McTT: A Verified Kernel for a Proof Assistant

JUNYOUNG JANG, McGill University, Canada
ANTOINE GAULIN, McGill University, Canada
JASON Z. S. HU*, Amazon, USA
BRIGITTE PIENTKA, McGill University, Canada

Proof assistants based on type theories have been widely successful from verifying safety-critical software to establishing a new standard of rigour by formalizing mathematics. But these proof assistants and even their type-checking kernels are also complex pieces of software, and software invariably has bugs, so why should we trust such proof assistants?

In this paper, we describe the McTT (Mechanized Type Theory) infrastructure to build a verified implementation of a kernel for a core Martin-Löf type theory (MLTT). McTT is implemented in Rocq and consists of two main components: In the theoretical component, we specify the type theory and prove theorems such as normalization, consistency and injectivity of type constructors of MLTT using an untyped domain model. In the algorithmic component, we relate the declarative specification of typing and the model of normalization in the theoretical component with a functional implementation within Rocq. From this algorithmic component, we extract an OCaml implementation and couple it with a front-end parser for execution. This extracted OCaml code is comparable to what a skilled human programmer would have written and we have successfully used it to type-check a series of small-scale examples.

McTT provides a fully verified kernel for a core MLTT with a full cumulative universe hierarchy. Every step in the compilation pipeline is verified except for the lexer and pretty-printer. As a result, McTT serves both as a framework to explore the meta-theory of advanced type theories and to investigate optimizations of and extensions to the type-checking kernel.

CCS Concepts: • **Theory of computation** → **Type theory**; *Logic and verification*; **Program verification**; *Denotational semantics*; **Constructive mathematics**.

Additional Key Words and Phrases: Type Theory, Verified Implementation, Martin-Löf Type Theory, Rocq, Normalization-by-Evaluation

ACM Reference Format:
Junyoung Jang, Antoine Gaulin, Jason Z. S. Hu, and Brigitte Pientka. 2025. McTT: A Verified Kernel for a Proof Assistant. *Proc. ACM Program. Lang.* 9, ICFP, Article 242 (August 2025), 32 pages. https://doi.org/10.1145/3747511

1 Introduction

Over the past decades, proof assistants based on type theories have been widely used to develop safety-critical software (e.g. CompCert [Leroy 2009a,b]). They have also been used to formalize a significant portion of mathematics with the goal of establishing a new standard of rigour [Avigad and Harrison 2014] in this field. For example, The Mathlib Community [2020] in the Lean proof assistant has roughly half a million lines of code and contains formalizations of many non-trivial mathematics, from number theory to perfectoid spaces [Buzzard et al. 2020]. These endeavors are

*Jason Z. S. Hu made his contribution during his Ph.D. study at McGill University, Canada.

Authors' Contact Information: Junyoung Jang, junyoung.jang@mail.mcgill.ca, School of Computer Science, McGill University, Montréal, Québec, Canada; Antoine Gaulin, antoine.gaulin@mail.mcgill.ca, School of Computer Science, McGill University, Montréal, Québec, Canada; Jason Z. S. Hu, zhonhu@amazon.com, Amazon, Seattle, Washington, USA; Brigitte Pientka, bpientka@cs.mcgill.ca, School of Computer Science, McGill University, Montréal, Québec, Canada.

ACM 2475-1421/2025/8-ART242
https://doi.org/10.1145/3747511

testimony to the success of proof assistants in practice. Nevertheless, proof assistants are complex and unavoidably have bugs, so why should we trust a proof assistant and its certification of a proof?

To minimize such concerns, proof assistants are typically designed to rely on a small, trusted kernel to verify a proof. This design approach narrows our focus to this trusted core, which is supposed to be close to its well-understood, underlying theory, e.g., Calculus of Inductive Construction (CIC) [Coquand and Paulin 1988; Paulin-Mohring 1993] or Martin-Löf type theory (MLTT) [Martin-Löf 1975]. However, in practice, kernels of modern proof assistants are no longer small, spanning over tens of thousands of lines of code. Rocq's kernel spans approximately 41K lines of OCaml code, while Lean's kernel consists of approximately 8K lines of C++ code. Furthermore, the kernel might significantly deviate from and extend the core type theories in non-trivial ways, both to experiment with new features and to support more practical features such as meta-variables for partial proofs or asynchronous type-checking. As a result, all type-theoretic proof assistants have encountered major bugs: In Rocq, on average, one critical bug has been found every year. Lean has experienced both soundness bugs[1] and segmentation fault.[2]

In this work, we investigate a long-standing question: *How can we build a verified type-checker for type-theoretic proof assistants?* – Two fundamental aspects need to be answered:

1) Check the Theory. The type theory underlying a proof assistant is itself a full-scale, complex system. Hence, verifying the type-checking kernel should start with formalizing the type theory itself, together with its meta-theory. Such an endeavour not only increases the trust in the type-theoretic foundation itself, but also such puts great pressure on many aspects of a proof assistant – from stretching the boundaries of the theoretical foundations, to understanding the good practices of developing such meta-theoretic results, and testing its implementation [Chapman 2008].

The most frequently adapted proof technique to prove normalization of MLTT is Tait's reducibility candidates [Tait 1967] (see, for example, [Abel et al. 2023, 2018; Adjedj et al. 2024; Pujet and Tabareau 2022, 2023]). These mechanizations often require significant technical set-up and time investment. Moreover, this style of normalization only gives us β normal forms.

An alternative method is normalization-by-evaluation (NbE), where we use a mathematical model that includes a notion of computation. Such NbE proofs have the advantage of directly giving a normalization algorithm [Berger and Schwichtenberg 1991] and are often much more compact. For example, Hu et al. [2023] mechanize an NbE algorithm for MLTT extended with a necessity modality based on Abel [2013]'s untyped domain model in just 11K lines of Agda code. In Rocq, Wieczorek and Biernacki [2018] lead the effort of mechanizing a variant of MLTT using NbE.

However, there is a subtle issue that arises in these mechanizations of type theories: existing proofs in Agda [Abel et al. 2018; Hu et al. 2023; Pujet and Tabareau 2023] critically rely on induction-recursion [Dybjer and Setzer 2003] to formalize the semantics. However, Rocq does not support induction-recursion. For this reason, prior Rocq mechanizations of MLTT (e.g. [Adjedj et al. 2024; Wieczorek and Biernacki 2018]) restrict MLTT to a finite number of universes. This has been a stumbling block in scaling mechanizations of MLTT in Rocq. This is unfortunate, since Rocq has more support for automation and better code extraction than Agda.

In addition, none of the previous mechanizations of normalization proofs for MLTT were developed with an eye towards extracting a practical type-checker and normalizer from its development. However, to establish a correspondence between a type-checker and its underlying theory, the declarative specification of the type theory must be developed hand in hand with an algorithmic one that forms the core of a usable type-checker.

[1] https://github.com/leanprover/lean4/issues/1433 and https://github.com/leanprover/lean4/issues/2125
[2] https://github.com/leanprover/lean4/issues/5188

2) Check the Checker. A verified type-checker for a (core) type theory provides the highest level of assurance and eliminates errors. The first notable verified kernel of a significant fragment of CIC goes back to Barras and Werner [1997]. Boespflug and Burel [2012] and Assaf et al. [2016] propose to translate Rocq proofs into the logical framework $\lambda\Pi$-Modulo, and then use that framework to certify the Rocq proofs. However, since the type-checker for $\lambda\Pi$-Modulo is smaller than that for Rocq, we cannot establish meta-theoretic results about encodings within the framework.

An alternative approach is pursued in the MetaRocq project, which aims to formalize Rocq in Rocq and provide tools for manipulating Rocq terms and developing certified plugins [Sozeau et al. 2020a,b, 2019]. The main idea is to reflect the representation of Rocq objects in the implementation of Rocq into Rocq itself and to prove meta-theoretic properties such as subject reduction, canonicity, or weak call-by-value standardization. However, while there is a normalization proof for the CIC, there is no normalization proof for the system that is actually modelled in Rocq.

While one might argue that we cannot prove and hence mechanize the consistency of the entire Rocq's type theory in Rocq, one could still hope to mechanize a significant fragment of it.

Contribution. In this paper, we develop McTT (Mechanized Type Theory), a verified type-checker of MLTT in Rocq. Channeling the spirit of the CompCert project [Leroy 2009a,b], we start by formalizing the type theory together with its meta-theory (*theoretical component*). To extract a usable type-checker, we also implement a type-checker for our type theory in Rocq (*algorithmic component*) and prove it correct with respect to the specification in the theoretical component.

In *the theoretical component*, we *check the theory*. In particular, we formalize a version of MLTT that has natural numbers, Π types, a full cumulative universe hierarchy, and universe subtyping. This formalization includes the mechanized proof of logical consistency using normalization-by-evaluation (NbE), closely following the proof of Abel [2013]. Compared to a similar previous mechanization effort [Wieczorek and Biernacki 2018] in Rocq, McTT formalizes a richer type theory. A key obstacle when mechanizing the meta-theory of MLTT with a full universe hierarchy is that its semantics is prevalently defined by using induction-recursion. We provide a principled way to formalize such definitions in Rocq by exploiting impredicativity. As such, the theoretical component of McTT not only increases our confidence in the core MLTT, but provides insights and derives good practices on how to formalize the meta-theory of type theories in proof assistants such as Rocq. Moreover, we develop the meta-theory of MLTT hand-in-hand with its algorithmic, functional implementation. This leads us to incorporate universe subtyping into MLTT. This is an extension which is usually supported in CIC, but not traditionally in MLTT. Universe subtyping is necessary to establish soundness and completeness of the algorithmic type checker.

In *the algorithmic component* of McTT, we *check the checker*. Specifically, we relate the model of normalization in the theoretical component and the actual implementations of a type checker and normalization engine, in order to prove that they are equivalent. This allows us to extract a fully verified type-checker to OCaml. The extracted code does not contain any proof witnesses and is comparable to what a skilled human programmer would have written. Furthermore, we connect the extracted type-checker with a front-end parser, so that we can read and type-check MLTT programs from source files using our certified type-checking kernel. In fact, we have used the McTT's extracted type-checker to type-check several small-scale examples to test that it works as intended. At this point, the only unverified component in the compiler pipeline is the lexer, as ocamllex does not support extraction to Rocq.

In summary, McTT provides a framework for building verified kernels for dependent type theories and bridges the gap between the underlying theory and its implementation. Concretely, it makes the following contributions:

- *Check the theory.* We present a mechanized semantics for a core MLTT with natural numbers, Π types, a full cumulative universe hierarchy, and covariant universe subtyping. In particular, we use an impredicative encoding with subtyping in Rocq's `Prop` universe to avoid induction-recursion. Using this encoding, we prove the completeness and soundness of the NbE algorithm using an untyped domain model [Abel 2013] and conclude the logical consistency of the type theory as a corollary.
- *Check the checker.* We implement an algorithmic type checker for our variant of MLTT in Rocq and prove that it is correct with respect to the theoretical specification. In particular, we prove soundness and completeness: an MLTT program is well-formed if and only if it is accepted by the type-checker.
- *Extract a verified type-checker.* From the algorithmic implementation of the type-checker, we extract a readable OCaml implementation which we have used to type-check several small-scale examples. We further embed this verified type-checker within our front-to-back pipeline where users are able to write a program in MLTT in a source file and then use our verified type-checker to check it.

The McTT infrastructure provides a platform for formalizing future extensions of MLTT and derive verified proof checkers. Furthermore, McTT allows us to experiment with new features even before we extend the theoretical and algorithmic components by directly modifying the extracted core type-checker written in OCaml.

The source code and a homepage are available on Github.

2 The McTT Infrastructure

In this section, we give a high-level overview of McTT. A pictorial summary can be found in Fig. 1. The heart of the infrastructure is the *theoretical and algorithmic component* implemented in Rocq.

