Internalizing Extension Tracking with Lattices of Type Theories

Jonathan Chan

21 November 2024

Abstract

While many proof assistants are founded upon common theoretical ground, they will also feature further extensions and axioms that augment their reasoning power. The individual theories based on each extension are individually sound, but how type checkers handle their interactions at compile time is not formally described. Furthermore, the way proof assistants currently enable extensions can be too coarse and imprecise when it comes to guarantees about the features a top-level definition may use.

I propose using the Dependent Calculus of Indistinguishability (DCOI) [Liu et al., 2024] for formalizing granular and precise extension tracking. DCOI is a type system with dependency tracking, where terms and variables are assigned dependency levels alongside their types. These dependency levels form a lattice that describes which levels are permitted to access what. In this project, each extension would have a corresponding dependency level, and the lattice would describe how extensions are permitted to interact.

1 Introduction

Dependent type theories form the theoretical foundations of many proof assistants used for automating and mechanizing theorems in areas ranging from pure mathematics to modelling real programming languages. Flavours of theories such as Martin-Löf Type Theory (MLTT) [Martin-Löf, 1972] and the Calculus of Inductive Constructions (CIC) [Pfenning and Paulin-Mohring, 1990] form the basis of some of the most widely used proof assistants. By the Curry–Howard correspondence, where propositions correspond to types and proofs correspond to terms, the core of proof assistants are language implementations. As such, they share many of the same components as implementations of typical programming languages, such as a parser, a type checker, and a compiler.

Like many language implementations, proof assistants will incorporate optional language features that influence the behaviour of its inner components. For programming languages, these features may be compiler optimizations, static analyses that restrict the language to enable these optimizations, or type system extensions that increase what the type checker accepts. Because the primary use case of proof assistants is to type check mechanized proofs, their language extensions are largely the latter, each of which alters the underlying type theory.

Consequently, the effects of extensions in proof assistants are nonlocal, as opposed to, say, marking a function as tail recursive, which does not affect the behaviour of functions around it. Using a particular type theoretic extension in a module, for instance, changes the reasoning principles that are or aren't allowed in that module, which can infect other modules that import it. Furthermore, there exist extensions that are mutually incompatible because together they would violate logical consistency of the type theory. In Section 2, I describe a few such extensions found in select proof assistants, and the ways some combinations are incompatible.

Thus when proof assistants provide mechanisms for language extensions, they are careful to track the usage of these extensions to rule out inconsistencies. The tracking done by the type checker is *external* to the type system: within the language itself, one cannot assert that a definition is permitted to depend on a particular extension, nor that it is guaranteed *not* to use it. Furthermore, while each extension may have been independently verified to be consistent, there are no formal descriptions of interactions between parts of code that use different sets of extensions. Ultimately, what we are missing is a framework for describing fine-grained control of proofs and programs across multiple type theories.

Luckily, there do exist frameworks for describing fine-grained control of proofs and programs across multiple *dependency levels*, which exist within the same type system, but are stratified by some hierarchical structure of what can be thought of intuitively as permission levels. In particular, I have worked on the Dependent Calculus of Indistinguishability (DCOI) [Liu et al., 2024] and its variant DCOI$^\omega$ [Liu et al., 2025], the latter of which is a type theory with dependency tracking, whose key properties I describe in Section 3.

I propose that DCOI can be used as a basis for a framework which internalizes extension tracking by stratifying their corresponding type theories into hierarchies of dependency levels, where compatibility between extensions maps to the intuitive notion of permission. In Section 4, I speculate on the details of this mapping, lay out the objectives for such a framework, and list possible first steps towards accomplishing them. There is much prior and related work that this project relies on and relates to, which I divide into work I have personally contributed to (Section 5) and other work in this space (Section 6).

2 Proof Assistant Extensions in Practice

To look at extensions in practice, let us focus on three popular proof assistants: Rocq [Coq Development Team, 2022], Agda [Norell, 2007], and Lean [de Moura et al., 2015]. Broadly speaking, they are all based on variants of MLTT or CIC, and have dependent functions, type universes, and inductive types, which are sufficient to encode a wide variety of logical propositions and proofs.

Each of these proof assistants, however, include features that extend the power of their foundations; below are a few notable extensions.

- **Impredicativity**: Rocq and Lean, being based on CIC, feature a universe Prop of propositions such that a quantification (*i.e.* dependent function type) $\forall(x : A).\, B$ is a proposition if B is a proposition, regardless of the universe in which A lives, which may be larger than Prop. Inductive types may also be defined in Prop, which permits its constructors to have argument types in universes larger than Prop. Such inductives are said to be *large*.
- **Definitional proof irrelevance**: The inhabitants of propositions in Lean are definitionally equal and thus treated as interchangeable during type checking; these propositions are said to be *strict*. Rocq has a separate SProp universe of proof-irrelevant propositions, and Agda has a predicative hierarchy Prop_i of such universes [Gilbert et al., 2019].

