarrow^{lo} b$.

Its proof, which we omit here, is adapted from the standardization proof for the untyped lambda calculus by Takahashi [1995] and Accattoli et al. [2019].

Corollary 5.26 (Normalization is decidable). *If* **wnf** a*, then its normal form can be computed.*

Proof. By Standardization and the fact that leftmost-outermost reduction is deterministic, we can repeatedly apply leftmost-outermost reduction to find the normal form for a. □

Lemma 5.27 (Indistinguishability is decidable). *If the lattice admits a decidable comparison operator, then indistinguishability is decidable. That is, given a level context* Ξ*, terms* A *and* B*, and a level* ℓ*, there is an algorithm that terminates with* `true` *or* `false` *depending on whether* $\Xi \vdash A \equiv^{\ell} B$.

Proof. The algorithm, whose definition we omit, simply recurs over the structure of A. The correctness proof mirrors the definition of the algorithm and proceeds by induction on A. The variable case requires that the comparison $\ell_0 \leq \ell$ be decidable. □

Definition 5.28 (Algorithm for type conversion). Given a level context Ξ and two well-formed types A and B, the algorithm deciding their equality is as follows.

(1) Run the algorithm from Corollary 5.26 on A and B to obtain normal forms A_0 and B_0.
(2) Compute the minimal level ℓ at which A_0 and B_0 can be level checked.
(3) Return the result of the decision procedure from Lemma 5.27 on the relation $\Xi \vdash A_0 \equiv^{\ell} B_0$.

In (3), Simulation justifies the use of normal forms A_0 and B_0 before we check indistinguishability, while downgrade (I-down) justifies the use of the lowest possible level ℓ computed from (2).

The only missing piece is the decidability of the algorithm for computing levels in (2). While conceptually simple, the algorithm depends on the structure of the lattice. For bottomless lattices, there is no minimal level we can return for values such as $\bot$ and $\mathcal{U}_i$. Instead, we need to conjure up a level ℓ that can ignore the most terms possible when comparing A and B. While there are other sufficient conditions for inferring the level ℓ, such as the size of the lattice being countable, we focus on the correctness of the algorithm for a lattice with a bottom element.

Theorem 5.29 (Decidability of type conversion with bottom). *Given $\Gamma \vdash a :^{\ell} A$ and $\Gamma \vdash B : \mathcal{U}_i$, the algorithm described in Definition 5.28 decides the relation $|\Gamma| \vdash A \equiv B$ if DCOI$^{\omega}$ is instantiated with a lattice with a bottom.*

Independent of the underlying lattice structure, the indexed version of definitional equality is always decidable since it allows us to skip the level inference from step (2).

Theorem 5.30 (Decidability of indexed type conversion). *Given a context Ξ, syntactically well-formed types A and B, and a level ℓ, $\Xi \vdash A_0 \Leftrightarrow^{\ell} B_0$ is decidable.*

Note that $\Xi \vdash A_0 \Leftrightarrow^{\ell} B_0$ is equivalent to the preconditions of rule E-Conv. For finite or countable lattices, the decision procedure from Theorem 5.30 can be iterated to recover the decidability of conversion in rule Wt-Conv.

6 Discussion and Future Work

6.1 Scrutinee Level Constraints

The level of eliminator scrutinees must be bounded by the level of the overall eliminator. In particular, in rule Wt-J, the level ℓ_2 of an equality proof p must be bounded by that of its elimination $\mathbf{J}\, c\, p^{\ell_2}$, and in rule Wt-Let, the level ℓ_1 of a pair a must be bounded by that of its elimination $\mathbf{let}\,(x^{\ell_0}, y) = a^{\ell_1}\ \mathbf{in}\ b$. The bound is required to prove the Simulation property. Suppose that scrutinees could instead be typed at any level. First, the indistinguishability rules would need to be updated so that scrutinees are indistinguishable if their level is higher than that of the overall eliminator, just like in rule I-App. The modified premises are boxed below.