2.1 Theoretical Component in Rocq

As in prior work by Abel et al. [2018], we focus on a variant of MLTT that includes natural numbers with a primitive recursor and Π types. We model variables using de Bruijn indices and support explicit substitutions. This facilitates reasoning and is common in normalization proofs. While this core fragment of MLTT is well understood, we also push our understanding of how to mechanize the meta-theory of MLTT in Rocq. In particular, we make the following three contributions:

Impredicativity Instead of Induction-Recursion. Modeling a full cumulative universe hierarchy is intuitive: each universe is indexed by a natural number, and lower universes are contained in higher universes. However, mechanizing its meta-theory is non-trivial. Previous mechanizations of MLTT in Agda [Abel et al. 2018; Hu et al. 2023; Pujet and Tabareau 2023] rely on induction-recursion to formalize the semantics, but Rocq does not support induction-recursion. One common workaround to this issue, the Bove-Capretta method [Bove 2009; Bove and Capretta 2005], cannot be not directly applied to full universe hierarchy. Prior Rocq mechanizations of MLTT (e.g. [Adjedj et al. 2024; Wieczorek and Biernacki 2018]) in Rocq restrict MLTT to at most two universes to avoid this issue. A key contribution of our mechanization is an alternative, principled way of modeling the semantics using impredicativity instead of induction-recursion in Rocq. We combine the inductively defined semantics of types and the recursively defined semantics of terms into a single inductive definition of a *weak partial* functional relation whose output is unique up to logical equivalence. This relation itself captures the semantics of types and returns the semantics of terms. This encoding not only provides an alternative way to model semantics without induction-recursion, but also takes advantage of the fact that the output of the functional relation is unique up to logical equivalence to simplify proofs of theorems that are more complex with induction-recursion. This

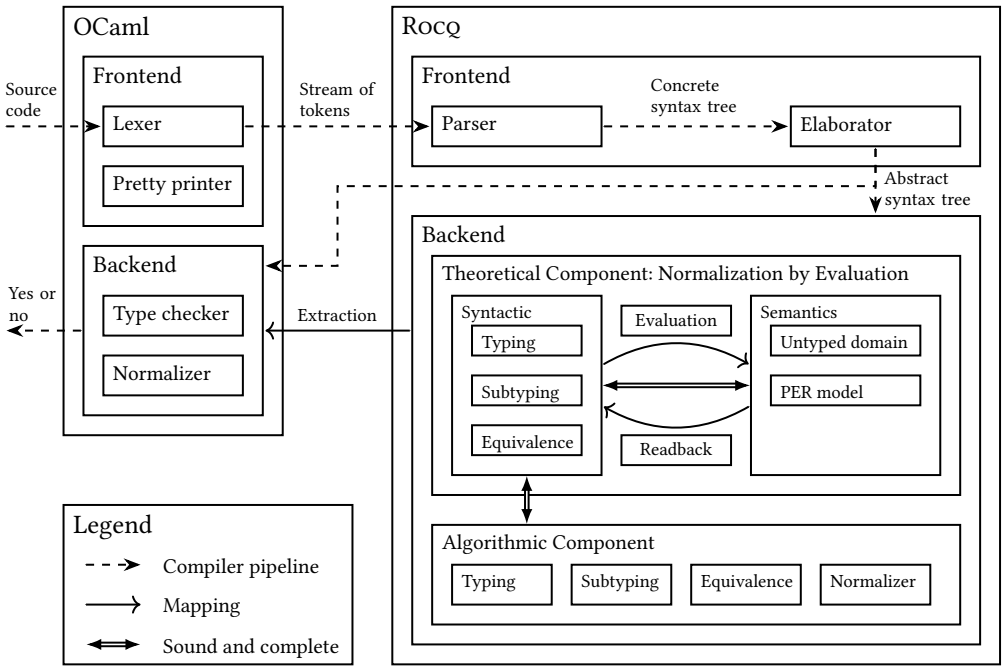

Fig. 1. Structure of McTT

equivalence-based definition further avoids postulating propositional extensionality and reduces the use of functional extensionality (cf. Sec. 4.1).

Universe Subtyping: Pursuing a Complete Algorithmic Type-checking. Universe subtyping is usually supported in CIC, but not in much prior work in MLTT [Abel 2013; Coquand 2018; Hu et al. 2023]. Instead, MLTT simply states: if a type is well-formed at universe level l, then it is also well-formed at universe level $1 + l$. However, we need a more general notion of universe subtyping to establish the soundness and completeness of algorithmic type-checking w.r.t. the declarative typing. For example, suppose that we have the following expression:

$$(\lambda(x{:}\mathsf{Type@1}).\lambda(y{:}\mathsf{Type@0}).y)\ \mathsf{Type@0}\ :\ (\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@1})$$

We first introduce a function with two arguments, where x is not used, and apply the function to $\mathsf{Type@0}$. Declaratively, this expression has type $\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@1}$, because we apply cumulativity to the variable y of $\mathsf{Type@0}$. On the other hand, in a type-checking algorithm, we first infer the type of the function and get its type $\Pi(x{:}\mathsf{Type@1}).\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@0}$. The function application leads to the overall type $\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@0}$ for the expression, but it is not related to the desired type $(\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@1})$. In general, it is challenging for algorithmic type-checking to retrospectively insert cumulativity properly. Using universe subtyping, we can naturally resolve this issue: $\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@0}$ is a subtype of $\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@1}$. Thus, the program has the latter type.

For this reason, we include a *covariant* subtyping relation between universes that propagates only under the outputs of Π types. This extension is sufficient to prove the correctness of algorithmic typing, while keeping the theoretical development manageable.

This example illustrates the value of developing the theory hand in hand with algorithmic implementations of that theory. Algorithmic and implementation concerns often lead to (ad-hoc) changes in the underlying theory, however those changes are not always accounted for in the theoretical development. This can then potentially give rise to subtle soundness bugs that are hard to detect. Developing the algorithmic implementation in a tight loop with the theory within a proof assistant provides a strong net that ensures that the theory is in sync with its implementation.

A Compact Mechanization of NbE. A main meta-theoretical property that we prove about MLTT is normalization based on NbE. NbE offers important advantages compared to the common reducibility candidate approach. In particular, it results in a compact mechanization. For example, our theoretical component of McTT in Rocq is only 12K LoC. This is significantly smaller than the mechanization using reducibility candidates without a full universe hierarchy and without universe subtyping (26k LoC in Rocq) [Adjedj et al. 2024]). Furthermore, NbE immediately provides an evaluation algorithm for (well-typed) programs of the language. In Rocq, we can also extract a usable implementation of a type-checker in OCaml. Last, NbE returns $\beta\eta$ normal forms, which can be compared syntactically. This is in contrast to reduction/rewrite rules in the approaches based on reducibility candidates which only give β normal forms and still require a separate comparison algorithm to handle η expansions [Abel et al. 2018].

2.2 Algorithmic Component in Rocq

In the algorithmic component, we implement a type-checking algorithm. This component consists of three algorithms: a functional implementation of the NbE algorithm, a subtyping algorithm, and a type-checking algorithm. These algorithms are eventually extracted into OCaml and form the kernel of the McTT executable.

We adapt a syntax-directed type-checking algorithm to control the use of subtyping. An object has type A if we are able to infer a type B from it and B is a subtype of A. We hence replace checking whether the type B is convertible to A with checking the subtyping relation. To check the subtyping relation for the types A and B, we first normalize the types A and B using the NbE algorithm. We then check whether the universe levels in the normalized A are always smaller than the levels in the *covariant* positions of normalized B.

We prove the correctness of all the implementations in the algorithmic components with respect to the declarative formulations in the theoretical components. An important theorem of McTT is that an MLTT program is well-formed if and only if it is accepted by the type-checker, hence by the extracted proof checker. As a consequence, we can have full confidence in the extracted proof checker.

2.3 Integration into the Compilation Pipeline

Our theoretical and algorithmic components are connected to a front-end that generates an abstract syntax tree (AST) from a given source file. The OCaml driver of McTT begins with a lexer generated by ocamllex, which produces a stream of tokens from a source file. Since ocamllex does not support extraction to Rocq, the lexer remains an unverified component in the pipeline. The stream of tokens is then forwarded to the parser, which is Rocq code generated by parser generator Menhir. The parser converts the tokens into a concrete syntax tree (CST), if the source file is grammatically well-formed. The parser is proven correct and complete as Menhir also generates the corresponding theorems. Then, the elaborator converts the CST into an abstract syntax tree (AST). At the current stage, the elaborator only converts variables represented in strings into de Bruijn indices. In the future, the elaborator can perform other extensions, such as module name expansions. The

soundness property of the elaborator is captured by a well-scopedness theorem, which says that if all open variables in strings of a CST are well-scoped, then this CST must elaborate to an AST.

The AST is used by both the theoretical and algorithmic components; in particular, it is passed to the extracted type-checker. Using the same AST across all components ensures that every step in the pipeline, including normalization, subtyping, and type-checking, is correct by construction, supported by a sequence of theorems. Finally, a normal form of the input program is pretty-printed.

McTT is designed to minimize the use of unverified components. Currently, unverified components involve the boundaries between the formal and the informal, e.g. the lexer and the pretty-printer. All other components are entirely implemented in Rocq, where we establish a series of soundness and completeness theorems about the parser, the elaborator, and more importantly the type-checker. The infrastructure provided by McTT is a significant stepping stone towards building verified proof checkers for type-theoretic proof assistants.

3 A Core Martin-Löf Type Theory with Universe Hierarchy and Universe Subtyping

In this section, we define the syntax of MLTT, its semantics, and finally the NbE procedure.

3.1 Syntactic Definitions of MLTT

3.1.1 Syntax of MLTT. The syntax of MLTT is similar to that presented by Abel [2013]. It includes natural numbers with a primitive recursor, Π types, closures using explicit substitutions, and a full universe hierarchy.

Variables			x, y, r		
De Bruijn indices	$\mathbb{N}$	$\ni$	d		
Universe levels	$\mathbb{N}$	$\ni$	i, j		
Contexts	Ctx	$\ni$	Γ, Δ	$::=$	$\cdot \mid \Gamma, x : A$
Expressions	Exp	$\ni$	A, B, C, M, N	$::=$	$x_d \mid \mathsf{Nat} \mid \mathsf{Type@}i \mid \Pi(x{:}A).B$
				$\mid$	$\mathsf{zero} \mid \mathsf{succ}\ M \mid \mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ N$
				$\mid$	$\lambda(x{:}A).M \mid M\ N \mid M[\sigma]$
Substitutions	Subst	$\ni$	σ, δ	$::=$	$\mathsf{id} \mid \mathsf{wk} \mid \sigma \circ \delta \mid \sigma, M/x_0$

In the Rocq mechanization, we use de Bruijn indices to represent names. However, for clarity of presentation, we use abstract names in the text. When we need to refer to de Bruijn indices, we put a subscript to the variable, e.g. x_d. The same name is used to denote the same variable with potentially different de Bruijn indices to help readers distinguish variables more easily. We use M and N for expressions that are terms and A, B, and C for those that are types.

The syntax for natural numbers (Nat) includes zero and succ M, the successor of M. The recursion principle $\mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ N$ takes four parameters. First, N is the natural number being recursed on. This number is called *scrutinee*. The base case is handled by M_{zero}, when N is equivalent to zero. If N is succ N', then the recursion is continued by $y, r.M_{\mathsf{succ}}$. In this case, the open variable y is the predecessor, i.e. N' in this case, and r denotes the recursive call on N'. The type $x.A$ represents the *motive*, the return type of the recursion. The overall type of the recursion is $A[N/x]$, i.e. A with x substituted by N.

Next, we have standard dependent function spaces, $\Pi(x{:}A).B$. The introduction form $\lambda(x{:}A).M$ binds a new variable x (or x_0 in de Bruijn indices) of type A that can occur in the body M of the function, while the elimination form $M\ N$ applies the function M to the argument N. Since we have dependent types, the result type from function application is $B[N/x]$.

