Internalizing Extensions in 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. [Needs to be rewritten to reflect new introduction.]

A first step towards a framework of extensions could use the Dependent Calculus of Indistinguishability (DCOI) [Liu et al., 2024] to formalize 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. This report explores how extensions could correspond to dependency levels, and how the lattice would describe how extensions are permitted to interact.

1 Introduction

At the core of a proof assistant founded on the Curry–Howard correspondence is a type checker that validates a proof of a proposition—a term inhabiting a corresponding type in a dependent type theory. These type theories are typically based on some flavour of Martin-Löf Type Theory (MLTT) [Martin-Löf, 1972] or the Calculus of Inductive Constructions (CIC) [Pfenning and Paulin-Mohring, 1990]. In practice, a proof assistant doesn't implement merely one type theory, but a whole host of them, as they will include language extensions that augment or modify its reasoning power.

These type theoretic extensions can consist of additional typing rules, constructs, and/or definitional equalities. Because each extension embodies semantically distinct reasoning principles, enabling an extension results in a separate theory altogether. Furthermore, there exist extensions that are mutually incompatible because together they violate logical consistency. For instance, *uniqueness of identity proofs*, which propositionally equates all proofs of the same equality, is incompatible with *univalence*, which adds additional and provably distinct proofs of equality. In Section 2, I describe these extensions and more found in select proof assistants, and the ways some combinations are incompatible.

Thus proof assistants are careful to track the usage of language extensions to rule out inconsistencies. However, the tracking done by their type checkers 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 prohibited from using an extension. Furthermore, given two incompatible extensions, a definition in one extension's theory may not be used at all in the another extension's theory, not even just in a type. This means one theory cannot be used as a metatheory to prove properties about definitions in the other theory if those theories are incompatible. For instance, considering classical axioms as a language extension, one would not be able to explore what is constructively proveable about classical principles.

Ultimately, what is missing is a framework for describing fine-grained control of proofs and programs across multiple type theories, where even incompatible theories can interact in interesting ways. These properties are reminiscent of type systems with *dependency analysis*. Here, terms are stratified by *dependency levels*, which can be thought of intuitively as permission levels tracking permitted usages. Even if we do not have access to a particular level, terms at that level can still be manipulated and reasoned about as long as they are not inspected or evaluated. [Is this too vague?]

I have worked on a type system with dependency tracking, the Dependent Calculus of Indistinguishability (DCOI) [Liu et al., 2024], along with its variant DCOI$^\omega$ [Liu et al., 2025], which is a logically consistent type theory. DCOI could potentially 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. I describe the key properties of DCOI relevant for this application in Section 3. I then speculate on the details of this mapping, lay out the objectives for such a framework, and list possible first steps towards accomplishing them in Section 4. 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.

2.1 Built-in features

Each of these proof assistants include features that extend the power of their foundations; below are a few notable extensions, some of which are hidden behind option flags.

Impredicativity. Rocq and Lean, being based on CIC, feature a universe Prop of propositions. This universe is *impredicative*, meaning 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*.

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 by self-application.

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

In contrast, 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. A universe of propositions is said to be *strict* when the inhabitants of its propositions are definitionally equal (*i.e.* proof irrelevant) and thus treated as interchangeable during type checking. Lean's Prop is always strict, while 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]. To use strict Prop, Rocq requires the flag `Allow StrictProp`, while Agda requires the option `{-# OPTIONS --prop #-}`.

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.

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

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

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

If we try to prove this by destructing b_1 and b_2 as $(bnat\ n_1\ m\ p_1)$ and $(bnat\ n_2\ m\ p_2)$, $getNat\ m\ b_1$ and $getNat\ m\ b_2$ reduce to n_1 and n_2.

$$eqBNat\ m\ (bnat\ n_1\ m\ p_1)\ (bnat\ n_2\ m\ p_2) : n_1 \equiv n_2 \to bnat\ n_1\ m\ p_1 \equiv bnat\ n_2\ m\ p_2$$

While we have an equality $n_1 \equiv n_2$, we do *not* have a proof of $p_1 \equiv p_2$[1]. Depending on how $\cdot \le \cdot$ 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 be ignored. Then rewriting the goal by the given equality is sufficient for it to be proven by reflexivity.

$$\begin{aligned}
eqBNat\ m\ (bnat\ n_1\ m\ p_1)\ (bnat\ n_2\ m\ p_2)\ e\ &: bnat\ n_1\ m\ p_1 \equiv bnat\ n_2\ m\ p_2 \\
&:= \mathsf{rewrite}\ e\ \mathsf{in}\ \mathsf{refl}
\end{aligned}$$

[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.