$$\text{I-J-bad}\quad \frac{\boxed{\Xi \vdash p_0 \equiv^{\ell}_{\ell_2} p_1} \qquad \Xi \vdash c_0 \equiv^{\ell} c_1}{\Xi \vdash \mathbf{J}\, c_0\, {p_0}^{\ell_2} \equiv^{\ell} \mathbf{J}\, c_1\, {p_1}^{\ell_2}}$$

$$\text{I-Let-bad}\quad \frac{\boxed{\Xi \vdash a_0 \equiv^{\ell}_{\ell_1} a_1} \qquad \Xi, x : \ell_0, y : \ell_1 \vdash b_0 \equiv^{\ell} b_1}{\Xi \vdash \mathbf{let}\,(x^{\ell_0}, y) = {a_0}^{\ell_1}\ \mathbf{in}\ b_0 \equiv^{\ell} \mathbf{let}\,(x^{\ell_0}, y) = {a_1}^{\ell_1}\ \mathbf{in}\ b_1}$$

As concrete examples, by rule I-J-bad, supposing $\mathbf{J}\, c\, p^{\mathsf{H}}$ were well typed in context Γ at level L with $\Gamma \vdash p :^{\mathsf{H}} a =^{\mathsf{L}} a$, we can conclude that $|\Gamma| \vdash \mathbf{J}\, c\, \mathbf{refl}^{\mathsf{H}} \equiv^{\mathsf{L}} \mathbf{J}\, c\, p^{\mathsf{H}}$. The problem is that the left-hand side reduces to c, while the right-hand side does not necessarily reduce to something indistinguishable from c, thereby violating simulation. The same issue arises with pairs, where we may be able to conclude by rule I-Let-bad that $|\Gamma| \vdash \mathbf{let}\,(x^{\mathsf{H}}, y) = (a^{\mathsf{H}}, b)^{\mathsf{H}}\ \mathbf{in}\ c \equiv^{\mathsf{L}} \mathbf{let}\,(x^{\mathsf{H}}, y) = d^{\mathsf{H}}\ \mathbf{in}\ c$, and the left-hand side reduces to $c[a/x, b/y]$, while the right-hand side again does not necessarily reduce to something indistinguishable from it. Even if the context were empty, simulation still cannot be proven without knowing *a priori* that a closed proof of equality p normalizes to **refl**, or that a closed term of a pair type normalizes to a pair.

The one exception is the eliminator for the empty type, whose scrutinee may be well typed at any arbitrary level in rule Wt-Absurd. Since **absurd** b has no reduction rules beyond the congruence rule P-Absurd, ignoring b in rule I-Absurd will not violate simulation.

If we try to enforce the scrutinee's level to be exactly the overall level ℓ, we would not be able to prove subsumption (Wt-sub), since the level of the scrutinee appears in the context. In both rules Wt-J and Wt-Let, this occurs when type checking the motive C, which abstracts over the

scrutinee. Supposing that both the level of the scrutinee and the overall level are ℓ, subsumption states that given a derivation at level ℓ, we need to construct a derivation at a higher level ℓ'. For **J** and **let**, we are given a derivation concluding something of the shape $\Gamma, x :^{\ell} \cdot \vdash C :^{\ell_0} \mathcal{U}_i$, but we require one of the shape $\Gamma, x :^{\ell'} \cdot \vdash C :^{\ell_0} \mathcal{U}_i$, which we are unable to conclude.

6.2 Typed Definitional Equality and Relational Semantic Models

Following Liu et al. [2024b], we formulate DCOI$^{\omega}$ as an instantiation of Barendregt's Pure Type Systems with a predicative universe hierarchy. A variation of PTS, referred to as Pure Type Systems with Judgmental Equality by Adams [2006], replaces the untyped conversion rule from PTS with a typed equality judgment. While untyped conversion is used in practice by theorem provers such as Coq, typed equality enables more expressive η laws that otherwise cannot be expressed.

Siles and Herbelin [2012] proves the equivalence between typed and untyped Pure Type Systems in the absence of η laws. Would a similar equivalence hold between DCOI$^{\omega}$ and a variant with typed equality? It turns out that even defining judgmental equality for DCOI$^{\omega}$ is challenging. Consider the following typed congruence rule for application, which naïvely adds type annotations to rule I-App.

$$\text{I-AppTy}\quad \frac{\Gamma \vdash b_0 \equiv^{\ell} b_1 \in \Pi x :^{\ell_0} A.\, B \qquad \Gamma \vdash a_0 \equiv^{\ell}_{\ell_0} a_1 \in A}{\Gamma \vdash b_0\ a_0{}^{\ell_0} \equiv^{\ell} b_1\ a_1{}^{\ell_0} \in B[a_0/x]}$$

Ignoring the levels, rule I-AppTy is exactly how the application equivalence rule is defined for a PTS with judgmental equality. However, rule I-AppTy breaks the type correctness property, which states that $\Gamma \vdash a \equiv^{\ell} b \in A$ implies $\Gamma \vdash a :^{\ell} A$ and $\Gamma \vdash b :^{\ell} A$.