Explicit substitutions come into play via closures, denoted $M[\sigma]$, which represent the application of the substitution σ to the expression M. We define substitutions following standard practice

$\boxed{\Gamma \vdash t : T}$ Term t has type T in Γ.

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i}{\Gamma \vdash \Pi(x{:}A).B : \mathsf{Type@}i} \qquad \frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda(x : A).M : \Pi(x : A).B}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i \qquad \Gamma \vdash M : \Pi(x : A).B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[\mathsf{id}, N/x]}$$

$$\frac{\Delta \vdash M : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash M[\sigma] : A[\sigma]} \qquad \frac{\Gamma \vdash A' : \mathsf{Type@}i \qquad \Gamma \vdash M : A \qquad \Gamma \vdash A \subseteq A'}{\Gamma \vdash M : A'}$$

$\boxed{\Gamma \vdash A \subseteq B}$ A is a subtype of B.

$$\frac{\Gamma \vdash B : \mathsf{Type@}i \qquad \Gamma \vdash A \approx B : \mathsf{Type@}i}{\Gamma \vdash A \subseteq B} \qquad \frac{\Gamma \vdash A \subseteq A' \qquad \Gamma \vdash A' \subseteq A''}{\Gamma \vdash A \subseteq A''} \qquad \frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \mathsf{Type@}i \subseteq \mathsf{Type@}j}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma \vdash A' : \mathsf{Type@}i \qquad \Gamma \vdash A \approx A' : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i \qquad \Gamma, x : A' \vdash B' : \mathsf{Type@}i \qquad \Gamma, x : A' \vdash B \subseteq B'}{\Gamma \vdash \Pi(x : A).B \subseteq \Pi(x : A').B'}$$

Fig. 2. Selected rules for MLTT

(see, for example, Abel [2013]) using four possible cases. The identity substitution id is a no-op to expressions, and is the left and right identity of composition $\sigma \circ \delta$. The weakening substitution wk weakens the topmost variable, i.e. that of de Bruijn index 0. Finally, we can also extend a substitution with an expression via $\sigma, M/x$.

3.1.2 Judgements of MLTT. Our variant of MLTT is given by seven mutually defined judgements:

- $\boxed{\vdash \Gamma}$: Γ is well-formed.
- $\boxed{\Gamma \vdash M : A}$: M is a well-formed expression of type A in context Γ.
- $\boxed{\Gamma \vdash \sigma : \Delta}$: σ is a well-formed substitution from Δ to Γ.
- $\boxed{\Gamma \vdash M \approx M' : A}$: M and M' are equivalent expressions of type A in context Γ.
- $\boxed{\Gamma \vdash \sigma \approx \sigma' : \Delta}$: σ and σ' are equivalent substitutions from Δ to Γ.
- $\boxed{\Gamma \vdash A \subseteq B}$: Type A is a subtype of B in context Γ.
- $\boxed{\vdash \Gamma \subseteq \Delta}$: Γ is a sub-context of Δ. This judgement generalizes the subtyping judgement.

The rules for MLTT with explicit substitutions are mostly standard (see Fig. 2). We only show the core fragment consisting of functions, function applications, Π-types, and closures. A more exhaustive definition of MLTT can be found in Appendix A (and its mechanized definition).

In the rules, the highlighted premises ensure that well-typed programs always have well-formed types. We prove that they are redundant with a presupposition lemma [Harper and Pfenning 2005], essentially stating that each subcomponent of a judgement must also be well-formed. Since each judgement has different subcomponents, each of them gets their own presupposition lemma. For example, the presuppostion for the typing judgement states that if $\Gamma \vdash M : A$, then $\vdash \Gamma$ and $\Gamma \vdash A : \mathsf{Type@}i$ for some i. For the rest of our discussion, we will ignore the highlighted premises.

Following Rocq's core type theory, we replace the standard type conversion rule with one that allows subsumption and we extend the standard MLTT with a covariant universe subtyping judgement. The subtyping judgement admits the subsumption of smaller universes in larger ones

and propagates this relation down to the output types of Π types (but not the input types). In this way, we implement a full *cumulative* universe hierarchy. The subtyping judgement allows a judgement that holds for a subtype to also hold for its supertype. For example, the following is the subsumption rule for typing:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \subseteq A'}{\Gamma \vdash M : A'}$$

This rule says that if M has type A and A is a subtype of A', then M also has type A'. In the standard MLTT, the associated rule only checks the equivalence between A and A'. In contrast to the symmetry of equivalence, subtyping is asymmetric or uni-directional. In other words, we can only go from A to A' but not the other way.

The reason for introducing subtyping to this system is grounded in establishing completeness of the type-checker w.r.t. the declarative typing rules. Were we to eliminate universe subtyping, the type-checker would admit more terms than the typing rules. Consider a weaker form of cumulativity employed by Abel [2013]; Hu et al. [2023], which does not involve subtyping and is what we originally formulated:

$$\frac{\Gamma \vdash A : \mathsf{Type@}i}{\Gamma \vdash A : \mathsf{Type@}1 + i}$$

Unfortunately, it turns out to be extremely challenging to develop a *complete* type-checking algorithm w.r.t this rule. As the example in the introduction illustrates, negative types, such as Π-types, are sensitive to when we apply this subtyping rule.

The bottom of Fig. 2 defines the subtyping rules. There are only four rules. The first rule says that equivalence between expressions is subsumed by subtyping, and it is effectively a reflexivity rule. The second rule is transitivity, which is expected for a subtyping relation. The third rule admits a smaller universe to be a subtype of a larger universe, thus supporting cumulativity. The final rule allows propagation of subtyping in the output types of Π types. The input types A and A' are equivalent since we do not support contravariant subtyping at this point, following Rocq.

Finally, the subtyping judgement between types is generalized pointwise to context subtyping, denoted as $\vdash \Gamma \subseteq \Delta$. In this judgement, the types in Γ are pointwise subtypes of those in Δ.

3.2 Semantics of MLTT

Next, we consider the semantic model of MLTT, which is intuitively just the untyped λ-calculus. Specifically, we use an untyped domain model à la Abel [2013] to formulate the NbE algorithm. This NbE algorithm has two steps:

(1) First, we evaluate a well-typed term into a domain value (Sec. 3.3.1). During this process, we reduce away the β redexes.
(2) Second, we read from the domain value back to the syntax of an object in normal form. During this process, we perform a type-directed η expansion.

Thus, combining both steps, we obtain the $\beta\eta$ normal form of the given well-typed term. We can define syntactic β-short forms using a standard notion of neutral and normal forms:

$$\begin{array}{lrcl} \text{Neutral forms} & \mathsf{Ne} \ni V & ::= & x_d \mid \mathsf{rec}_{x.W}\ W_{\mathsf{zero}}\ (y, r.W_{\mathsf{succ}})\ V \mid V\ W \\ \text{Normal forms} & \mathsf{Nf} \ni W & ::= & V \mid \mathsf{Nat} \mid \mathsf{Type@}i \mid \Pi(x{:}W_A).W_B \mid \mathsf{zero} \mid \mathsf{succ}\ W \mid \lambda(x{:}W_A).W_M \end{array}$$

Neutral forms are either variables or elimination principles blocked by neutral forms. Normal forms are either neutral, type constructors, or introduction principles applied to normal forms. Note that these normal forms are not necessarily η-long. For example, a variable $x : \Pi(y{:}W_A).W_B$ is normal, but can be η-expanded to $\lambda(y{:}W_A).x\ y$. In order to properly η-expand a term, we need additional type information, which our semantic model will include.

3.2.1 Untyped Domain Model. Following Abel [2013] we give the definition of the untyped domain model. It is the target to which well-typed terms are evaluated to.

$$
\begin{array}{llcll}
\text{De Bruijn levels} & \mathbb{N} & \ni & z & \\
\text{Neutral domain values} & \mathsf{D}^{\mathsf{Ne}} & \ni & v & ::= \quad l_z \mid \mathsf{rec}(A, b, M_{\mathsf{succ}}, \rho, v) \mid v\ w \\
\text{Domain values} & \mathsf{D} & \ni & a, b, m, n & ::= \quad \uparrow^a (v) \mid \mathsf{N} \mid \mathsf{U@}i \mid \Pi(a, B, \rho) \\
 & & & & \quad\ \mid \quad \mathsf{ze} \mid \mathsf{su}(a) \mid \Lambda(M, \rho) \\
\text{Normal domain values} & \mathsf{D}^{\mathsf{Nf}} & \ni & w & ::= \quad \downarrow^a (m) \\
\text{Evaluation environments} & \mathsf{Env} & \ni & \rho &
\end{array}
$$

In contrast to the syntax's de Bruijn indices, domain variables are represented with de Bruijn levels. This means that we count from the left instead of the right, resulting in every variable having a unique de Bruijn level throughout a program, thus removing the need to shift indices. De Bruijn indices and levels have a correspondence. In general, given a context $\Gamma, x{:}T, \Delta$, the de Bruijn index d of x is $|\Delta|$, the length of Δ, whereas the de Bruijn level z of x is $|\Gamma|$. Therefore, we have $d + z + 1 = |\Gamma, x : T, \Delta|$. This correspondence will be used later in the readback function to read de Bruijn indices back from de Bruijn levels. Otherwise, the domain is defined similarly to the (syntactic) neutral and normal forms.

The model is split into three definitions. Neutral domain values are just domain variables (l_z) or elimination forms. They correspond closely to syntactic neutral forms. Domain values can be reflected neutral values annotated with a domain type, $\uparrow^a (v)$, or introduction forms. They correspond closely to the syntactic normal forms, with one major difference: to mechanize dependent functions in the domain values, we employ defunctionalization [Reynolds 1998] to capture the ambient evaluation environment ρ and an open type B or an open term M. This is also standard [Abel 2013; Hu et al. 2023]. Finally, normal domain values are reified domain values annotated with a domain type, $\downarrow^a (m)$. This represents the η-long form of m, although $\downarrow^a (m)$ is not itself η-long since reification is simply a marker. Rather, we will later use a to guide the η-expansions that need to be performed during readback.

Next, an environment ρ is just a function from de Bruijn indices to D. We need a number of tools related to evaluation environments. We omit their concrete definitions for space consideration. First, the empty environment emp :: Env interprets the empty context as a constant function. We use ze as the default value of emp, but the concrete value does not actually matter. Later, we will prove the soundness property, which requires that all variables are bound and implies that we will never access the default value at all. Then, we also need two useful operations on environments: The extension function $\mathtt{ext}(\rho, m)$ shifts everything in ρ up by 1 and produces m on 0; its inverse drop function $\mathtt{drop}(\rho)$ shifts everything in ρ down by 1, forgetting about the previous 0^{th} value.

3.3 Normalization by Evaluation for MLTT

Next, we define the normalization procedure. It consists of two main steps. First, we define evaluation functions that map syntactic objects to semantic objects, eliminating any β-redexes in the process. Second, we define readback functions that map semantic objects back to syntactic objects, performing all the necessary η-expansions in the process. Then, we can compose evaluation and readback to obtain a normalization procedure for computing $\beta\eta$-normal forms.

3.3.1 Evaluation. As the first step of the NbE algorithm, we evaluate a well-typed term into a domain value. Evaluation is a partial function (denoted by $\rightharpoonup$). Since Rocq requires all functions to be total, our implementation models evaluation as functional relations.

$$
\begin{aligned}
[\![\_]\!](\_) &:: \mathsf{Exp} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{D} \\
[\![x_d]\!](\rho) &:= \rho(d) \\
[\![\Pi(x : A).B]\!](\rho) &:= \Pi([\![A]\!](\rho), B, \rho) \\
[\![\lambda(x : A).M]\!](\rho) &:= \Lambda(M, \rho) \\
[\![M\ N]\!](\rho) &:= [\![M]\!](\rho) \cdot [\![N]\!](\rho) \\
[\![t[\sigma]]\!](\rho) &:= [\![t]\!]([\![\sigma]\!](\rho))
\end{aligned}
$$

$$
\begin{aligned}
\_ \cdot \_ &:: \mathsf{D} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{D} \\
(\Lambda(M, \rho)) \cdot a &:= [\![M]\!](\mathtt{ext}(\rho, a)) \\
(\uparrow^{\Pi(a,B,\rho)}(v)) \cdot m &:= \uparrow^{[\![B]\!](\mathtt{ext}(\rho,m))}(v\ \downarrow^{a}(m))
\end{aligned}
$$

Fig. 3. Evaluation functions

We define four partial functions for the whole evaluation procedure.