Uniqueness of identity proofs (UIP). The *uniqueness of identity proofs* (UIP) asserts that inhabitants of the same propositional equality are themselves propositionally equal. It can be proven using *Axiom K* [Streicher, 1993], a computational eliminator for propositional equalities of type $a \equiv a$.

$$K : \forall(A : \mathsf{Type})\ (a : A)\ (P : a \equiv a \to \mathsf{Prop})\ (p : a \equiv a).\ P\ \mathsf{refl} \to P\ p$$
$$K\ A\ a\ P\ \mathsf{refl} \rightsquigarrow \mathsf{refl}$$

Agda's default pattern matching behaviour, which permits matching on an equality of $a \equiv a$ as reflexivity, admits a proof of UIP as well as defining Axiom K. While not inherently part of Rocq's type theory, Axiom K is axiomatized in the standard library as `Logic.Eqdep.eq_rect_eq`.

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 augments the reasoning power of the type theory, there are inductive 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].

Strong elimination. Destructing or eliminating an element of an inductive datatype into a type at a larger universe is known as *strong* or *large* elimination. That is, a term whose type is in Type_i is eliminated to return a term whose type is in Type_j for some $j > i$. For proofs of propositions in Prop, this includes eliminating into any non-proposition type.

Strong elimination is a necessary ingredient in discriminating constructors of proof-relevant datatypes, such as the booleans. While true and false are syntactically distinct, proving their propositional inequivalence requires lifting the booleans to propositions truthhood $\top$ and falsehood $\bot$.

$$\mathit{lift} : \mathsf{Bool} \to \mathsf{Prop}$$
$$\mathit{lift} := \lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ \top\ \mathsf{else}\ \bot$$

The branching expression is a strong elimination because it returns a type, or equivalently because its return type is Prop, a universe. Letting $\mathsf{cong}\ f$ be a proof of congruence of f over an equality, to complete the proof of $\mathsf{true} \equiv \mathsf{false} \to \bot$, the trivial proof of truthhood tt is rewritten by the lifted equality $\top \equiv \bot$.

$$\mathit{trueNotFalse} : \mathsf{true} \equiv \mathsf{false} \to \bot$$
$$\mathit{trueNotFalse} := \lambda e.\ \mathsf{rewrite}\ (\mathsf{cong}\ \mathit{lift}\ e)\ \mathsf{in}\ \mathsf{tt}$$

2.2 Axioms

Rocq, Lean, and Agda all have mechanisms for defining axioms or postulates, which are declarations of constants without definitions. Although not all axioms are consistent, there are many well-studied additions commonly used in practice that are worth considering as extensions in their own right.

Extensional principles. Some models of type theory semantically equate things that are not syntactically (either definitionally or propositionally) equal; extensional principles adds semantic equalities as propositional equalities. Examples include *function extensionality*, which equates two functions if they are pointwise equal, and *propositional*

extensionality, which equates two propositions if they are biimplicated. These axioms are found in the standard libraries of Lean as `funext` and `propext`, and of Rocq as `Logic.FunctionalExtensionality.functional_extensionality` and `Logic.PropExtensionality.propositional_extensionality`. A notable consequence of propositional extensionality is propositional proof irrelevance.

Another example is *univalence*, which 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. Univalence together with proof irrelevance implies propositional extensionality, since biimplicated 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 function and propositional extensionality is encountered when encoding a function as a relation whose functionality is proven *a posteriori*. This is a frequent pattern in proof assistants, as inductive relations often have better ergonomic support than dependently-typed reasoning over functions. For example, consider a two-place predicate over a representation of types Ty and terms Tm, which has the type $Ty \to Tm \to \mathsf{Prop}$. If we have trouble defining this predicate recursively due to termination issues or inductively due to strict positivity issues, we can instead view it as a function from Ty to a predicate $Tm \to \mathsf{Prop}$ and try encoding it as a relation:

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

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 Tms. Functionality of R demonstrates that Tys have unique interpretations. To show that R is functional, *i.e.*

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

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

The disadvantage of axiomatic equalities is that rewriting by them does not reduce, which can make reasoning about terms rewritten by such equalities 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.

Classical principles. There are a number of classical axioms that do not hold intuitionistically. The most common is the principle of *excluded middle* (EM), which asserts that all propositions are either true (inhabited) or false (uninhabited). EM is equivalent to several other principles, including *double negation elimination* (DNE), $\forall (A : \mathsf{Prop}).\ \neg\neg A \to A$, and *Peirce's law*, $\forall (A\ B : \mathsf{Prop}).\ ((A \to B) \to A) \to A$. More powerful axioms which imply EM include the axiom of choice and the (in)definite description operators, which deal with extracting a concrete piece of data out of merely knowing that such a piece of data exists without constructing it.