When $\ell_0 \not\leq \ell$, the function arguments are not observable, so we do not need to check the equality between a_0 and a_1 other than their respective well-typedness. However, DCOI$^{\omega}$ does not prohibit the type B from depending on the variable x as the type $\Pi x :^{\ell_0} A.\, B$ can be well formed at a level unrelated to the observer level ℓ by Regularity. As a result, there is no guarantee that $B[a_0/x]$ and $B[a_1/x]$ are indistinguishable types.

As a simple counterexample, we can pick b_0 and b_1 to be level L polymorphic identity functions with a level H (thus irrelevant) type parameter. Instantiating a_0 and a_1 with distinct types, we end up with two distinct function types $B[a_0/x] = {}^{\mathsf{L}}a_0 \to a_0$ and $B[a_1/x] = {}^{\mathsf{L}}a_1 \to a_1$. Type correctness breaks as soon as we add input type annotations to functions. Abel and Scherer [2012] observed a similar issue in the design of their type-directed conversion algorithm. Their workaround is to simply reject programs whose types depend on irrelevant arguments.

The problem of extending DCOI$^{\omega}$ with judgmental equality also directly relates to the problem of defining a relational model for justifying properties such as extensionality, or a generalized semantic notion of noninterference that goes beyond our syntactic Simulation property. Since logical relations are indexed by types, when defining the interpretation for functions, we run into the same issue where run-time irrelevant arguments that are vacuously related by guarded equality may result in distinguishable types after being substituted into the output type.

Rather than patching rule I-AppTy with side conditions until syntactic properties such as type correctness are satisfied, a more pressing problem is whether rule I-AppTy is semantically sound. The counterexample above for polymorphic identity functions shows that $b_0\ a_0{}^{\ell_0}$ and $b_1\ a_1{}^{\ell_0}$ do not necessarily *syntactically* inhabit both $B[a_0/x]$ and $B[a_1/x]$. However, semantically, two instantiated identity functions with different type annotations do not behave differently at run time.

In fact, we believe it is impossible to construct an example where $b_0\ a_0{}^{\ell_0}$ and $b_1\ a_1{}^{\ell_0}$ do not behave like terms from both $B[a_0/x]$ and $B[a_1/x]$. To be more adversarial, suppose that B contains a large elimination so that $B[a_0/x]$ and $B[a_1/x]$ are incompatible types. If $b_0\ a_0{}^{\ell_0}$ and $b_1\ a_1{}^{\ell_0}$ evaluate to different head forms, then semantic soundness would fail. However, knowing that $b_0\ a_0{}^{\ell_0}$ and $b_1\ a_1{}^{\ell_0}$

are indistinguishable at ℓ *already* rules out the case where they have different head forms. Therefore, such a counterexample cannot be derived.

As a result, despite the technical issues demonstrated in this section, we are optimistic that our logical predicate can be further extended to a relational model to justify more complex properties.

6.3 Casting with Irrelevant Equalities

Equality type has the special property that there is only one canonical form **refl**. The fundamental theorem guarantees that a closed proof of equality can only evaluate to **refl**. As a result, the elimination form $\mathbf{J}\, c\, p^{\ell}$ can be erased to contain only the body c since no information is gained from matching against a singleton type.

Despite the erasability of equality proofs at runtime, DCOI does not allow casting a relevant term using an irrelevant proof. In Section 6.1, we see that this restriction is necessary for the simulation property to hold since an open equality proof can normalize to either **refl** or to a neutral term.

Do we need to evaluate equality proofs? Suppose that we modify the operational semantics of DCOI$^{\omega}$ by replacing P-JRefl and the congruence rule with the single rule $\mathbf{J}\, c\, p^{\ell} \Rightarrow c$ so that the J eliminator is erased during evaluation without having to reduce p. In a system with this rule, because the equality proof is discarded during evaluation, it can be ignored by indistinguishability and still validate the simulation property.

The addition of this rule does not invalidate logical consistency or the fact that our equational theory is consistent. However, in the presence of this rule, we lose decidable type conversion. With a proof of $A = A \to A$ in the context, one can define a well-typed term that reduces to the diverging term $(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)$, since the J eliminator no longer gets stuck at a bogus equality proof. This rule also violates type preservation: given a term a of type **Nat** and a bogus assumption x of type $\mathbf{Nat} =^{\ell} \bot$, we have $\mathbf{J}\, a\, x^{\ell} \Rightarrow a$ where the former is of type $\bot$ but the latter is of type **Nat**.