Evaluation of expressions	$[\![\_]\!](\_) :: \mathsf{Exp} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{D}$
Evaluation of substitutions	$[\![\_]\!]_s(\_) :: \mathsf{Subst} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{Env}$
Domain application	$\_ \cdot \_ :: \mathsf{D} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{D}$
Domain recursion for `Nat`	$\mathsf{rec} \cdot (\_, \_, \_, \_, \_) :: \mathsf{Exp} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{Exp} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{D}$

The first function is the main one: $[\![M]\!](\rho)$ performs evaluation of a syntactic expression M in the evaluation environment ρ. The second function performs evaluation of substitutions into environments. This is required since we use explicit substitutions. The last two functions perform computations inside the domain: $m \cdot n$ is function application and $\mathsf{rec} \cdot (A, m_{\mathsf{zero}}, M_{\mathsf{succ}}, \rho, m)$ is recursion over natural numbers. We show the definition of the main evaluation function together with the Domain application in Fig. 3. The complete definition is available in Appendix A (and its mechanized documentation).

Evaluation is a straightforward process: Every object is evaluated by applying congruences as much as possible, then, on elimination forms, we apply the relevant domain computation. One subtlety occurs in the treatment of open objects, such as functions or the successor branch of the recursor. Specifically, these correspond to premises that are in a larger context than their conclusion, which means that our current evaluation environment ρ does not yet provide an interpretation of the additional variables. Thus, we cannot evaluate them until a later stage when all the necessary information is available. In the meantime, we exploit defunctionalization and simply keep track of the syntactic forms and of the current evaluation environment.

3.3.2 Readback. In the second step of the NbE algorithm, we read the domain values back to a syntactic normal form while performing any necessary η-expansion. The readback process consists of three mutually defined partial functions (see Fig. 4), which are again defined as functional relations in Rocq. First, R^{Nf} reads back a normal domain object to a $\beta\eta$-normal syntactic object. R^{Nf} is the main function and it is responsible for performing the η-expansion. Second, R^{Ty} reads back a type in $\beta\eta$-normal form. R^{Ty} is in particular used to read back domain objects of the form $\downarrow^{\mathsf{U}@i}(a)$. Third, R^{Ne} reads back neutral domain objects to neutral syntactic objects. R^{Ne} is responsible for converting between de Bruijn levels and indices and it is largely defined by congruence rules.

The readback functions take in addition a natural number z as a parameter, which measures the length of current typing context. This number is used in the variable case to convert from de Bruijn levels into de Bruijn indices:

$$\mathsf{R}^{\mathsf{Ne}}_{z'}(l_z) := x_{\max(z'-z-1,0)}$$

The formula has been explained at the beginning of Sec. 3.2.1.

$$
\begin{aligned}
\mathrm{R}^{\mathsf{Nf}}_{\_}(\_) &:: \mathbb{N} \rightharpoonup \mathsf{D}^{\mathsf{Nf}} \rightharpoonup \mathsf{Nf} \\
\mathrm{R}^{\mathsf{Nf}}_{z}(\downarrow^{a}\ (\uparrow^{b}\ (v))) &:= \mathrm{R}^{\mathsf{Ne}}_{z}(v) && \text{(where } a = \uparrow^{a'}(c) \text{ or } \mathsf{N}\text{)} \\
\mathrm{R}^{\mathsf{Nf}}_{z}(\downarrow^{\mathsf{U}@i}\ (a)) &:= \mathrm{R}^{\mathsf{Ty}}_{z}(a) \\
\mathrm{R}^{\mathsf{Nf}}_{z}(\downarrow^{\Pi(a,B,\rho)}\ (m)) &:= \lambda(x : \mathrm{R}^{\mathsf{Ty}}_{z}(a)).\mathrm{R}^{\mathsf{Nf}}_{1+z}(\downarrow^{[\![B]\!](\mathtt{ext}(\rho,\uparrow^{a}(l_z)))}\ (m \cdot \uparrow^{a}\ (l_z)))
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{R}^{\mathsf{Ty}}_{\_}(\_) &:: \mathbb{N} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{Nf} \\
\mathrm{R}^{\mathsf{Ty}}_{z}(\Pi(a,B,\rho)) &:= \Pi(x : \mathrm{R}^{\mathsf{Ty}}_{z}(a)).\mathrm{R}^{\mathsf{Ty}}_{1+z}([\![B]\!](\mathtt{ext}(\rho,\uparrow^{a}\ (l_z)))) \\
\mathrm{R}^{\mathsf{Ty}}_{z}(\uparrow^{a}\ (v)) &:= \mathrm{R}^{\mathsf{Ne}}_{z}(v)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{R}^{\mathsf{Ne}}_{\_}(\_) &:: \mathbb{N} \rightharpoonup \mathsf{D}^{\mathsf{Ne}} \rightharpoonup \mathsf{Ne} \\
\mathrm{R}^{\mathsf{Ne}}_{z'}(l_z) &:= x_{\max(z'-z-1,0)} \\
\mathrm{R}^{\mathsf{Ne}}_{z}(v\ w) &:= \mathrm{R}^{\mathsf{Ne}}_{z}(v)\ \mathrm{R}^{\mathsf{Nf}}_{z}(w)
\end{aligned}
$$

Fig. 4. Definition of readbacks

3.3.3 Normalization. With the evaluation and the readback functions, we are able to provide the NbE algorithm. First, we define a helper function to compute the initial environment:

$$
\begin{aligned}
\uparrow &:: \mathsf{Ctx} \rightharpoonup \mathsf{Env} \\
\uparrow^{\cdot} &:= \mathsf{emp} \\
\uparrow^{\Gamma,x:A} &:= \mathtt{ext}(\uparrow^{\Gamma}, \uparrow^{[\![A]\!}(\uparrow^{\Gamma})\ (l_{|\Gamma|}))
\end{aligned}
$$

Then the NbE algorithm is defined by first evaluating the term and the type, and then reading back the domain value of the term annotated by the domain value of the type.

Definition 3.1. For $\Gamma \vdash M : A$, the NbE algorithm for terms is

$$\mathrm{nbe}^{A}_{\Gamma}(M) := \mathrm{R}^{\mathsf{Nf}}_{|\Gamma|}(\downarrow^{[\![A]\!](\uparrow^{\Gamma})}\ ([\![M]\!](\uparrow^{\Gamma})))$$

We need another variant of the NbE algorithm, which normalizes a type. The steps are similar, but we use R^{Ty} for readback, because we know we are handling a type.

Definition 3.2. For $\Gamma \vdash A : \mathtt{Type}@i$, the NbE algorithm for types is

$$\mathrm{nbe}_{\Gamma}(A) := \mathrm{R}^{\mathsf{Ty}}_{|\Gamma|}([\![A]\!](\uparrow^{\Gamma}))$$

4 Completeness and Soundness of NbE

In this section, we establish that our NbE is a complete and sound algorithm. In this setting, completeness means that NbE gives the same normal form for two equivalent input expressions. On the other hand, soundness means that the normal form out of NbE is equivalent to the input expression. Both properties together imply several important colloraries on typing, subtyping, and equivalence: decidability of equivalence, decidability of subtyping, and completeness and soundness of algorithmic subtyping and type-checking.

However, as the NbE algorithm crosses through a domain value, we need to define a relation model between two domain types/values and another model between a domain value and an expression to show the completeness and soundness.

4.1 Partial Equivalence Relation (PER) Model

We first need to provide the partial equivalence relation (PER) model, which relates two domain types and domain values. In previous work [Abel 2013; Hu et al. 2023], the PER model is defined using induction-recursion [Dybjer and Setzer 2003; Martin-Löf 1984]. However, Rocq does not

support induction-recursion. As an alternative, we use the impredicative `Prop` universe in Rocq. In this universe, we can define a proposition by quantifying over all propositions in `Prop`, including the one being defined. This feature augments the logical power of Rocq enough to express the PER model of a dependent type theory. In fact, Wieczorek and Biernacki [2018] use `Prop` to capture induction-recursion to mechanize NbE of MLTT by adapting the Bove-Capretta method [Bove 2009; Bove and Capretta 2005]. In this approach, their PER model is unrolled into two separate relations, one for domain types and one for domain values. However, this is only possible for a bounded universe hierarchy, and indeed their mechanization is restricted to one universe. This limitation of applying the Bove-Capretta method to universe hierarchy has been noticed by Abel et al. [2017], but their solution is based on sized types, which are not available in Rocq either.

The key intuition of our solution is that we do not need computability of the recursive definition. Previous methods [Bove 2009; Bove and Capretta 2005; Larchey-Wendling and Monin 2018] focus on a faithful encoding of induction-recursion that keeps computability of the recursive function. However, in our case, we use the PER model only to prove properties of NbE and not to define the NbE algorithm. Thus, we can encode the inductive-recursive PER model into a (weak partial) functional relation, which does not directly provide computability. This functional relation is defined inductively and relates two domain types in the universe of a given level. Then, as an output, it "returns" a binary relation between domain values of those related domain types.

To define the PER model, we need a few more auxiliary definitions. First, we define three PERs based on readback, $Nf \subseteq \mathsf{D}^{\mathsf{Nf}} \times \mathsf{D}^{\mathsf{Nf}}$, $Ne \subseteq \mathsf{D}^{\mathsf{Ne}} \times \mathsf{D}^{\mathsf{Ne}}$, and $Ty \subseteq \mathsf{D} \times \mathsf{D}$:

$$\frac{\forall\, z\,.\, \mathsf{R}^{\mathsf{Nf}}_z(w) = \mathsf{R}^{\mathsf{Nf}}_z(w')}{w \approx w' \in Nf} \qquad \frac{\forall\, z\,.\, \mathsf{R}^{\mathsf{Ne}}_z(v) = \mathsf{R}^{\mathsf{Ne}}_z(v')}{v \approx v' \in Ne} \qquad \frac{\forall\, z\,.\, \mathsf{R}^{\mathsf{Ty}}_z(a) = \mathsf{R}^{\mathsf{Ty}}_z(a')}{a \approx a' \in Ty}$$

Nf relates two normal domain values if and only if their readbacks are equal in any given context. Similarly, *Ne* relates two neutral domain values based on readbacks, and *Ty* relates two domain types based on readbacks. We use these PERs as base cases for the PER model. Furthermore, we describe the realizability of the PER model using these PERs, which leads to the completeness theorem of NbE. Therefore, they also play the key roles for the completeness theorem as well.

Now, we first define the PER model $Neu \subseteq \mathsf{D} \times \mathsf{D}$ for domain values of neutral domain types.

$$\frac{w \approx w' \in Ne}{\uparrow^{a}(w) \approx \uparrow^{a'}(w') \in Neu}$$

This relation *Neu* simply ignores the type annotations and relates two neutral domain values using *Ne*. We can safely ignore these type annotations as we will use *Neu* only for well-typed terms and check the relation between types first with our PER model for domain types. Next, we define the PER model $Nat \subseteq \mathsf{D} \times \mathsf{D}$ for domain values of the natural number type.

$$\frac{}{\mathsf{ze} \approx \mathsf{ze} \in Nat} \qquad \frac{m \approx n \in Nat}{\mathsf{su}(m) \approx \mathsf{su}(n) \in Nat} \qquad \frac{v \approx v' \in Ne}{\uparrow^{a}(v) \approx \uparrow^{a'}(v') \in Nat}$$

This relates ze with itself, su(—) of related natural numbers, and two neutral domain values. Again, the case of neutral domain values ignores the type annotations.