- **Uniqueness of identity proofs** (UIP): Agda's default pattern matching behaviour admits that inhabitants of the same propositional equality are themselves propositionally equal, which is proven via defining **Axiom K** [Streicher, 1993], a computational eliminator for propositional equalities of the form $a \equiv a$.
- **Strong elimination**: Also known as *large* elimination, an element of an inductive datatype can be destructed or eliminated into a type at a larger universe. That is, a term whose type is in Type_i can be eliminated to return a term whose type is in Type_j for some $j > i$. For proofs of propositions in Prop, large elimination corresponds to eliminating into any non-proposition.

Each of these increase expressivity by allowing more types to be stated and inhabited. Impredicativity allows for self-referential propositions by quantifying over Prop. For instance, given a proof that all propositions imply their double negation,

$$dn : \forall (P : \mathsf{Prop}).\, P \to \neg\neg P,$$

the double negation of this proposition itself holds as well.

$$dn\ (\forall (P : \mathsf{Prop}).\, P \to \neg\neg P)\ dn : \neg\neg(\forall (P : \mathsf{Prop}).\, P \to \neg\neg P)$$

In the predicative setting, a quantification over a universe $\mathsf{Type}_0 : \mathsf{Type}_1$ itself has type Type_1, and in particular,

$$\Pi(A : \mathsf{Type}_0).\, A \to \neg\neg A : \mathsf{Type}_1,$$

so an element of this type may not be applied to the type itself.

Definitional proof irrelevance is useful to avoid having to prove equalities explicitly. Consider a relation on two naturals asserting the usual less-than relation, along with a type of bounded naturals from which the natural contained can be recovered.

$$\begin{aligned}
\cdot \leq \cdot &: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Prop} \\
BNat &: \mathsf{Nat} \to \mathsf{Type}_0 \\
bNat &: \Pi(n\ m : \mathsf{Nat}).\, n \leq m \to BNat\ m \\
getNat &: \Pi(m : \mathsf{Nat}).\, BNat\ m \to \mathsf{Nat} \\
getNat\ m\ (bNat\ n\ m\ p) &\rightsquigarrow n
\end{aligned}$$

A desirable property of bounded naturals is that two bounded naturals are equal if their contained naturals are equal.

$$eqBNat : \forall (m : \mathsf{Nat})(b_1 b_2 : \mathsf{BNat}\ m).\, getNat\ m\ b_1 \equiv getNat\ m\ b_2 \to b_1 \equiv b_2$$

Trying to prove this by destructing the bounded naturals as $(bnat\ n_1\ m\ p_1)$ and $(bnat\ n_2\ m\ p_2)$, while we have an equality $e : n_1 \equiv n_2$ by reduction of $getNat$, we do *not* have a proof of $p_1 \equiv p_2$[1]. Depending on how $\cdot \leq \cdot$ is implemented, it may be possible to prove propositionally that any two inequality proofs are equal. Alternatively, if propositions are definitionally proof irrelevant, the inequality proofs can effectively be ignored, and rewriting the goal by e is sufficient.

UIP is similarly useful to avoid reasoning about equalities between equalities when the only canonical proof of an equality is reflexivity, especially in settings without proof irrelevance. While UIP does augment the reasoning power of the type theory, there are inductive

[1] Technically, this requires a proof of the equality where p_1 has been transported across the equality e so that it has the same type as p_2.

types whose equality proofs are already propositionally equal. In particular, if a type has decidable equality, *i.e.* $(x \equiv y) \vee \neg(x \equiv y)$ for any given x, y of that type, then its equalities are themselves equal [Hedberg, 1998].

Naturally, not all types are proof-irrelevant, such as the booleans, since true and false are distinct. However, proving $\neg(\mathsf{true} \equiv \mathsf{false})$ requires strong elimination to lift the booleans to propositions of truthhood $\top$ and falsehood $\bot$.

$$\begin{aligned} &\mathit{lift} : \mathsf{Bool} \to \mathsf{Prop} \\ &\mathit{lift} \coloneqq \lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ \top\ \mathsf{else}\ \bot \end{aligned}$$

This is a strong elimination because it returns a type, or equivalently because its return type is a universe. To complete the proof, given a proof of $\mathsf{true} \equiv \mathsf{false}$, $\top \equiv \bot$ holds by congruence over *lift*, across which the trivial proof $\mathtt{tt} : \top$ can be transported to $\bot$.