Because a large majority of mathematics is done classically, many communities mechanizing mathematics freely use classical principles. The axiom of excluded middle, for instance,

is declared in Rocq as `Logic.Classical_Prop.classic`, and in Lean as `em`. The `Logic` subdirectory of Rocq's standard library contains the classical axioms along with proofs about their properties. Similarly, Lean's mathematical library mathlib [mathlib Community, 2020] contains proofs that rely on classical axioms, and tactics such as `tauto` automatically apply classical reasoning. [Not sure what the point I was trying to make about constructively implementing classical principles so I've commented that out for now...]

2.3 Extensions and inconsistencies

One has to be careful 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.

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 not well supported.[2] In

[2] https://github.com/coq/coq/issues/9458

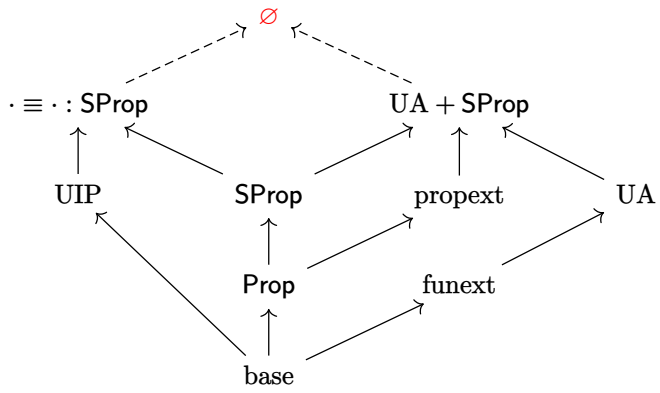

Figure 1: A compatibility graph of theories with impredicative Prop, proof irrelevance (SProp), UIP, univalence (UA), function extensionality (funext), propositional extensionality (propext), and (in)compatible combinations.

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 asserting against an extension; that is, there is no general mechanism to tell the type checker to fail if a particular definition uses some feature or axiom. Such an assertion would guarantee it safe to be used by other definitions that use an incompatible extension. For instance, ensuring that a proof does not use Axiom K means that it may be used by another proof that does use univalence. In this specific case, Agda does have the `--without-K` flag for this purpose, but few options have a corresponding anti-option. Meanwhile, Rocq and Lean's axiom printing mechanism does not modify type checking behaviour.

- 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 appropriate 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. While a description exists for a set of compatible theories by taking their greatest encompassing theory, this is not the case when there are two incompatible theories that exist side by side. Such theories may be described independently, but would be incomplete descriptions of the entire core of a proof assistant.

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, 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 information 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.

Rather than irrelevance or information flow, the target application is extension tracking, where each dependency level corresponds to a set of added extensions. Relevant to this application are a number of key aspects and properties of DCOI that highlight how dependency tracking interacts with terms and typing.

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, the dependency levels of DCOI are instead mapped 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 built-in 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.

Following the rules of DCOI, each individual theory must each be logically consistent. If an inconsistency exists at any theory, by the elimination of higher falsehoods, the inconsistency will propagate to all lower theories, including the bottom theory. Then by subsumption, the inconsistency at the bottom theory can be raised to propagate to all higher theories, and the entire lattice will be inconsistent. This means that if eliminating falsehoods works exactly as in DCOI, any theory that features nontermination will not be permitted.

As the goal is to exclude incompatible extensions from a proof assistant, disallowing logically inconsistent theories is a desirable trait. Nevertheless, there may be a few ways to modify falsehood elimination to permit them. One way is to instead disallow eliminating falsehoods to lower levels, only to the same level. Another is to take ideas from works from the Trellys project, such as λ^θ [Casinghino et al., 2014] and Sep^3 [Kimmell et al., 2012], and impose a value or termination restriction on falsehoods being eliminated. If the falsehood in an inconsistent theory is nonterminating or not a value, then it cannot be eliminated at all, preventing its propagating to lower theories.

One catch is that a theory whose extension is a new definitional equality (*i.e.* a new rule for indistinguishability) cannot be contained within its level. 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[3], Agda[4], and Lean[5] 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

[3] https://coq.inria.fr/distrib/current/stdlib/
[4] https://agda.github.io/agda-stdlib/master/
[5] https://leanprover-community.github.io/mathlib4_docs/

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.

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 Multi-system frameworks

Two-level type theory. The most similar work to extension tracking is two-level type theory (2LTT) [Annenkov et al., 2023]. It consists of an inner homotopical type theory with univalence and an outer intensional type theory with UIP, along with a conversion operation $\uparrow \cdot$ from the inner theory to the outer. The inner and outer type theories have independent type formers, including separate inner (path) equality types $\cdot \equiv^{\mathsf{i}} \cdot$ and outer (strict) equality types $\cdot \equiv^{\circ} \cdot$. Importantly, converting an inner equality does *not* yield the outer equality type; otherwise, univalence on inner equalities could be converted to univalence on outer equalities, which would contradict UIP of the outer equality.