Now, we can define the PER model $\mathcal{U}_i$ for domain types in the i-th universe. We write

$$a \approx a' \in \mathcal{U}_i \searrow R$$

to relate domain types a and a', and a binary relation R relating two domain values of types a and a'. The relation R is regarded as a "return" predicate, so $\mathcal{U}_i$ is a functional relation. We write $a \approx a' \in \mathcal{U}_i$ if we are only concerned about the relation between a and a'. The definition is given in Fig. 5. There are 4 cases: one for types in neutral forms and one for each type constructor

$$\frac{b \approx b' \in Ne \qquad R \Leftrightarrow Neu}{\uparrow^a(b) \approx \uparrow^{a'}(b') \in \mathcal{U}_i \searrow R} \qquad \frac{j < i \qquad R \Leftrightarrow \mathcal{U}_j}{\mathsf{U@}j \approx \mathsf{U@}j \in \mathcal{U}_i \searrow R} \qquad \frac{R \Leftrightarrow Nat}{\mathsf{N} \approx \mathsf{N} \in \mathcal{U}_i \searrow R}$$

$$\frac{\begin{array}{c} a \approx a' \in \mathcal{U}_i \searrow R_I \\ R_I \text{ is a PER} \qquad \forall\, \mathcal{D} :: m \approx m' \in R_I \,.\, [\![B]\!](\mathtt{ext}(\rho, m)) \approx [\![B']\!](\mathtt{ext}(\rho', m')) \in \mathcal{U}_i \searrow R_O(\mathcal{D}) \\ \forall\, n, n' \,.\, n \approx n' \in R \leftrightarrow (\forall\, \mathcal{D} :: m \approx m' \in R_I \,.\, n \cdot m \approx n' \cdot m' \in R_O(\mathcal{D})) \end{array}}{\Pi(a, B, \rho) \approx \Pi(a', B', \rho') \in \mathcal{U}_i \searrow R}$$

Fig. 5. PER Model for Domain Types and Values

$$\frac{b \approx b' \in Ne}{\uparrow^a(b) <:_i \uparrow^{a'}(b')} \qquad \frac{j \le k < i}{\mathsf{U@}j <:_i \mathsf{U@}k} \qquad \frac{}{\mathsf{N} <:_i \mathsf{N}}$$

$$\frac{\begin{array}{c} a \approx a' \in \mathcal{U}_i \searrow R_I \qquad \forall\, \mathcal{D} :: m \approx m' \in R_I \,.\, [\![B]\!](\mathtt{ext}(\rho, m)) <:_i [\![B']\!](\mathtt{ext}(\rho', m')) \\ \Pi(a, B, \rho) \approx \Pi(a, B, \rho) \in \mathcal{U}_i \qquad \Pi(a', B', \rho') \approx \Pi(a', B', \rho') \in \mathcal{U}_i \end{array}}{\Pi(a, B, \rho) <:_i \Pi(a', B', \rho')}$$

Fig. 6. Semantic Subtyping

(U@j, N, and Π). In the Π case, we have one highlighted premise. This premise enhances the induction principle for $\mathcal{U}$ so that we can prove $\mathcal{U}$ and its return relation R are actually PERs without postulating any axioms. After we prove that R is actually PER, we remove this highlighted premise as it is logically redundant.

It is worth noting that we define $\mathcal{U}$ as a *weak* partial functional relation, in the sense that the return relation R is only unique up to relational equivalence. Here, the relational equivalence $P \Leftrightarrow Q$ between two n-ary relations P and Q is defined as

$$\forall\, x_1, x_2, \ldots, x_n \,.\, P\ x_1\ x_2\ \ldots\ x_n \leftrightarrow Q\ x_1\ x_2\ \ldots\ x_n$$

We employ this weakness to avoid excessive uses of extensionality axioms, especially propositional and functional extensionality. In fact, this approach allows us to remove propositional extensionality entirely, and only keep one use of functional extensionality for convenience.

Given the definition, it becomes clear why $\mathcal{U}$ is encoded in the impredicative universe `Prop`. In the universe case, $\mathcal{U}_i$ returns R, a relation equivalent to $\mathcal{U}_j$ for $j < i$. In a predicative setting, this would have required $\mathcal{U}_i$ to live in a higher universe than $\mathcal{U}_j$, but this is not possible because they are the same inductive definition. In `Prop`, this problem no longer exists thanks to impredicativity.

In addition to the PER model, since we are working with universe subtyping in the syntactic judgements, we also need to introduce subtyping to the semantics as well. The semantic subtyping relation $a <:_i a'$ between two domain types a and a' at universe level i is defined in Fig. 6. The four rules in semantic subtyping reflects the definition of syntactic subtyping. For example, in the Π case, the input types a and a' are semantically equivalent by $\mathcal{U}$, because we employ covariant subtyping in our syntactic rules.

Finally, the PER model and semantic subtyping are extended to PER and semantic subtyping between contexts and environments in order to define the semantic typing judgement for the completeness theorem. We list the cons rule of the PER $\Gamma \approx \Gamma' \searrow R$ between contexts Γ and Γ' and the cons rule of semantic subtyping between contexts $\Gamma <: \Gamma'$ in Fig. 7. We omit the rules for the empty context since they simply relate the empty context to itself. Note that the PER between contexts also gives the PER R for environments as a "returned" value, similarly to the definition of $\mathcal{U}$. In case of the empty context, this returned PER is the total relation, i.e. any two environments are related under the PER. Another subtle but important point in the definition is that universe

$$\frac{\Gamma \approx \Gamma' \searrow R_T \qquad R_T \text{ is a PER} \qquad \forall\, \mathcal{D} :: \rho \approx \rho' \in R_T \,.\, [\![A]\!](\rho) \approx [\![A']\!](\rho') \in \mathcal{U}_i \searrow R_H(\mathcal{D}) \qquad \forall\, \rho, \rho' \,.\, \rho \approx \rho' \in R \leftrightarrow (\forall\, \mathcal{D} :: \mathsf{drop}(\rho) \approx \mathsf{drop}(\rho') \in R_T \,.\, \rho(0) \approx \rho'(0) \in R_H(\mathcal{D}))}{\Gamma, x{:}A \approx \Gamma', x{:}A' \searrow R}$$

$$\frac{\Gamma <: \Gamma' \qquad \Gamma \approx \Gamma \searrow R_T \qquad \forall\, \mathcal{D} :: \rho \approx \rho' \in R_T \,.\, [\![A]\!](\rho) <:_i [\![A']\!](\rho') \qquad \Gamma, x{:}A \approx \Gamma, x{:}A \searrow R_1 \qquad \Gamma', x{:}A' \approx \Gamma', x{:}A' \searrow R_2}{\Gamma, x{:}A <: \Gamma', x{:}A'}$$

Fig. 7. The Cons Rule of the PER Model and Semantic Subtyping for Contexts

level i, in which we relate the heads $[\![A]\!](\rho)$ and $[\![A']\!](\rho')$, is hidden in the conclusion. This forces us to prove a slightly more general property about $\mathcal{U}$ in order to prove that the PER model for contexts is indeed transitive in Sec. 4.2.2.

In the following section, we prove the properties of the PER model for domain types and values leading to the completeness theorem.

4.2 Properties of the PER Model

4.2.1 Irrelevance. To show the interesting properties of the PER models, we first need to specify what determines R in the definition of $\mathcal{U}$. Our intuition is that we do not need a full derivation of $a \approx b \in \mathcal{U}_i \searrow R$ to determine R, but having a domain type a that satisfies $a \approx b \in \mathcal{U}_i$ is enough. The following lemma formalizes this intuition:

Lemma 4.1 ($\mathcal{U}$ Right Irrelevance). *If $a \approx b \in \mathcal{U}_i \searrow R$ and $a \approx b' \in \mathcal{U}_{i'} \searrow R'$, then $R \Leftrightarrow R'$.*

Thus, as long as we have the domain type a, the relation between domain values of that type is unique up to relational equivalence.

4.2.2 Structural Property of the PER Model. Now, we show the structural properties of the PER model and semantic subtyping: the PER model is indeed a PER and semantic subtyping is transitive. We start by showing that the PER model is symmetric, both for domain types and values.

Lemma 4.2 (Symmetry of $\mathcal{U}$). *If $a \approx a' \in \mathcal{U}_i \searrow R$, then*

- *$a' \approx a \in \mathcal{U}_i \searrow R$, and,*
- *$m \approx m' \in R$ implies $m' \approx m \in R$.*

With this, we can show that it is also transitive.

Lemma 4.3 (Transitivity of $\mathcal{U}$). *If $a_1 \approx a_2 \in \mathcal{U}_i \searrow R$, then*

- *$a_2 \approx a_3 \in \mathcal{U}_j \searrow R$ implies $a_1 \approx a_3 \in \mathcal{U}_i \searrow R$, and,*
- *$m_1 \approx m_2 \in R$ and $m_2 \approx m_3 \in R$ implies $m_1 \approx m_3 \in R$*

Note that this theorem is stronger than the naïve transitivity because we can choose a different universe level j for $a_2 \approx a_3 \in \mathcal{U}_j \searrow R$. This flexibility is the key for the transitivity of the PER for contexts, as three transitively related contexts might use different universe levels to relate their heads. The transitivity of the semantic subtyping for domain types, the symmetry and the transitivity of the PER model for contexts and environments, and the transitivity of the semantic subtyping for contexts follow from the above two in this order.

4.2.3 Realizability of the PER Model. Another important property of $\mathcal{U}$ is realizability, which is essential to establish both completeness and soundness. We define the realizability using *Nf*, *Ne*, and *Ty* following Abel [2013]:

Theorem 4.4 (Realizability of PER Model). *Given $a \approx a' \in \mathcal{U}_i \searrow R$,*

- *For domain values:* $m \approx m' \in R$ *implies* $\downarrow^a(m) \approx \downarrow^{a'}(m') \in Nf$;
- *For neutral domain values:* $w \approx w' \in Ne$ *implies* $\uparrow^a(w) \approx \uparrow^{a'}(w') \in R$;
- *For domain types:* $a \approx a' \in Ty$.

Proof. By induction on $a \approx a' \in \mathcal{U}_i \searrow R$. We need to prove a lemma that $m \approx m' \in Nat$ implies $\downarrow^{\mathsf{N}} m \approx \downarrow^{\mathsf{N}} m' \in Nf$ along the way, which is shown by induction on $m \approx m' \in Nat$. □

4.2.4 Other Properties of the PER Model. To show the fundamental theorem for completeness (that syntactic typing rules are admissible for semantic equivalence judgement for completeness), we need some lemmas about $\mathcal{U}$. Particularly, we need that $\mathcal{U}$ is cumulative in its universe level.

Lemma 4.5 (Cumulativity of $\mathcal{U}$). *If* $a \approx a' \in \mathcal{U}_i \searrow R$, *then* $a \approx a' \in \mathcal{U}_j \searrow R$ *for* $j \geq i$.

Another key property is that the PER respects subtyping, which is crucial for our subtyping rule.

Lemma 4.6 (Subtyping of $\mathcal{U}$). *If* $a <:_i a'$, $a \approx b \in \mathcal{U}_i \searrow R$, *and* $a' \approx b' \in \mathcal{U}_i \searrow R'$ *then* $m \approx m' \in R$ *implies* $m \approx m' \in R'$.