Werner [2008] attempts to address these problems while still allowing the irrelevant treatment of equality proofs. In his system, the elimination form for equalities is annotated with the terms appearing in the equality. Reduction does not examine the equality proof, but instead proceeds when the terms on both sides of the equality are convertible. Werner's rule requires that the terms in the equality be relevant, which is not always desirable.

It is possible to recover decidable type conversion for a non-strict elimination rule through annotations [Liu and Weirich 2023], or by exploring beyond intensional equality types [Pujet and Tabareau 2022]. In future work, we would like to explore the extension of DCOI$^{\omega}$ with these alternative versions of propositional equality.

7 Related Work

Dependent types and irrelevance. Many dependent type systems include a notion of irrelevance, although the term has been used inconsistently and refers to either run-time irrelevance or compile-time irrelevance. The former refers to the erasure of computationally-irrelevant terms during compilation to optimize code. The latter erases or ignores irrelevant terms during type conversion to accept more programs. Here, we liberally use relevance tracking to refer to both concepts.

The Implicit Calculus of Constructions (ICC) [Miquel 2001], its decidable variant ICC* [Barras and Bernardo 2008], and the Erasure Pure Type System (EPTS) [Mishra-Linger and Sheard 2008] support irrelevance by including two abstraction forms, distinguishing functions that do and do not use their arguments. ICC and ICC* ensure that irrelevant variables do not syntactically appear in relevant computations, whereas EPTS uses a design based on Pfenning [2001] to prevent variables marked with the irrelevant modality from being used relevantly. Despite the difference in mechanisms, the languages are similar in expressiveness. In the type of an irrelevant abstraction, $\forall x{:}A.B$, the parameter x may relevantly appear in B. Thus, there is no way to express that the output *type* of a

function does not depend on its input. ICC and ICC* allow compile-time irrelevance by erasing irrelevant components before conversion. But, these systems cannot support irrelevent projections as erasure would allow conversion between distinct types.

In contrast, Pfenning's modal type system [Pfenning 2001] and its extension by Abel and Scherer [2012] forbids irrelevant parameters from appearing in both the body of an abstraction *and* its result type. As a result, irrelevent function types cannot model type polymorphism, and the system does not include irrelevant projections. Furthermore, as pointed out by Tejiščák [2020], Pfenning's style of relevance tracking prevents the definition of type families indexed by an irrelevant argument. One such example is the binary type `Bin n`, whose index `n` keeps track of the natural number it corresponds to. To avoid exponential overhead during execution, the index `n` should be marked as irrelevant. However, this is impossible with Pfenning's system as irrelevant arguments are convertible, but `Bin n` and `Bin m` are different types for different `m` and `n`.

Liu et al. [2024b] design DCOI by observing that the failure of encoding irrelevant type indices is due to a fixed view of relevance. They generalize the notion of irrelevance as a relative notion indexed by an observer level. A term and its type can have different views of irrelevance. The indistinguishability judgment uses the observer level to determine which components of a term can be discarded during type conversion. This allows DCOI, and thus DCOI$^\omega$, to encode `Bin` while still allowing compile-time irrelevance for arguments that are irrelevant for both terms and types.

Mishra-Linger and Sheard [2008] distinguish between extrinsic and intrinsic relevance tracking. The former determines relevance purely based on how terms are used. All systems we have discussed so far, including DCOI and DCOI$^\omega$, are extrinsic systems.

An example of the intrinsic approach is the **Prop** sort, introduced by Paulin-Mohring [1989], which identifies computationally irrelevant terms that should be erased during program extraction. The soundness of erasure during program extraction (referred to as *external* or *post-mortem erasure* by Abel and Scherer [2012]) is ensured by a form of dependency tracking. In Coq, for example, only singleton types (*e.g.* equality proofs) or empty types in **Prop** can be destructed to produce run-time relevant terms. As discussed in Section 6.3, Coq's **Prop** sort is limited to run-time irrelevance for decidability of type checking. Werner [2008]'s type theory with a proof-irrelevant **Prop** fails to satisfy normalization when **Prop** is impredicative, a result later shown by Abel and Coquand [2020].

