Internalizing Extensions in Lattices of Type Theories

Jonathan Chan

Proof assistants such as Rocq [Coq Development Team, 2022], Agda [Norell, 2007], and Lean [de Moura et al., 2015] are tools that automatically check the correctness of mechanized proofs. They are founded on the Curry–Howard correspondence, which associates logical propositions to types, proofs of propositions to terms of the corresponding types, and proof checking to type checking. Thus, at the core of a proof assistant is a type checker for a type theory, 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, proof assistants do not implement merely one type theory, but a whole host of them, as they include language extensions that augment or modify their reasoning power.

These type theoretic extensions consist of new typing rules, axioms, constructs, and/or definitional equalities. Because each extension embodies semantically distinct reasoning principles, enabling an extension results in a separate theory altogether. However, some extensions are mutually incompatible and cannot be enabled simultaneously because together they violate logical consistency, which says that falsehood cannot be proven.

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. Being able to specify exactly where the need for various extensions comes from helps us be more mindful about enabling them, especially if they prevent future reuse of definitions due to incompatible extensions. 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 theories that use 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. This fine-grained control of terms is reminiscent of type systems with *dependency analysis*. In such systems, terms are stratified by *dependency levels*, which can be thought of intuitively as permission levels tracking where terms may be used. These levels are structured as a lattice, whose order is a subsumption relation: a larger (higher) level has more permissions than a smaller (lower) level. However, 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.

I have worked on a dependent type system with dependency tracking, the Dependent Calculus of Indistinguishability (DCOI) [Liu et al., 2024b], along with its variant DCOI$^\omega$ [Liu et al., 2025], which is a logically consistent type theory. DCOI could be used as a basis for a framework that internalizes extension tracking by stratifying their corresponding type theories into hierarchies of dependency levels. This creates a lattice whose points are the sets of extensions enabled in each theory, ordered by subset. We begin with a bottom dependency level for the base type theory corresponding to the empty set of extensions. For each additional construct corresponding to an extension, we add a new dependency level above the theory it extends.

This project aims to 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, ones that expand the rules for existing constructs, ones that add new computational constructs with reduction rules, and ones that add new axiomatic constructs without reduction rules.

2. What are useful applications of being able to freely refer to other theories? Are there meaningful theorems about one theory that cannot be proven in that theory, but can be proven in a different yet potentially incompatible theory?

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

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 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[1], Agda[2], and Lean[3] are good sources for inspiration, as many of their files use various common features and axioms. An implementation would also serve to verify which extensions are indeed invalid by demonstrating the inconsistencies they yield.

Additionally, writing and checking large proofs would be feasible, which helps with exploring more complex applications. For example, two-level type theory (2LTT) [Altenkirch et al., 2016; Annenkov et al., 2023] is a type theory which contains an inner Homotopy Type Theory [Univalent Foundations Program, 2013] and an outer MLTT, designed so that the outer theory acts as a metatheory for proving things about the inner theory that cannot be proven within the inner theory alone. We can use libraries[4] for 2LTT in Agda as inspiration for testing whether this project's implementation of the same extensions can handle the same proof load.

The useability of the implementation would inform the design of the system, such as level inference and level polymorphism. Annotating definitions and arguments with every single extension it uses is an unreasonable burden on a proof assistant user. The prototype implementation accompanying DCOI [Liu et al., 2024a] therefore has rudimentary level inference, defaulting to a minimum level. However, the prototype's lattice is a total order, and its examples typically use no more than two levels. An implementation with a lattice containing incompatible extensions and applications that involve more than two theories would pinpoint what is required from level inference in practice for a more sophisticated inference algorithm.

Rewriting libraries in this implementation is also an exercise in determining where code duplication occurs and whether level polymorphism would help eliminate it. Although subsumption allows lifting definitions vertically, so to speak, from lower theories to higher ones, it does not allow transporting definitions horizontally from one theory to a different, incompatible theory. In addition, while a function can quantify over terms at specific levels, it cannot quantify over *all* levels, nor over levels that satisfy some ordering constraint. Having a library of examples in the implementation would reveal whether these are real concerns to be addressed and what kind of level polymorphism could address them.

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 taking 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.

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 DCOI, a syntactic model of dependency tracking with dependent types is unexplored.

[1] https://coq.inria.fr/distrib/current/stdlib/
[2] https://agda.github.io/agda-stdlib/master/
[3] https://leanprover-community.github.io/mathlib4_docs/
[4] https://github.com/UnivalencePrinciple/2LTT-Agda

References

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending Homotopy Type Theory with Strict Equality. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 62)*, Jean-Marc Talbot and Laurent Regnier (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:17. https://doi.org/10.4230/LIPIcs.CSL.2016.21

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

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

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

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

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

Yiyun Liu, Jonathan Chan, and Stephanie Weirich. 2024a. *Artifact associated with Consistency of a Dependent Calculus of Indistinguishability.* University of Pennsylvania. https://doi.org/10.5281/zenodo.13930551

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.

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

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

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