Some of these extensions are hidden behind an option flag. To use strict Prop, Rocq requires the flag `Allow StrictProp`, while Agda requires the option `{-# OPTIONS --prop #-}`. While not inherently part of Rocq's type theory, Axiom K is axiomatized in the standard library as `Coq.Logic.Eqdep.eq_rect_eq`. Many other axioms are included in standard libraries to further augment reasoning power; below are a few notable examples.

- **Function extensionality** propositionally equates two functions if they are pointwise equal; in Lean as `funext` and in Rocq as `functional_extensionality`.
- **Propositional extensionality** propositionally equates two propositions if they are biïmplicated; in Lean as `propext` and in Rocq as `propositional_extensionality`. Propositional proof irrelevance is a consequence of propositional extensionality.
- **Univalence** asserts an equivalence between propositional equality and equivalence, *i.e.* given two types A, B, the equivalence $(A \equiv B) \simeq (A \simeq B)$ holds [Univalent Foundations Program, 2013]. There are several ways to define equivalence; the idea is that it captures a propositionally proof-irrelevant isomorphism.
- **Excluded middle** asserts that all propositions are either true (inhabited) or false (uninhabited); in Lean as `em` and in Rocq as `Coq.Logic.Classical_Prop.classic`. This is equivalent to several other principles, including **double negation elimination** $(\forall (A : \mathsf{Prop}).\, \neg\neg A \to A)$ and **Peirce's law** $(\forall (A\ B : \mathsf{Prop}).\, ((A \to B) \to A) \to A)$.

Function extensionality, propositional extensionality, and univalence are principles that extend propositional equality such that more things are equal. Univalence together with proof irrelevance implies propositional extensionality, since biïmplicated propositions are isomorphic by irrelevance, and univalence gives an equality from the isomorphism. Univalence alone also implies function extensionality by a more complex argument [Univalent Foundations Program, 2013, Chapter 4.9].

One application of these new equalities may be encountered when encoding a function as a relation whose functionality is proven *a posteriori*. This is a frequent pattern in proof assistants, as induction relations often have better ergonomic support than dependently-typed reasoning over functions. Consider an encoding of a two-place predicate over a representation of types *Ty* and terms *Tm* — a function from *Ty* to predicates $\mathit{Tm} \to \mathsf{Prop}$ — as a relation:

$$R : \mathit{Ty} \to (\mathit{Tm} \to \mathsf{Prop}) \to \mathsf{Prop}.$$

Such a relation could be a *logical relation* [Tait, 1967] used to model typed lambda calculi, where a *Ty* is interpreted as a set of *Tm*s. Functionality of R demonstrates that *Ty*s have unique interpretations. To show that R is functional, *i.e.*

$$\forall (A : \mathit{Ty})(P\ Q : \mathit{Tm} \to \mathsf{Prop}).\, R\ A\ P \to R\ A\ Q \to P \equiv Q,$$

it suffices to show that $\forall(a : \mathit{Tm}).\, P\ a \leftrightarrow Q\ a$, since $\forall(a : \mathit{Tm}).\, P\ a \equiv Q\ a$ follows from propositional extensionality, and finally $P \equiv Q$ from function extensionality.

The disadvantage of axiomatic equalities is that substitutions will not further reduce on them, which can make reasoning about the substituted terms difficult. There are type theories beyond MLTT and CIC that are designed so that these principles are instead provable theorems, such as cubical type theories [Bezem et al., 2019; Cohen et al., 2018; Angiuli et al., 2017, 2021] and Cubical Agda [Vezzosi et al., 2019] for univalence, and observational type theory [Altenkirch and McBride, 2006; Altenkirch et al., 2007; Pujet and Tabareau, 2022, 2023, 2024] for function and propositional extensionality.

Excluded middle, double negation elimination, and other equivalent propositions are *classical* reasoning principles that do not hold intuitionistically. Nevertheless, because a large majority of mathematics is done classically, many communities mechanizing mathematics will freely use classical principles. In particular, Lean's mathematical library mathlib [mathlib Community, 2020] contains proofs that rely on classical axioms, and tactics such as `tauto` will automatically apply classical reasoning. There is work towards constructively integrating classical principles into type theory consistently, typically in the form of calculi with control operators [Parigot, 1992; Curien and Herbelin, 2000; Miquey, 2017; Cong and Asai, 2018].

One has to be careful, however, that a chosen set of features and axioms do not render the type theory logically inconsistent and thus useless for proving. A number of them are known to be incompatible with one another; below are a few such combinations.