4.3 Completeness

We first define the semantic equivalence judgement $\Gamma \vDash M \approx M' : A$ for completeness:

$$\frac{\Gamma \approx \Gamma' \searrow R_E \qquad \forall\, \mathcal{D} :: \rho \approx \rho' \in R_E \;.\; [\![A]\!](\rho) \approx [\![A]\!](\rho') \in \mathcal{U}_i \searrow R \;\wedge\; [\![M]\!](\rho) \approx [\![M']\!](\rho') \in R}{\Gamma \vDash M \approx M' : A}$$

From this, we derive the semantic typing judgement $\Gamma \vDash M : A$ by defining it as an alias of $\Gamma \vDash M \approx M : A$. Then, we use the lemmas for $\mathcal{U}$ to show the following fundamental theorem of the semantic judgements for completeness:

Theorem 4.7 (Fundamental Theorem). *If* $\Gamma \vdash M \approx M' : A$ *then* $\Gamma \vDash M \approx M' : A$.

Proof. By induction on the given equivalence derivation. This should be mutually proven with fundamental theorems for other judgements such as one for $\Gamma \vdash M : A$. In our mechanization, we handle each case as a separate lemma as they are fairly long. □

Now, we can derive the completeness of NbE:

Theorem 4.8 (Completeness of NbE). *If* $\Gamma \vdash M \approx M' : A$ *then* $nbe^A_\Gamma(M) = nbe^A_\Gamma(M')$.

Proof. We first show a lemma that $\Gamma \approx \Gamma' \searrow R_E$ implies $\uparrow^\Gamma \approx \uparrow^{\Gamma'} \in R_E$ by induction on the given derivation. Then, we prove the main goal by applying the fundamental theorem together with this lemma □

4.4 Kripke Gluing Model

Now, we move our focus to the soundness of NbE. For soundness, we need to connect a term and its normalization result that was read back from the semantic world. Thus, the model for soundness should "glue" the syntactic types/terms with domain types/values [Coquand and Dybjer 1997]. Moreover, as we need to deal with open terms, our model should be stable under weakenings. Following Abel [2013], we call this condition "Kripke". In short, we will define a Kripke gluing model for McTT in this section.

First, we need to formalize the definition of weakenings. The following judgement $\Gamma \vdash_w \sigma : \Delta$ means that σ is a weakening from Δ to Γ.

$$\frac{\Gamma \vdash \sigma \approx \mathsf{id} : \Delta}{\Gamma \vdash_w \sigma : \Delta} \qquad \frac{\Gamma \vdash_w \tau : \Delta', x{:}A \qquad \vdash \Delta' \subseteq \Delta \qquad \Gamma \vdash \sigma \approx \mathsf{wk} \circ \tau : \Delta}{\Gamma \vdash_w \sigma : \Delta}$$

$$\frac{P \Leftrightarrow Neu_i^a \qquad El \Leftrightarrow Neu_i^a}{\uparrow^b a \in \mathcal{U}_i^* \searrow P \searrow El} \qquad \frac{P \Leftrightarrow Nat_i \qquad El \Leftrightarrow Nat_i}{\mathsf{N} \in \mathcal{U}_i^* \searrow P \searrow El}$$

$$\frac{\begin{array}{c} j < i \qquad \forall\, \Gamma, A\,.\, \Gamma \vdash A \circledR P \leftrightarrow (\Gamma \vdash A \approx \mathsf{Type@}j : \mathsf{Type@}i) \\ \forall\, \Gamma, M, A, m. \\ \Gamma \vdash M : A \circledR m \in El \leftrightarrow (\Gamma \vdash M : A \wedge \Gamma \vdash A \circledR P \wedge m \in \mathcal{U}_j^* \searrow P' \searrow El' \wedge \Gamma \vdash M \circledR P') \end{array}}{\mathsf{U@}j \in \mathcal{U}_i^* \searrow P \searrow El}$$

$$\frac{\begin{array}{c} a \in \mathcal{U}_i^* \searrow P_I \searrow El_I \\ a \approx a \in \mathcal{U}_i \searrow R_I \qquad \forall\, \mathcal{D} :: n \approx n \in R_I\,.\, [\![B]\!](\mathtt{ext}(\rho, n)) \in \mathcal{U}_i^* \searrow P_O(\mathcal{D}) \searrow El_O(\mathcal{D}) \\ \Pi(a, B, \rho) \approx \Pi(a, B, \rho) \in \mathcal{U}_i \searrow R \qquad P \Leftrightarrow \Pi_i(R_I, P_I, El_I, P_O) \qquad El \Leftrightarrow \Pi_i(R, R_I, P_I, El_I, El_O) \end{array}}{\Pi(a, B, \rho) \in \mathcal{U}_i^* \searrow P \searrow El}$$

Fig. 8. Gluing Model for Domain Types and Values

Note that the rule for wk also handles context subtyping. This allows us to apply subtyping between contexts to weakening judgement without a separate rule for subtyping.

Now, we define the Kripke gluing model. Usually, one defines the model using a recursion over a derivation of the PER model for domain types. However, our $\mathcal{U}$ is defined in Prop, and we cannot match on it to generate a new predicate.[3] Thus, we need to define an inductive predicate $\mathcal{U}^*$ reusing cases of $\mathcal{U}$. There are two main differences between $\mathcal{U}^*$ and $\mathcal{U}$: 1) $\mathcal{U}^*$ is a unary relation on domain types, while $\mathcal{U}$ is binary; 2) $\mathcal{U}^*$ outputs two predicates, while $\mathcal{U}$ only outputs one. More precisely, the judgement $a \in \mathcal{U}_i^* \searrow P \searrow El$ returns a unary predicate P for syntactic types related to a and a binary predicate El for syntactic terms related to a domain value of type a. To give a clear definition of $\mathcal{U}^*$, we first need to specify concrete predicates P and El. We will use notations $\Gamma \vdash A \circledR P$ for gluing of a syntactic type A, and $\Gamma \vdash M : A \circledR m \in El$ for gluing of the syntactic term M and domain value m.

For the neutral case, we define Neu_i^a that relates a syntactic type A under a context Γ with a given neutral domain type a and universe level i ($\Gamma \vdash A \circledR Neu_i^a$). For brevity, Neu_i^a is also overloaded for the relation that relates a syntactic term M of type A with a domain value m ($\Gamma \vdash M : A \circledR m \in Neu_i^a$).

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \forall\, \Delta, \sigma\,.\, \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash A[\sigma] \approx \mathsf{R}^{\mathsf{Ne}}_{|\Delta|}(a) : \mathsf{Type@}i}{\Gamma \vdash A \circledR Neu_i^a}$$

$$\frac{\Gamma \vdash A \circledR Neu_i^a \quad \Gamma \vdash M : A \quad v \approx v \in Ne \quad \forall\, \Delta, \sigma\,.\, \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash M[\sigma] \approx \mathsf{R}^{\mathsf{Ne}}_{|\Delta|}(v) : \mathsf{Type@}i}{\Gamma \vdash M : A \circledR \uparrow^{a'} (v) \in Neu_i^a}$$

Note that both $\Gamma \vdash A \circledR Neu_i^a$ and $\Gamma \vdash M : A \circledR m \in Neu_i^a$ require domain and syntax to be related even after an arbitrary weakening from Γ. We repeat this requirement for the N case whenever neutral terms appear. We use the name Nat_i for the relation that checks syntactic type A is a natural number type ($\Gamma \vdash A \circledR Nat_i$) and for the relation between syntactic natural number term M with domain natural number m ($\Gamma \vdash M : A \circledR m \in Nat_i$). This relation needs to be inductive, and we

[3] It is possible to match on Prop to generate a proof of a predicate of Prop. However, Prop itself lives in Type, and one cannot generate a member of a type by pattern matching on Prop.

$$\frac{\forall \Delta, \sigma, \rho \,.\, \Delta \vdash \sigma \mathrel{®} \rho \in Sb \leftrightarrow \Delta \vdash \sigma : \cdot}{\cdot \searrow Sb}$$

$$\frac{\Gamma \searrow Sb_T \qquad \Gamma \vdash A : \mathsf{Type@}i \qquad Sb \Leftrightarrow Cons_i(\Gamma, A, Sb_T) \qquad \forall \Delta, \sigma, \rho, P, El \,.\, \Delta \vdash \sigma \mathrel{®} \rho \in Sb_T \to [\![A]\!](\rho) \in \mathcal{U}_i^* \searrow P \searrow El \wedge \Delta \vdash A[\sigma] \mathrel{®} P}{\Gamma, x{:}A \searrow Sb}$$

Fig. 9. Gluing Model for Contexts

extract the inductive part into $\Gamma \vdash M \mathrel{®} m \in Nat$.

$$\frac{\Gamma \vdash A \approx \mathsf{Nat} : \mathsf{Type@}i}{\Gamma \vdash A \mathrel{®} Nat_i} \qquad \frac{\Gamma \vdash A \mathrel{®} Nat_i \qquad \Gamma \vdash M \mathrel{®} m \in Nat}{\Gamma \vdash M : A \mathrel{®} m \in Nat_i} \qquad \frac{\Gamma \vdash M \approx \mathsf{zero} : \mathsf{Nat}}{\Gamma \vdash M \mathrel{®} \mathsf{ze} \in Nat}$$

$$\frac{\Gamma \vdash M \approx \mathsf{succ}\ M' : \mathsf{Nat} \qquad \Gamma \vdash M' \mathrel{®} m' \in Nat}{\Gamma \vdash M \mathrel{®} \mathsf{su}(m') \in Nat} \qquad \frac{v \approx v \in Ne \qquad \forall\, \Delta, \sigma \,.\, \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash M[\sigma] \approx \mathsf{R}^{\mathsf{Ne}}_{|\Delta|}(v) : \mathsf{Type@}i}{\Gamma \vdash M \mathrel{®} \uparrow^a (w) \in Nat}$$

Likewise, we define $\Pi_i(R_I, P_I, El_I, P_O)$ for Π-types and $\Pi_i(R, R_I, P_I, El_I, El_O)$ for terms of Π-types, where the subscripts I and O indicate a relation on inputs and outputs, respectively, and R is the PER associated to the whole function space:

$$\frac{\Gamma \vdash A \approx \Pi(x{:}B).C : \mathsf{Type@}i \qquad \forall\, \Delta, \sigma \,.\, \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash B[\sigma] \mathrel{®} P_I \qquad \forall\, \Delta, \sigma, N, n \,.\, \Delta \vdash_w \sigma : \Gamma \wedge \Delta \vdash N : B[\sigma] \mathrel{®} n \in El_I \to \forall\, \mathcal{D} :: n \approx n \in R_I \,.\, \Delta \vdash C[\sigma, N] \mathrel{®} P_O(\mathcal{D})}{\Gamma \vdash A \mathrel{®} \Pi_i(R_I, P_I, El_I, P_O)}$$

$$\frac{\Gamma \vdash M : A \qquad m \approx m \in R \qquad \Gamma \vdash A \approx \Pi(x{:}B).C : \mathsf{Type@}i \qquad \forall\, \Delta, \sigma \,.\, \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash B[\sigma] \mathrel{®} P_I \qquad \forall\, \Delta, \sigma, N, n \,.\, \Delta \vdash_w \sigma : \Gamma \wedge \Delta \vdash N : B[\sigma] \mathrel{®} n \in El_I \to \forall\, \mathcal{D} :: m \approx m \in R_I \,.\, \Delta \vdash M[\sigma]\ N : C[\sigma, N] \mathrel{®} m \cdot n \in El_O(\mathcal{D})}{\Gamma \vdash M : A \mathrel{®} m \in \Pi_i(R, R_I, P_I, El_I, El_O)}$$

Using these predicates, we give the definition of $\mathcal{U}^*$ in Fig. 8.