Gilbert et al. [2019] introduce the language sMLTT, an extension to MLTT with a proof-irrelevant sort **sProp**. Similar to Werner's system, inhabitants of a type $A : \mathbf{sProp}$ are treated as definitionally equal. They introduce an alternative criterion to singleton elimination, which characterizes inductive types that can be safely eliminated from **sProp** into the relevant universe while preserving decidability. They show the consistency of their design for both predicative and impredicative variants of **sProp** through a syntactic translation to Extensional Type Theory. Similar to DCOI$^\omega$, their system allows the empty type to be eliminated from **sProp** to the relevant universes.

Bracket types and squash types [Awodey and Bauer 2004; Mendler 1990] introduce bracket and squash operators that remove computational information from types. Inhabitants of a squashed type are definitionally equal. To ensure consistency, elimination is restricted so that computationally-relevant terms cannot access the hidden content. Squash types can also be viewed as an intrinsic treatment of irrelevance since a squashed type is always irrelevant regardless of usage.

Despite the differences between extrinsic and intrinsic irrelevance, they bear many similarities. For example, irrelevant function types can be implemented as regular function types with a squashed input type in the systems by Awodey and Bauer [2004]; Gilbert et al. [2019]; Mendler [1990]. For the other direction, the squash operator can be implemented in the form of a weak irrelevant dependent pair [Abel and Scherer 2012], similar to our encoding of the box type.

Dependent types and counting. Quantitative type systems provide a type of fine-grained static analysis that counts the number of times a computation uses each of its inputs. For flexibility, counting uses an abstract semiring, generalizing type systems for bounded linearity, information flow, and differential privacy. Using a boolean semiring, quantitative type systems can track dependency and perform external erasure.

McBride [2016] extends a dependently typed system with usage tracking for computationally relevant terms. However, the typing judgment in his system fails to admit the substitution property, an issue that is later addressed by [Atkey 2018] in the design of the Quantitative Type Theory (QTT). The typing judgment in QTT takes the form $\Gamma \vdash M :^{\sigma} T$ where σ ranges over the constants 0 and 1 from the semiring, reminiscent of Pfenning's $\div$ and : modalities. Similar to the system by McBride [2016], usage tracking does not apply to types and is disabled in type formation rules.

GraD [Choudhury et al. 2021] later gives a more uniform design where type formation rules are presented in the style of a PTS and share the same judgment form as typing for terms. This allows one to analyze usage information for a type expression. Unlike QTT, the type soundness of resource tracking in GraD is proven using a syntactic approach through heap semantics. No instances of GraD have been proven consistent as a logic.

Graded Modal Dependent Type Theory (GRTT) [Moon et al. 2021] is a predicative type theory that tracks resource usage separately for types and terms. The fine-grained usage tracking allows one to recover parametricity by rejecting undesired type- or term-level access of variables. Moon et al. [2021] take advantage of the usage information to optimize type checking through their prototype implementation. However, this optimization, while reminiscent of the proof irrelevance feature discussed in Section 7, is not formalized. Type conversion in GRTT is oblivious of the usage information. The lack of interaction between modalities and type conversion allows them to recover strong normalization by adapting strong normalization for CC by Geuvers [1994].

Abel et al. [2023] introduce a graded modal dependent type theory that also allows tracking usage for both terms and types. Since modalities do not interact with type conversion, they reuse the technique in Abel et al. [2017] to mechanically prove metatheoretic results including normalization and decidable type conversion for a standard dependently typed language that is free of usage tracking. Grading is then defined separately from the typing judgment and their semantics is refined to account for usage. In DCOI$^{\omega}$, since type conversion relies on indistinguishability, properties about dependency tracking must be developed before we define our semantic model.

Dependent types and indistinguishability. The design of DCOI [Liu et al. 2024b] and its instantiation DCOI$^{\omega}$ is directly inspired by the Dependent Dependency Calculus (DDC) [Choudhury et al. 2022]. Similar to DCOI, DDC supports both run-time and compile-time irrelevance. However, it achieves this goal by combining two different mechanisms. For run-time irrelevance, DDC uses a mechanism similar to EPTS [Mishra-Linger and Sheard 2008] to allow erasable irrelevant type indices. For compile-time irrelevance, it uses the resurrection mechanism from Abel and Scherer [2012]; Pfenning [2001]. While their system includes a indistinguishability relation similar to ours, their type conversion rule performs conversion at some fixed level C. Liu et al. [2024b] show that as a result, it is difficult to generalize DDC's conversion rule to support indistinguishability at different levels, necessary for the elimination of indexed equality types.