- Strong elimination is inconsistent for large impredicative inductives. Hook and Howe [1986] show that impredicative dependent pairs with pair projections, which can correspond to strong elimination, are inconsistent. Coquand [1992] also demonstrates the inconsistency with an inductive type U with a single constructor of type $\forall(X : \mathsf{Prop}).\, (X \to U) \to U$.
- Strong elimination is also inconsistent for inductive propositions when Prop is proof irrelevant. As seen above, strong elimination suffices to show that $\neg(\mathsf{true} \equiv \mathsf{false})$. If Bool is defined in a proof-irrelevant Prop, $\mathsf{true} \equiv \mathsf{false}$ would hold by definition, which is a contradiction.
- Strong elimination is once again inconsistent for inductive propositions in the presence of impredicativity and classical principles such as excluded middle. A modern implementation of the construction by Barbanera and Berardi [1996], such as `Coq.Logic.Berardi` in Rocq's standard library, uses excluded middle to derive propositional proof irrelevance, which can be used as above to derive a contradiction.
- UIP is inconsistent with univalence. Intuitively, univalence produces an equality between two types given an equivalence between them, and there are types that are equivalent in multiple, provably different ways, so there are equalities between them that are provably different, thus violating UIP. Concretely, Bool is equivalent to itself in two different ways, either by mapping booleans to themselves or to their negation, so there are distinct proofs of $\mathsf{Bool} \equiv \mathsf{Bool}$ [Univalent Foundations Program, 2013, Example 3.1.9].

Figure 1 illustrates some of the aforementioned relationships between extensions. The arrows point from one theory to a greater encompassing theory; for instance, a theory with propositional extensionality extends the equalities of a theory with a universe of propositions, and a theory with univalence can derive function extensionality. At the top of the graph, the dotted arrows indicate the incompatibility of a theory that implies UIP with one that contains univalence: there is no possible encompassing theory.

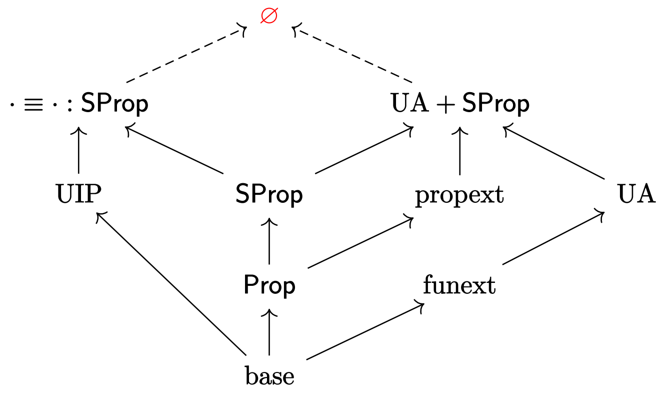

Figure 1: A compatibility graph of theories with impredicative Prop, proof irrelevance (SProp), UIP, univalence (UA), function extensionality (funext), propositional extensionality (propext), and (in)compatible combinations.

To prevent inconsistencies, features may be hidden behind option flags, or disallowed entirely. Rocq, Lean, and Agda all disallow strong elimination for inductive propositions in Prop and SProp, with the exception of *syntactic subsingletons*, which are inductives that syntactically have at most one inhabitant, such as $\top$, $\bot$, and conjunction of propositions. While Rocq's impredicative Prop universe is not proof irrelevant, strong elimination is still forbidden even for inductives that aren't large to allow the use of classical principles. There is a compiler flag `-impredicative-set` to enable impredicativity for Set while still allowing strong elimination of small impredicative inductives, but this flag is no longer supported. In Agda, the `--with-K` flag enables Axiom K and the `--without-K` flag disables it, while univalence can be proven as part of the features enabled by Cubical Agda using the `--cubical` flag. There is also a `--safe` flag which disallows, among other combinations, having both `--with-K` and `--cubical`.

There is also some limited ability in tracking the usage of features and axioms. In Rocq, the axioms and unsafe flags used by a definition can be listed using the command `Print Assumptions`, while in Lean, the axioms can be listed using `#print axioms`. In Agda, along with checking for inconsistent option combinations, `--safe` will also ensure the absence of any `postulate`s. Since option flags can be enabled at module-level granularity, there is also a notion of *(co)infective* flags: an infective flag used in one module must be used by all modules that depend on that module, while a coinfective flag used in one module may only depend on modules that also use that flag.

While users of these proof assistants have been getting along fine with these compiler tools for tracking features and axioms for the past decade or more, there is plenty of room of improvement; I have identified three shortcomings these systems.

- There is no way of safeguarding against a feature or an axiom; that is, it cannot be guaranteed generally that a particular definition does *not* use something. If a definition can be guaranteed not to use some given feature, then it is safe to be used by other definitions that are incompatible with that feature. For instance, ensuring that a proof does not use Axiom K means that it may be used by another proof that does use univalence.
- The scope of feature flags is too coarse. They range from project-level compiler flags down to module-level option flags, but even this level of granularity is not the ap-

propriate one: modules are intended for organizing definitions by semantic content, rather than by the collection of features they happen to all use. This prevents reuse of a definition in one module inside another module if they happen to have incompatible features, even if that particular definition does not depend on any features at all.