We also need to extend the gluing model to one for contexts for soundness. In Fig. 9, we define $\Gamma \searrow Sb$. Here, Sb is a predicate relating a syntactic substitution σ from Δ to Γ with an environment ρ. To represent this, we use the notation $\Delta \vdash \sigma \mathrel{®} \rho \in Sb$. This definition uses the following predicate $Cons_i(\Gamma, A, Sb_T)$:

$$\frac{\Delta \vdash \sigma : \Gamma, x{:}A \qquad [\![A]\!](\rho) \in \mathcal{U}_i^* \searrow P \searrow El \qquad \Delta \vdash x[\sigma] : A[\mathsf{wk} \circ \sigma] \mathrel{®} \rho(0) \in El \qquad \Delta \vdash \mathsf{wk} \circ \sigma \mathrel{®} \mathsf{drop}(\rho) \in Sb_T}{\Gamma \vdash \sigma \mathrel{®} \rho \in Cons_i(\Gamma, x, A, Sb_T)}$$

This predicate relates the substitution entry for latest variable in the context ($x[\sigma]$) with the top-most value in the environment ($\rho(0)$), and relate weakened substitution ($\mathsf{wk} \circ \sigma$) with the shifted environment ($\mathsf{drop}(\rho)$).

In the following section, we prove the properties of the Kripke gluing model for domain types and values leading to the soundness theorem.

4.5 Properties of the Kripke Gluing Model

4.5.1 Monotonicity. As mentioned previously, this model is Kripke in the sense that it is stable under weakenings. We can formalize this property as follows:

$$\frac{\Gamma \vdash M : A \qquad a \in \mathcal{U}_i^* \searrow P \searrow El \qquad \Gamma \vdash A \circledR P \qquad \downarrow^a m \approx \downarrow^a m \in Nf \qquad \forall \Delta, \sigma\ .\ \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash M[\sigma] \approx \mathrm{R}^{\mathsf{Nf}}_{|\Delta|}(\downarrow^a m) : A[\sigma]}{\Gamma \vdash M : A \circledR m \in Nf_i^a}$$

$$\frac{\Gamma \vdash M : A \qquad a \in \mathcal{U}_i^* \searrow P \searrow El \qquad \Gamma \vdash A \circledR P \qquad w \approx w \in Ne \qquad \forall \Delta, \sigma\ .\ \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash M[\sigma] \approx \mathrm{R}^{\mathsf{Ne}}_{|\Delta|} w : A[\sigma]}{\Gamma \vdash M : A \circledR w \in Ne_i^a}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad a \approx a \in Ty \qquad \forall \Delta, \sigma\ .\ \Delta \vdash_w \sigma : \Gamma \to \Delta \vdash A[\sigma] \approx \mathrm{R}^{\mathsf{Ty}}_{|\Delta|} w : \mathsf{Type@}i}{\Gamma \vdash A \circledR Ty_i^a}$$

Fig. 10. The Definition of Nf_i^a, Ne_i^a, and Ty_i^a

Lemma 4.9 (Monotonicity). *Given $a \in \mathcal{U}_i^* \searrow P \searrow El$,*

- *For types: $\Gamma \vdash A \circledR P$ and $\Delta \vdash_w \sigma : \Gamma$ imply $\Delta \vdash A[\sigma] \circledR P$,*
- *For terms: $\Gamma \vdash M : A \circledR m \in El$ and $\Delta \vdash_w \sigma : \Gamma$ imply $\Delta \vdash M[\sigma] : A[\sigma] \circledR m \in El$.*

4.5.2 Realizability. As in completeness, the realizability of the Kripke gluing model is a key step towards the soundness theorem. To state and prove the realizability of the Kripke gluing model, we first need to introduce gluing versions of *Nf*, *Ne*, and *Ty*, called Nf_i^a, Ne_i^a, and Ty_i^a. Their definitions are given in Fig. 10.

Note that two main differences exist compared to the previous *Nf*, *Ne*, and *Ty* for completeness. *Nf*, *Ne*, and *Ty* for completeness relate the readbacks of two domain terms/types. On the other hand, the Nf_i^a, Ne_i^a, and Ty_i^a for soundness relate the readback of a domain term/type and a syntactic term/type. More importantly, *Nf*, *Ne*, and *Ty* for completeness relate normal forms using the equality on their syntactic forms whereas Nf_i^a, Ne_i^a, and Ty_i^a use typed equivalence to compare them. These changes allow us to prove the soundness theorem, as the theorem states typed equivalence between a term and its normal form.

Now, we can state the following realizability theorem:

Theorem 4.10 (Realizability of Gluing Model). *Given $a \in \mathcal{U}_i^* \searrow P \searrow El$,*

- *For domain values: $\Gamma \vdash M : A \circledR m \in El$ implies $\Gamma \vdash M : A \circledR m \in Nf_i^a$;*
- *For neutral domain values: $\Gamma \vdash M : A \circledR w \in Ne_i^a$ implies $\Gamma \vdash M : A \circledR \uparrow^a w \in El$;*
- *For domain types: $\Gamma \vdash A \circledR P$ implies $\Gamma \vdash A \circledR Ty_i^a$.*

Proof. By induction on $a \in \mathcal{U}_i^* \searrow P \searrow El$. As in the realizability of the PER model, we need a lemma to deal with $\Gamma \vdash M \circledR m \in Nat$, which is proven by induction on $\Gamma \vdash M \circledR m \in Nat$. □

4.5.3 Other Properties of the Kripke Gluing Model. Again, the fundamental theorem for soundness requires the cumulativity of the gluing model as its key property. However, as the Π case of the gluing model uses *El* in a negative position, we also need a "lowering" lemma that brings a term into a lower universe. The formal statement of the theorem follows:

Lemma 4.11 (Cumulativity and Lowering of Gluing Model). *Given $a \in \mathcal{U}_i^* \searrow P \searrow El$ and $a \in \mathcal{U}_j^* \searrow P' \searrow El'$ for $i \leq j$,*

- *Cumulativity for types: $\Gamma \vdash A \circledR P$ implies $\Gamma \vdash A \circledR P'$;*
- *Cumulativity for terms: $\Gamma \vdash M : A \circledR m \in El$ implies $\Gamma \vdash M : A \circledR m \in El'$;*
- *Lowering for terms: $\Gamma \vdash A \circledR P$ and $\Gamma \vdash M : A \circledR m \in El'$ implies $\Gamma \vdash M : A \circledR m \in El$.*

We also need a lemma about interaction between the semantic subtyping and the gluing model. First, if two domain types are in the semantic subtyping relation, then glued syntactic types are also in subtyping relation:

Lemma 4.12 (Subtyping of Gluing Model for Types). *If* $a <:_i a'$, $a \in \mathcal{U}_i^* \searrow P \searrow El$, *and* $a' \in \mathcal{U}_i^* \searrow P' \searrow El'$ *then* $\Gamma \vdash A \circledR P$ *and* $\Gamma \vdash A' \circledR P'$ *implies* $\Gamma \vdash A \subseteq A'$.

Also, the term predicates from the gluing model respect subtyping relation.

Lemma 4.13 (Subtyping of Gluing Model for Terms). *If* $a <:_i a'$, $a \in \mathcal{U}_i^* \searrow P \searrow El$, *and* $a' \in \mathcal{U}_i^* \searrow P' \searrow El'$ *then* $\Gamma \vdash A' \circledR P'$ *and* $\Gamma \vdash M : A \circledR m \in El$ *implies* $\Gamma \vdash M : A' \circledR m \in El'$.

4.6 Soundness

As for the completeness, we first define the semantic typing judgement $\Gamma \Vdash M : A$ for soundness:

$$\frac{\Gamma \searrow Sb \qquad \forall\, \Delta, \sigma, \rho, P, El\ .\ \Delta \vdash \sigma \circledR \rho \in Sb \to [\![A]\!](\rho) \in \mathcal{U}_i^* \searrow P \searrow El \wedge \Delta \vdash M[\sigma] : A[\sigma] \circledR [\![M]\!](\rho) \in El}{\Gamma \Vdash M : A}$$

Then, we show the following fundamental theorem of the semantic typing judgements for soundness:

Theorem 4.14 (Fundamental Theorem for Soundness). *If* $\Gamma \vdash M : A$ *then* $\Gamma \Vdash M : A$.

Proof. By induction on the given typing derivation. As in the case of the completeness, we prove each case separately due to its complexity. □

We can derive the soundness of NbE from this:

Theorem 4.15 (Soundness of NbE). *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash M \approx nbe_\Gamma^A(M') : A$

Proof. We first show a lemma that $\Gamma \searrow Sb$ implies $\Gamma \vdash \mathsf{id} \circledR \uparrow^\Gamma \in Sb$ by induction on the given derivation. Then, we prove the main goal by applying the fundamental theorem and this lemma. □

4.7 Key Corollaries

The fundamental theorems for the PER and gluing model allows us to derive several semantic results. Other than the completeness and soundness of NbE, another key result is the consistency of McTT:

Corollary 4.16 (Consistency). *There is no such* M *and* i *satisfying* $\Gamma \vdash M : \Pi(x : \mathsf{Type@}i).x$.

McTT also contains other interesting results such as the idempotency of NbE and injectivity of Π with regard to our equivalence. One can find them in our semantic consequence module.

5 Algorithmic Component

In the previous sections, we have introduced the NbE algorithm that computes $\beta\eta$ normal forms of well-formed expressions, and proved its completeness and soundness. Based on the NbE algorithm, we can provide a set of algorithmic rules for type-checking. For simplicity, our type-checking algorithm is essentially uni-directional; this is possible because we made sure that types can always be inferred from given expressions. During type-checking, we sometimes need to check whether an expression of an inferred type can also have a desired type. For example, in Sec. 2.1, we give an example program where we infer the type $\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@0}$, but the desired type is $\Pi(y{:}\mathsf{Type@0}).\mathsf{Type@1}$. Therefore, we also need an algorithm for subtyping. As a result, we are motivated to define the following four algorithmic judgements:

$\boxed{\vdash^{\mathsf{Nf}}_A W \subseteq W'}$ Algorithmic subtyping between normal types

$$\frac{W \text{ is not Type or } \Pi}{\vdash^{\mathsf{Nf}}_A W \subseteq W} \qquad \frac{i \leq j}{\vdash^{\mathsf{Nf}}_A \mathsf{Type}@i \subseteq \mathsf{Type}@j} \qquad \frac{\vdash^{\mathsf{Nf}}_A W_2 \subseteq W_2'}{\vdash^{\mathsf{Nf}}_A \Pi(x : W_1).W_2 \subseteq \Pi(x : W_1).W_2'}$$

$\boxed{\Gamma \vdash_A A \subseteq B}$ Algorithmic subtyping between well-formed types

$$\frac{\vdash^{\mathsf{Nf}}_A \mathsf{nbe}_\Gamma(A) \subseteq \mathsf{nbe}_\Gamma(B)}{\Gamma \vdash_A A \subseteq B}$$

$\boxed{\Gamma \vdash_A M \Leftarrow A}$ Term M is checked against type A

$$\frac{\Gamma \vdash_A M \Rightarrow W \qquad \Gamma \vdash_A W \subseteq A}{\Gamma \vdash_A M \Leftarrow A}$$

$\boxed{\Gamma \vdash_A M \Rightarrow W}$ W is an inferred normal type of term M

$$\frac{x : A \in \Gamma}{\Gamma \vdash_A x \Rightarrow \mathsf{nbe}_\Gamma(A)} \qquad \frac{\Gamma \vdash_A A \Rightarrow \mathsf{Type}@i \qquad \Gamma, x : A \vdash_A B \Rightarrow \mathsf{Type}@j}{\Gamma \vdash_A \Pi(x : A).B \Rightarrow \mathsf{Type}@\max(i, j)}$$

$$\frac{\Gamma \vdash_A A \Rightarrow \mathsf{Type}@i \qquad \Gamma, x : A \vdash_A M \Rightarrow W}{\Gamma \vdash_A \lambda(x : A).M \Rightarrow \Pi(x : \mathsf{nbe}_\Gamma(A)).W} \qquad \frac{\Gamma \vdash_A M \Rightarrow \Pi(x : W).W' \qquad \Gamma \vdash_A N \Leftarrow W}{\Gamma \vdash_A M\ N \Rightarrow \mathsf{nbe}_\Gamma(W'[\mathsf{id}, N/x])}$$

Fig. 11. Algorithmic judgements

- $\boxed{\Gamma \vdash_A M \Leftarrow A}$: Expression M checks against type A.
- $\boxed{\Gamma \vdash_A M \Rightarrow W}$: A *normal* type W is inferred from expression M.
- $\boxed{\vdash^{\mathsf{Nf}}_A W \subseteq W'}$: Normal type W is an algorithmic subtype of normal type W'.
- $\boxed{\Gamma \vdash_A A \subseteq B}$: Type A is an algorithmic subtype of type B.