Type-theory in color (TTC) [Bernardy and Guilhem 2013] indexes the judgments of the Calculus of Constructions with sets of colors, called taints, to support run-time erasure and internalize parametricity. TTC does not include compile-time irrelevance. While DCOI$^{\omega}$ does not internalize parametricity, TTC's dependency tracking through taints is similar to dependency levels in DCOI$^{\omega}$. However, in TTC, variables can be used only if their taint *exactly* matches the current taint. Functions keep track of a set of anti-taints, colors that the argument is oblivious to, in addition to

the set of colors that the argument depends on. The erasure operator $\lfloor a \rfloor_i$ erases all components dependent on the color i from the term a and also serves the purpose of interpreting types as predicates. The ability to shift the view by choosing which color to erase is like the observer level from the indistinguishability judgment of DCOI$^\omega$. Furthermore, like Abel and Scherer [2012]; Pfenning [2001] and unlike DCOI$^\omega$, in TTC, the same taint is used for checking a term and the well-formedness of its type. In DCOI$^\omega$, the observer level between terms and types are decoupled.

Sterling and Harper [2022] interpret information flow through the new perspective of phase distinctions. For each dependency level ℓ, there is a corresponding proposition $\langle \ell \rangle$, which, if inhabited, indicates that program lacks clearance to access data at level ℓ. Data sealed behind level ℓ thus becomes indistinguishable. In future work, we would like to explore whether we can reformulate DCOI from this perspective, and how we can leverage the synthetic methods described in Sterling and Harper [2021, 2022] to give a more semantic account of our non-interference result.

Dependent types and mechanized logical relations. Barras [1996] proves strong normalization for the Calculus of Constructions in Coq using a generalized notion of Girard's reducibility candidates [Girard et al. 1989] to model the system. Similar to Geuvers [1994], Barras leverages the fact that CC does not have large eliminations, so that terms can be erased from type level computations. The technique is not applicable to systems with large eliminations.

Mechanizing a logical relation for a dependently typed language with large eliminations is difficult because it requires one to specify the set of semantically well-formed types while simultaneously interpreting those types as sets of terms. Therefore, Abel et al. [2017] use induction–recursion in Agda to define a Kripke-style logical relation for a dependent type theory with judgmental equality and large eliminations. The inductive part specifies the set of valid types and the recursive part assigns meanings to the types. Abel et al. [2023] builds off their mechanization to prove decidable type checking, consistency, and preservation for a graded modal type theory.

Similar to our approach, Adjedj et al. [2024]; Anand and Rahli [2014]; Wieczorek and Biernacki [2018] define their logical relations as an inductive type in Coq and then prove functionality *a posteriori*. Compared to Abel et al. [2017]'s inductive–recursive model, this approach requires less power from the metalanguage [Adjedj et al. 2024]. However, an impredicative **Prop** sort is required to model an object language with infinitely many universe levels [Anand and Rahli 2014].

8 Conclusion

In this work, we define DCOI$^\omega$, an instantiation of DCOI with a predicative universe hierarchy, and extended with more expressive elimination forms. We use a syntactic logical predicate to show that DCOI$^\omega$ satisfies normalization, logical consistency, and decidable type conversion in addition to the syntactic soundness results previously established; these results are all mechanized in Coq. Logical consistency makes DCOI$^\omega$ suitable for both programming and internalized verification.

In future work, we plan to further explore the use of DCOI$^\omega$ as the foundation for a practical system. First, we will show that the annotations employed by our prototype implementation support full decidable type checking in the style of a bidirectional system Adjedj et al. [2024]; Dunfield and Krishnaswami [2013]. In this context we can then explore various extensions of the system, including dependency level inference, dependency level quantification, and subtyping. We also plan to investigate applications of dependency tracking beyond irrelevance, such as safe interoperability, information flow and staged computation.

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 Zenodo [Liu et al. 2024a].