- Tracking features and axioms is only a compiler tool and is not formally described along with the underlying type theories themselves. Furthermore, there is no formal description of the interactions between the different theories that result from including various features and axioms.

An ideal system for enabling and disabling features and axioms, then, should track which ones are and aren't used, at least at the definition level, and it should be formally described with the type theory as a whole. This suggests that a system for dependency tracking would be a good starting point. Therefore, I propose that the Dependent Calculus of Indistinguishability (DCOI) [Liu et al., 2024], a type system that incorporates dependency tracking and dependent types, could be a suitable framework for tracking multiple type theories.

3 A Primer on DCOI

DCOI is a Pure Type System (PTS) [Barendregt, 1991] augmented with dependency tracking Abadi et al. [1999]. An instantiation of DCOI's PTS rules and axioms governing how types and terms interact is DCOI$^\omega$ [Liu et al., 2025], which has dependent types and a predicative universe hierarchy, making it suitable as a foundation for theorem proving. In a typing judgement, dependency tracking appears as annotations on both the variables in the context, to indicate how they may be used by the term being typed, and alongside the type of the term, to indicate how the term itself may be used. As an example, consider the following derivable typing judgement for a constant function.

$$A :^{\mathsf{H}} \mathsf{Type} \vdash \lambda x^{\mathsf{L}}\, y^{\mathsf{H}}.\, x :^{\mathsf{L}} A^{\mathsf{L}} \to A^{\mathsf{H}} \to A$$

The concrete levels used here are low (L) and high (H) where $\mathsf{L} < \mathsf{H}$, though DCOI is general over any meet-semilattice. In the context of security flow, these levels correspond to low- and high-security computations where low-security computations may not inspect the values of high-security ones. They can also be thought of in terms of computational irrelevance, where something marked as computationally irrelevant (H) must not play a part in the execution of relevant programs (L), and may even be erased away after compilation.

This constant function at low, which returns its first argument x and ignores its second argument y, must therefore mark x and its type as low to return it, and marking y and its type as high guarantees that it could not return it. While the body of the low function cannot return a high argument, its type *can* depend on a high term, demonstrated by the high-annotated type A in the context, which is used in the type of the function. The intuition is that A does not play a part in the run-time execution of the constant function, but is otherwise permitted to participate in compile-time type checking.

There are a number of key aspects and properties of DCOI, covered shortly, that highlight how dependency tracking interacts with terms and typing. These are not comprehensive, but are relevant to the desired applications of DCOI in the next section.

Relative relevance. The intuition of computational relevance and irrelevance is not fixed to the low and high levels, but is a relative concept between any two ordered dependency levels. Suppose there is a super-high level S such that $\mathsf{L} < \mathsf{H} < \mathsf{S}$. Then just as a low term may not meaningfully use a high term, a high term also may not meaningfully use a super-high term. The following derivable typing judgement demonstrates how these three levels can interact.

$$P :^{\mathsf{H}} \mathsf{Nat}^{\mathsf{S}} \to \mathsf{Prop}, n :^{\mathsf{S}} \mathsf{Nat} \vdash \lambda p^{\mathsf{L}}. p :^{\mathsf{L}} (P\ s^{\mathsf{S}})^{\mathsf{L}} \to P\ (s+1)^{\mathsf{S}}$$

In the context, P is a high predicate which takes as argument a super-high natural, along with a super-high natural n. Once again, the term being typed is a low function, while higher terms are involved in its type. Although the function is an identity function, its domain and codomain types are syntactically different applications of P, but this judgement still holds because the arguments of P are super-high and therefore irrelevant *with respect to P* at high.

Indistinguishability. In general, if $\ell_1 < \ell_2$, then at observer level ℓ_1, $f\ x^{\ell_2}$ must be definitionally equal to $f\ y^{\ell_2}$ regardless of what x and y are. We say that they are *indistinguishable* at level ℓ_1. This key equality is what permits the above example to type check, since $P\ s^{\mathsf{S}}$ is thus indistinguishable from $P\ (s+1)^{\mathsf{S}}$ at high. Similarly, calling the constant function above k, $k\ x^{\mathsf{L}}\ y^{\mathsf{H}}$ is indistinguishable from $k\ x^{\mathsf{L}}\ z^{\mathsf{H}}$ at low, which expresses the idea that k is truly constant in its second argument.

DCOI internalizes indistinguishability by indexing its propositional equality type with an observer level. In particular, the propositional equality $k\ x^{\mathsf{L}}\ y^{\mathsf{H}} \equiv^{\mathsf{L}} k\ x^{\mathsf{L}}\ z^{\mathsf{H}}$ is provable by reflexivity since the two sides are already indistinguishable at low, the observer level of the equality.