These inner and outer levels are different from dependency levels in DCOI, where all levels share the same type formers, and a lower equality can be raised to a higher equality by subsumption. In particular, if a lattice of type theories includes one that supports UIP, then that level will prove that proofs of the same equality at *all* levels are themselves equal. Meanwhile, in 2LTT, UIP only holds for proofs the outer equality and not for converted proofs of the inner equality.

The conversion operator can be thought of as an explicit subsumption, and how it interacts with the inner and outer equalities is similar to how DCOI's propositional equality interacts with indistinguishability at lower and higher levels. To demonstrate, given two inner terms x, y, the implication $\uparrow x \equiv^{\circ} \uparrow y \to x \equiv^{\mathsf{i}} y$ holds in 2LTT while the converse generally does not. Similarly, in DCOI, $x \equiv^{\mathsf{H}} y \to x \equiv^{\mathsf{L}} y$ holds by downgrading while the converse also generally does not.

The Trellys project. Instead of combining multiple type theories, the Trellys project focussed on combining dependently-typed logical reasoning with (potentially nonterminating) functional programming. The main works within the project are λ^{θ} [Casinghino et al., 2014], which classifies typing judgements of a single language into logical and programmatic fragments; Sep^3 [Kimmell et al., 2012], which syntactically separate proofs from programs; and Nax [Ahn, 2014], which augments dependent types with Mendler-style recursion schemes.

Of these three, DCOI is closest to λ^{θ}, whose logical and programmatic classifications are similar to DCOI's dependency levels. Because the logical fragment is subsumed within the programmatic fragment, the additional features found in the programmatic fragment can be thought of as an extension of the logical one. Notable features of the extension include isorecursive types and unrestricted recursion, allowing for nonterminating programs. Normalization of the logical fragment is ensured by only allowing boxed programs to be applied to its functions, only allowing unboxing of values, and restricting reduction to call by value. While the boxing mechanism is similar to domain level annotations in DCOI, the value restrictions are specific to handling presence of potential divergence.

The proof of normalization uses a step-indexed logical relation, where stepping only occurs in the programmatic fragment. However, λ^{θ} only has a single type universe with no type polymorphism, which the logical relation takes advantage of, so this proof technique will not extend to DCOI or any variant with a universe hierarchy.

6.2 Other proof assistants

Section 2 broadly covers a number of optional features and common axioms in Rocq, Lean, and Agda. There are many other proof assistants of varying relevance not discussed above.

Idris 2 [Brady, 2021] is a dependently typed programming language with partiality. Definitions can be marked as `total`, `covering`, or `partial`; in principle, totality ensures consistency, covering ensures type safety while allowing divergence, and partiality does not ensure either. Because partiality subsumes covering subsumes totality, these modifiers can also be thought of as members of a lattice. Idris 2 is closer to λ^θ than to DCOI, especially because it is also call by value, and the partiality modifier is designed so that diverging terms can appear in types and be reasoned about while not being reduced during type checking. Although Idris 2 is based on Quantitative Type Theory [Atkey, 2018], there is no formal description of its core type system that describes all of its features, especially as it is a rapidly evolving language.

F* [Swamy et al., 2016] is a proof assistant with dependency tracking for different effects. Its dependency levels include a `Tot` level for total programs, and a `Dv` level above it for potentially diverging programs. In contrast to λ^θ, the total fragment of F* may not refer to the diverging fragment, so the proof of weak normalization of the total fragment involves a logical relation that does not consider levels above `Tot`. All the effects in F* are implemented as indexed monads, which get compiled away to the new core calculus TotalF* [Rastogi et al., 2021]; divergence aside, effects do not extend the internal type system.

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

Ki Yung Ahn. 2014. *The Nax Language: Unifying Functional Programming and Logical Reasoning in a Language based on Mendler-style Recursion Schemes and Term-indexed TypesTerm-indexed Types*. Ph. D. Dissertation. Portland State University. https://doi.org/10.15760/etd.2086

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

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

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

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

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

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 33–45. https://doi.org/10.1145/2535838.2535883

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

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

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

Garrin Kimmell, Aaron Stump, Harley D. Eades, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. 2012. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification* (Philadelphia, Pennsylvania, USA) *(PLPV '12)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2103776.2103780

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

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

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

Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. 2021. Programming and Proving with Indexed Effects. https://fstar-lang.org/papers/indexedeffects/indexedeffects.pdf

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

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. *ACM SIGPLAN Notices* 51, 1 (Jan. 2016), 256–270. https://doi.org/10.1145/2914770.2837655

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