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, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019), e19. https://doi.org/10.1017/S0956796819000170

Andreas Abel and Thierry Coquand. 2020. Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality. *Logical Methods in Computer Science* Volume 16, Issue 2 (June 2020), 14:1–14:5. https://doi.org/10.23638/lmcs-16(2:14)2020

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP, Article 220 (Aug. 2023), 35 pages. https://doi.org/10.1145/3607862

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158111

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012), 1–36. https://doi.org/10.2168/lmcs-8(1:29)2012

Beniamino Accattoli, Claudia Faggian, and Giulio Guerrieri. 2019. Factorization and Normalization, Essentially. In *Programming Languages and Systems*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 159–180. https://doi.org/10.1007/978-3-030-34175-6_9

Robin Adams. 2006. Pure type systems with judgemental equality. *Journal of Functional Programming* 16, 2 (2006), 219–246.

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. https://doi.org/10.1145/3636501.3636951

Agda Development Team. 2023. *Agda*. Programming Logic Group. https://wiki.portal.chalmers.se/agda/Main/HomePage

Abhishek Anand and Vincent Rahli. 2014. Towards a Formally Verified Proof Assistant. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 27–44. https://doi.org/10.1007/978-3-319-08970-6_3

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

Steven Awodey and Andrej Bauer. 2004. Propositions as [Types]. *Journal of Logic and Computation* 14, 4 (08 2004), 447–471. https://doi.org/10.1093/logcom/14.4.447

Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 462–490. https://doi.org/10.1017/s0956796800020025

Bruno Barras. 1996. *Coq en Coq*. Ph. D. Dissertation. INRIA.

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379. https://doi.org/10.1007/978-3-540-78499-9_26

Jean-Philippe Bernardy and Moulin Guilhem. 2013. Type-theory in color. *SIGPLAN Not.* 48, 9 (Sept. 2013), 61–72. https://doi.org/10.1145/2544174.2500577

Pritam Choudhury, Harley 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, Article 50 (Jan. 2021), 32 pages. https://doi.org/10.1145/3434331

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.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., USA.

Coq Development Team. 2019. *The Coq Proof Assistant*. INRIA. https://doi.org/10.5281/zenodo.3476303

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction (Lecture Notes in Computer Science, Vol. 9195)*. Springer International Publishing, Cham, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/2500365.2500582

Herman Geuvers. 1994. A short and flexible proof of strong normalization for the calculus of constructions. In *International Workshop on Types for Proofs and Programs*. Springer, Berlin, Heidelberg, 14–38. https://doi.org/10.1007/3-540-60579-7_2

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, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7. Cambridge University Press, Cambridge.

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

Yiyun Liu, Jonathan Chan, and Stephanie Weirich. 2024a. *Artifact associated with Consistency of a Dependent Calculus of Indistinguishability*. https://doi.org/10.5281/zenodo.14252132

Yiyun Liu and Stephanie Weirich. 2023. Dependently-Typed Programming with Logical Equality Reflection. *Proc. ACM Program. Lang.* 7, ICFP, Article 210 (Aug. 2023), 37 pages. https://doi.org/10.1145/3607852

Conor McBride. 2016. *I got plenty o'nuttin'*. Springer, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Nax Paul Mendler. 1990. Quotient types via coequalizers in Martin-Löf type theory. In *First workshop on logical frameworks*. INRIA, France, 349.

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, Los Alamitos, CA, USA, 221–230. https://doi.org/10.1109/lics.2001.932499

Loïc Pujet and Nicolas Tabareau. 2022. Observational equality: now for good. *Proc. ACM Program. Lang.* 6, POPL, Article 32 (Jan. 2022), 27 pages. https://doi.org/10.1145/3498693

Vincent Siles and Hugo Herbelin. 2012. Pure type system conversion is always typable. *Journal of Functional Programming* 22, 2 (2012), 153–180. https://doi.org/10.1017/S0956796812000044

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

Jonathan Sterling and Robert Harper. 2022. Sheaf Semantics of Termination-Insensitive Noninterference. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Masako Takahashi. 1995. Parallel Reductions in λ-Calculus. *Information and Computation* 118, 1 (1995), 120–127. https://doi.org/10.1006/inco.1995.1057

Matúš Tejiščák. 2020. A dependently typed calculus with pattern matching and erasure inference. *Proc. ACM Program. Lang.* 4, ICFP, Article 91 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408973

Benjamin Werner. 2008. On the strength of proof-irrelevant type theories. *Logical Methods in Computer Science* 4 (Sept. 2008), 1–20. Issue 3. https://doi.org/10.2168/lmcs-4(3:13)2008

Paweł Wieczorek and Dariusz Biernacki. 2018. A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 266–279. https://doi.org/10.1145/3167091

Received 2024-07-11; accepted 2024-11-07