Elimination of higher falsehoods. The principle that lower-level terms may not meaningfully depend on higher-level terms means that, just as lower-level functions may not return higher-level arguments, destructors that return lower-level terms may not destruct higher-level terms. This holds even if the term being destructed contains no inner information (such as $\top$ or an equality proof), since reducing the destruction on a constructor requires knowing whether the term being destructed is a constructor at all.

The sole exception is the eliminator for $\bot$, since it has no constructors, so there is no information to reveal. The computational interpretation of having a proof of $\bot$ to eliminate is that we have reached an impossible dead branch, so what we do with it will never matter since it will never execute. The ability of eliminate higher-level proofs of falsehood into lower-level terms is useful when the type of a function rules out a particular branch, and the function needs to be assigned a lower level than its type.

Subsumption and downgrading. While lower-level terms cannot inspect higher-level terms, higher-level terms can inspect lower-level terms. Furthermore, a lower-level term can be raised to a higher level by *subsumption*: if a term is well typed at level ℓ_1, then it is also well typed with the same type at a higher level $\ell_2 > \ell_1$.

However, if two terms are indistinguishable by some observer level ℓ_2, then they can be indistinguishable by a *lower* observer level ℓ_1 by *downgrading*. From a security flow perspective, the higher the observer level, the more secure values may be observed, so the more things are distinguishable, since securer values will need to be compared as well instead of being ignored. Going down an observer level means more things are being hidden away, so more values will appear to be indistinguishable from one another.

4 Lattices of Type Theories

While the PTS rules and axioms of DCOI can be instantiated to produce different type systems, each instantiation, such as DCOI$^{\omega}$, is a fixed type system with a single set of terms and typing rules. Towards the goal of integrating granular feature tracking into type theory itself, this project poses the question: *What if different dependency levels corresponded to different type theories?*

More precisely, to track extensions on top of a base type theory, we would begin with a bottom dependency level corresponding to this base. Each new dependency level above bottom would contain one additional construct corresponding to a new feature or axiom. For instance, there could be an Axiom K eliminator that type checks only at a level for UIP and above, and there could be a builtin excluded middle axiom that type checks only at a level for classical reasoning and above.

Because level annotations are part of contexts and typing judgements, when a particular definition is safe to use is specified with precision, which guarantees that a particular definition never exploits an extension without permission. A definition that can be typed at the bottom level would be safe to use at all levels by subsumption, and guaranteed to never employ, say, classical reasoning. Indistinguishability reflects this guarantee, as it asserts the property that uses of values from higher forbidden theories can only trivial, such as ignoring the value or passing it around uninspected.

As dependency levels form a meet-semilattice, any two theories must have a meet, which corresponds to only the constructs that they both have in common, and which are therefore safe to use in either theory. If the join of two theories exist, then the constructs introduced in either one can be used at the joined level. Crucially, not all joins exist; a UIP level cannot be joined with a univalence level, since their coëxistence is contradictory. The shape of the lattice depends on the compatibilities between theories, as well as implication order of extensions, since one theory that encompasses the consequences of another can be placed above that other theory. The compatibility graph in Figure 1 is an example of a concrete lattice of theories, where the arrows point towards the greater theory and indicate the direction in which definitions can be raised.

One condition on the individual theories is that they must each be logically consistent. If an inconsistency exists at any theory, by the elimination of higher falsehoods, the inconsistency will infect all lower theories, including the bottom theory. Then by subsumption, the inconsistency at the bottom theory can be raised to infect all higher theories, and the entire lattice will be inconsistent. This means that any theory that features nontermination will not be permitted.

Another catch is that a theory whose extension is a new definitional equality (*i.e.* a new rule for indistinguishability) will also be ineffective. Even if that equality is defined for a given observer level, it will hold for all lower observer levels by downgrading, and the extension will be available to all lower theories. This effect cannot be mitigated using restrictive premises, as violating downgrading will violate many other desirable properties, including transitivity of definitional equality [Liu et al., 2025].

An unusual property of using DCOI for tracking theories is that the type of a term may itself be well typed within a different theory from that of the term. It's unclear what it means when, for instance, a term in the base theory can be assigned a type that uses classical principles.

4.1 Objectives

This project should answer the following questions:

1. What kinds of extensions would fit within this framework? Some broad classifications of extensions might be ones that add new type universes (*e.g.* SProp), ones that expand the rules for existing constructs (*e.g.* impredicativity, strong elimination), ones that add new computational constructs with reduction rules (*e.g.* Axiom K), and ones that add new axiomatic constructs without reduction rules (*e.g.* function and propositional extensionality, excluded middle).

2. How would a particular lattice of theories be modelled to show desirable properties such as logical consistency? Ideally, the technique used to model a particular lattice should be broadly applicable and sufficiently extensible to be applied to a different lattice without redoing all the work, so that adding more extensions remains sustainable.

3. Are there properties resulting from using DCOI that aren't expected of a feature tracking system, such as a term and its type using different sets of features? What are the consequences of these properties, and are they beneficial or detrimental?

To answer these questions, the project would be divided into two portions. The first is an implementation of a type checker for a specific lattice of type theories. The lattice should contain a sufficiently diverse set of labels and their orders to answer Question 1. Figure 1 is a good place to start, as it contains theories in different classifications with different interactions.

To evaluate the viability of such a type checker, a standard library would be implemented to exercise all levels of the lattice. The standard libraries of Rocq[2], Agda[3], and Lean[4] are good sources for inspiration, as many of their files use the features and axioms mentioned in Section 2. An implementation would also serve to verify which extensions are indeed invalid by demonstrating the inconsistencies or ineffectivities they yield.

With an implementation, useability concerns can be explored, such as level inference. Annotating definitions and arguments with every single extension it uses is an unreasonable burden on a practical proof assistant user, and it may be possible to infer the annotations either based on the syntactic constructs used or on what set of features are required for successful type checking.

The second portion is a formalized and ideally mechanized proof of consistency. Because consistency is a semantic property and depends on the strength of the metatheory used to model the type theory, the formalization should model a lattice with (at least at first) only one level above the base theory, the simplest nontrivial lattice. The focus would be on how to combine two different models of type theory, not on accommodating as many as possible from the outset.

A sensible starting point would be the mechanization of DCOI$^\omega$ Liu et al. [2025], which proves consistency and normalization of what would be the base theory in the lattice, and picking a reasonable feature to extend it with. However, this mechanization uses a syntactic logical relation indexed by well-founded universe levels as its semantic model, which may limit its extensibility; it cannot be straightforwardly extended to accommodate impredicativity, nor to accommodate typed definitional equality. A viable solution to Question 2 must overcome this limitation.

[2] https://coq.inria.fr/distrib/current/stdlib/

[3] https://agda.github.io/agda-stdlib/master/

[4] https://leanprover-community.github.io/mathlib4_docs/

A possible alternative is to use *syntactic* modelling [Boulier et al., 2017], which would involve a type-preserving translation into another type theory whose consistency is well established, guaranteeing consistency of the original system. While there exist syntactic models of other type theories [Gilbert et al., 2019; Winterhalter, 2024] with notions of irrelevance, which is one application of indistinguishability, a syntactic model of dependency tracking with dependent types is unexplored.

The process of accomplishing these two portions of the project should answer Question 3, either by the implementation revealing unexpected examples that can or cannot be type checked, or by metatheoretical properties that hold based on the modelling technique chosen. Only once these properties are revealed will we know what further work can be done, from augmenting the implementation closer to a practical proof assistant, to proving more complex theorems like normalization and decidability of type checking, or proving consistency for a larger lattice.

5 Prior work

This project builds on prior work on DCOI [Liu et al., 2024] and DCOI$^\omega$ [Liu et al., 2025], on both of which I am second author. For the former paper, I implemented a prototype type checker for DCOI augmented with inductive types by extending the minimal dependent type checker `pi-forall` [Weirich, 2022], and wrote examples using the type checker and motivating examples for DCOI. I also proved a few of the lemmas in the mechanization. For the latter paper, I wrote about half of the prose, mostly for the earlier sections, and proved a few of the lemmas as well. As part of an investigation toward incorporating a relational model for DCOI, I mechanized a PER model for MLTT based on the logical relation used to prove consistency of DCOI$^\omega$, but ultimately the gap between MLTT and DCOI could not be bridged, so this work does not appear in the final paper.

Outside of DCOI, I have worked on Stratified Type Theory (StraTT) [Chan and Weirich, 2025], which annotates typing judgements similarly to dependency tracking, but the annotations are universe levels, and restrictions on what levels may be used where enforces consistency where traditionally it is enforced by disallowing type-in-type. In other words, instead of stratifying universes into a hierarchy, typing judgements themselves are stratified. Although StraTT is not a dependency tracking system in the same way DCOI is, it demonstrates that there may be multiple ways to retain usage information that enforces desired properties such as consistency or irrelevance. Even if the particular setup for DCOI turns out not to be suitable for this project, it may be reasonable to instead explore a more StraTT-like structure.

6 Related work

6.1 Two-level type theory

6.2 Modal type theory

6.3 Proof assistants

6.4 Applications of extensions

7 Conclusion

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

Thorsten Altenkirch and Conor McBride. 2006. Towards Observational Type Theory. http://strictlypositive.org/ott.pdf

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational equality, now!. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification* (Freiburg, Germany) *(PLPV '07)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/1292597.1292608

Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. 2021. Syntax and models of Cartesian cubical type theory. *Mathematical Structures in Computer Science* 31, 4 (2021), 424–468. https://doi.org/10.1017/S0960129521000347

Carlo Angiuli, Kuen-Bang (Favonia) Hou, and Robert Harper. 2017. Computational Higher Type Theory III: Univalent Universes and Exact Equality. https://doi.org/10.48550/arXiv.1712.01800

Franco Barbanera and Stefano Berardi. 1996. Proof-irrelevance out of excluded-middle and choice in the calculus of constructions. *Journal of Functional Programming* 6, 3 (1996), 519–526. https://doi.org/10.1017/S0956796800001829

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

Marc Bezem, Thierry Coquand, and Simon Huber. 2019. The Univalence Axiom in Cubical Sets. *Journal of Automated Reasoning* 63 (2019), 159–171.

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 182–194. https://doi.org/10.1145/3018610.3018620

Jonathan Chan and Stephanie Weirich. 2025. Stratified Type Theory. In *Programming Languages and Systems*, Viktor Vafeiadis (Ed.). Springer Nature Switzerland, Cham, 0–0.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:34. https://doi.org/10.4230/LIPIcs.TYPES.2015.5

Youyou Cong and Kenichi Asai. 2018. Handling delimited continuations with dependent types. *Proc. ACM Program. Lang.* 2, ICFP, Article 69 (July 2018), 31 pages. https://doi.org/10.1145/3236764

The Coq Development Team. 2022. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.5846982

Thierry Coquand. 1992. The paradox of trees. *BIT Numerical Mathematics* 32 (March 1992), 10–14. https://doi.org/10.1007/BF01995104

Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 233–243. https://doi.org/10.1145/351240.351262

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, Cham, Cham, Switzerland, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL, Article 3 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290316

Michael Hedberg. 1998. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming* 8, 4 (1998), 413–436. https://doi.org/10.1017/S0956796898003153

James G. Hook and Douglas J. Howe. 1986. *Impredicative Strong Existential Equivalent to Type:Type*. Technical Report TR86-760. Cornell University. https://hdl.handle.net/1813/6600

Yiyun Liu, Jonathan Chan, Jessica Shi, and Stephanie Weirich. 2024. 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. 2025. Consistency of a Dependent Calculus of Indistinguishability. *Proc. ACM Program. Lang.* 9, POPL (Jan. 2025), 27 pages. https://doi.org/10.1145

Per Martin-Löf. 1972. An intuitionistic theory of types.

The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. https://doi.org/10.1145/3372885.3373824

Étienne Miquey. 2017. A Classical Sequent Calculus with Dependent Types. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 777–803. https://doi.org/10.1007/978-3-662-54434-1_29

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Chalmers University of Technology and Göteborg University, Göteborg, Sweden. https://research.chalmers.se/en/publication/46311

Michel Parigot. 1992. $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning*, Andrei Voronkov (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–201. https://doi.org/10.1007/BFb0013061

Frank Pfenning and Christine Paulin-Mohring. 1990. Inductively defined types in the Calculus of Constructions. In *Mathematical Foundations of Programming Semantics*, M. Main, A. Melton, M. Mislove, and D. Schmidt (Eds.). Vol. 442. Springer-Verlag, Berlin/Heidelberg, Germany, 209–228. https://doi.org/10.1007/BFb0040259

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

Loïc Pujet and Nicolas Tabareau. 2023. Impredicative Observational Equality. *Proc. ACM Program. Lang.* 7, POPL, Article 74 (Jan. 2023), 26 pages. https://doi.org/10.1145/3571739

Loïc Pujet and Nicolas Tabareau. 2024. Observational Equality Meets CIC. In *Programming Languages and Systems*, Stephanie Weirich (Ed.), Vol. 14576. Springer Nature Switzerland, Cham, 275–301. https://doi.org/10.1007/978-3-031-57262-3_12

Thomas Streicher. 1993. *Investigations into intensional type theory*. Ph. D. Dissertation. Ludwig Maximilian Universität, Munich, Germany. https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf

William Walker Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.* 3, ICFP, Article 87 (July 2019), 29 pages. https://doi.org/10.1145/3341691

Stephanie Weirich. 2022. Implementing Dependent Types in pi-forall. https://doi.org/10.48550/arxiv.2207.02129 Lecture notes for the Oregon Programming Languages Summer School.

Théo Winterhalter. 2024. Dependent Ghosts Have a Reflection for Free. *Proc. ACM Program. Lang.* 8, ICFP, Article 258 (Aug. 2024), 29 pages. https://doi.org/10.1145/3674647