Selected rules are shown in Fig. 11, and rest of the rules are available on Appendix A (also in its mechanization). The judgement $\Gamma \vdash_A M \Leftarrow A$ implements type-checking by checking M against a desired type A and is the entry point of the overall type-checking procedure. This judgement first infers a normal type W from M, and then checks if W is a subtype of A.

The main workhorse of the type-checking algorithm is the inference judgement $\Gamma \vdash_A M \Rightarrow W$. Type inference proceeds in a standard manner by recursively walking down the AST so that for any expression, at most one inference rule applies. Note that the judgement always infers a *normal* type, which ensures a unique return for a given input expression. The NbE algorithm is properly invoked to ensure that this invariant is maintained. In certain rules, the checking direction might also be used. For example, in the case of the function application $M\ N$, we first make sure that M has a function type $\Pi(x : W).W'$. To ensure that the function application is well-formed, $\Gamma \vdash_A N \Leftarrow W'$ is invoked to check whether N has type W'. If that is the case, we can safely say that the overall type of $M\ N$ is $W'[\mathsf{id}, N/x]$, and we normalize this type before returning it.

Another critical piece that type-checking relies on is the algorithmic subtyping. The main entry is $\Gamma \vdash_A A \subseteq B$, which has only one case. It simply invokes NbE to normalize the input types A and B, and passes the results in normal forms to the core subtyping procedure $\vdash^{\mathsf{Nf}}_A W \subseteq W'$. Note that the context Γ is only used for normalization; the core subtyping comparison algorithm does not rely on the context at all. Fig. 11 shows the full set of three rules for $\vdash^{\mathsf{Nf}}_A W \subseteq W'$. If W and W' are

neither universes nor Π types, then they must be equal. For two universes, the first universe level must be smaller than or equal to the other. Given two Π types, the input types must be the same, while the output type of the first Π must be a subtype of the second output type.

We prove that the algorithmic definitions are sound and complete w.r.t. the declarative definitions. We proceed by first reasoning about the algorithmic subtyping. The easier direction is the soundness of the two algorithmic subtyping judgements:

Theorem 5.1 (Soundness of algorithmic subtyping for normal types). *If* $\vdash_A^{Nf} W \subseteq W'$ *and for any* Γ *and* i*, such that* $\Gamma \vdash W : \mathsf{Type}@i$ *and* $\Gamma \vdash W' : \mathsf{Type}@i$*, then* $\Gamma \vdash W \subseteq W'$.

In other words, algorithmic subtyping for normal types always holds regardless of the typing context. Then, the soundness of algorithmic subtyping for general types is proved using the soundness of NbE:

Theorem 5.2 (Soundness of algorithmic subtyping). *If* $\Gamma \vdash_A A \subseteq B$, $\Gamma \vdash A : \mathsf{Type}@i$ *and* $\Gamma \vdash B : \mathsf{Type}@i$*, then* $\Gamma \vdash A \subseteq B$.

The completeness direction is a bit more challenging because we must show that algorithmic subtyping is transitive and works for the Π case.

Lemma 5.3 (Transitivity of algorithmic subtyping). *If* $\Gamma \vdash_A A_0 \subseteq A_1$ *and* $\Gamma \vdash_A A_1 \subseteq A_2$*, then* $\Gamma \vdash_A A_0 \subseteq A_2$.

Theorem 5.4 (Completeness of algorithmic subtyping). *If* $\Gamma \vdash A \subseteq B$*, then* $\Gamma \vdash_A A \subseteq B$.

Proof. Induction on $\Gamma \vdash A \subseteq B$. In the Π case, since we support only covariant subtyping, we know the input types are equivalent. This is checked by the equality of their normal forms. Moreover, the equivalence between input types also ensures that the contexts in which the output types are normalized are also equivalent. Therefore, the normal forms of output types can also be compared by algorithmic subtyping. At this point, the induction hypothesis applies and discharges this case. □

Next, we show that algorithmic typing is also sound and complete w.r.t. the declarative typing. The easier direction is again soundness, which is proved by mutual induction on type-checking and type inference:

Theorem 5.5 (Soundness of algorithmic typing).

- *If* $\Gamma \vdash_A M \Leftarrow A$, $\vdash \Gamma$ *and* $\Gamma \vdash A : \mathsf{Type}@i$*, then* $\Gamma \vdash M : A$.
- *If* $\Gamma \vdash_A M \Rightarrow W$ *and* $\vdash \Gamma$*, then* $\Gamma \vdash M : W$.

Completeness is a collective consequence of all previous results:

Theorem 5.6 (Completeness of algorithmic typing). *If* M *contains no closure and* $\Gamma \vdash M : A$*, then* $\Gamma \vdash_A M \Leftarrow A$ *and there exists* W *such that* $\Gamma \vdash_A M \Rightarrow W$ *and* $\Gamma \vdash W \subseteq A$.

6 Implementation and Extraction

So far, we have provided complete and sound NbE algorithm, subtyping algorithm and type-checking algorithm. However, these algorithms cannot be directly implemented as functions because Rocq requires functions to be *total*; while NbE, the base algorithm of all, is partial in general and only terminates when inputs are well-formed. For this reason, in Sec. 3.3 and 5, we define evaluation, readback, and algorithmic typing as functional relations to prove properties about them.

However, one immediate problem is that these functional relations cannot be extracted to executable functions while our purpose is to provide a verified *implementation*. Moreover, an

executable version of these algorithms should not carry proof information during runtime for *efficiency*. Last, the algorithms should be extracted into *human-readable* and comparable to what a skilled human programmer would write. This means that the code can be verified by a skilled human programmer independently. This not only increases our confidence in the final result but also allows developers to experiment with new features before fully verifying them.

To summarize, the main challenge in providing a verified proof checker is to ensure that the extracted code: 1) satisfies the properties proven in the theoretical component; 2) does not carry any extraneous information during runtime; 3) is human-readable. Achieving these three goals at the same time is a nontrivial problem. For example, Hu et al. [2023]'s extraction from Agda to Haskell carries proof information. As a consequence, performance significantly declines and the extracted code base is messy, hard to work with, and difficult to read for a human programmer.

One potential approach to resolve this issue in Rocq is to use the `coq-partialfun` library [Winterhalter 2023], which uses free monads to represent partial functions, so that the latter can be executed within Rocq. This library is used by Adjedj et al. [2024] to implement a type-checking algorithm in Rocq. However, extraction from this library carries some line noise due to the monadic style and is not very human-readable.

In McTT, we use a variation of Bove [2009]'s method, where for each function, we define a call graph to restrict its domain. This call graph also serves as the termination measure. We define this call graph as an inductive relation in `Prop`. Since Rocq's extraction mechanism omits all instances of `Prop`, we automatically obtain a code base that is free of proofs and is as readable as a human-written source program. For example, we define the call graph of `eval_exp` (i.e. the name of the functional relation that formalizes $[\![M]\!](\rho) = m$ in McTT) as the following inductive type `eval_exp_order`. We only show the application case below:

```
Inductive eval_exp_order : exp -> env -> Prop :=
(* ... *)
| eeo_app : forall M N p,
  eval_exp_order M p ->
  eval_exp_order N p ->
  (forall m n, eval_exp M p m -> eval_exp N p n -> eval_app_order m n) ->
  eval_exp_order (a_app M N) p
```

This case says that the call graph for the function application `a_app M N` case includes the sub-graphs for `M` and `N`, and also another sub-graph for the result of applying the evaluation of `M` to that of `N`.

We now briefly discuss the evaluation function `eval_exp_impl`. As a naming convention, we use the post-fix `impl` to refer to the implementation of the corresponding functional relation `eval_exp`. In addition to the expressions `m:exp` and the environment `p:env`, it also takes in the call graph defined by the inductive type `eval_exp_order m p`. Its return states that there exists a normal form `d` s.t. `eval_exp m p d`. In Rocq, we write the Σ type `{ d | eval_exp m p d }` for it. Mimicking the definition of `eval_exp_order` (i.e. the definition of the call graph), we define `eval_exp_impl` as follows: we first recursively evaluate `M` and `N` using `eval_exp_impl`. Then, we apply `eval_app_impl` (which corresponds to $\_ \cdot \_$) to the results. The function is implemented using the Equations plugin [Sozeau and Mangin 2019] to discharge proof obligations more easily.

```
Equations eval_exp_impl m p (H : eval_exp_order m p) : { d | eval_exp m p d } :=
(* ... *)
| a_app M N , p, H =>
    let (m , Hm) := eval_exp_impl M p _ in
    let (n , Hn) := eval_exp_impl N p _ in
    let (a, Ha) := eval_app_impl m n _ in
    exist _ a _
```

Rocq's extraction removes the call graph and the witness that the resulting normal form `d` is identical to the one the declarative algorithm returns. In other words, we remove `eval_exp_order m p` from the input and `eval_exp m p d` from the output. As a result, the extracted code is without any redundant proof witness and is what a skilled human programmer would have written directly in OCaml.

```
let rec eval_exp_impl m p : domain =
(* ... *)
| A_app (e, e0) -> eval_app_impl (eval_exp_impl e p) (eval_exp_impl e0 p)
```

Now, how can we obtain an instance of `eval_exp_order` when we use this function? Our solution is to prove that `eval_exp_order` is sound: If there exists a `d`, s.t. `eval_exp M p d` then we have `eval_exp_order M p`. Our completeness theorem for NbE guarantees that there is a proof of `exists d, eval_exp M p d` for a well-typed term `M`. In fact, we can easily get this order when we implement a type-checker by type-checking `M` first.

We have addressed two goals out of three in the implementation: the extracted code carries no extra information and is human-readable. The last problem is to prove that the implementation is indeed sound and complete w.r.t. the corresponding functional relation. In fact, soundness is an immediate consequence of the return type of the function: `eval_exp_impl` returns `{ d | eval_exp m p d }`, i.e. a domain value `d` that satisfies `eval_exp m p d`. Completeness, on the other hand, says that every instance of a functional relation finds an execution in the implementation. For evaluation, the lemma states that if `eval_exp m p d`, then `eval_exp_impl m p` terminates and returns the same `d`. This property is easy to conclude, because `eval_exp m p d` is deterministic and `eval_exp_order` is sound.

We apply this procedure for all algorithms (readback, NbE, subtyping and type-checking) to obtain a fully verified pipeline after extraction satisfying all three criteria.

7 Related Work

7.1 Mechanization of Type Theories

The mechanization of type theories dates back to Barras and Werner [1997]'s effort on mechanizing the Calculus of Constructions. More recently, Abel et al. [2018] mechanize a weak normalization proof for MLTT with two universes using Tait's reducibility candidates [Tait 1967] in Agda. This mechanization is the basis for many subsequent developments [Abel et al. 2023; Liu et al. 2025; Pujet and Tabareau 2022, 2023]. Recently, Adjedj et al. [2024] also mechanize this proof in Rocq concentrating on a fixed number of universes. However, Tait-style normalization proofs often require significant technical set-up and time investment when we formalize them in a proof assistant. Furthermore, this style of proof does not directly give us a complete $\beta\eta$-normalization algorithm.

Our McTT infrastructure uses normalization-by-evaluation (NbE), which often leads to more compact proofs. Normalization by evaluation is a technique to piggyback the normalization problem of the target system onto some computational domain. NbE was originally developed by Martin-Löf [1975] to prove the β normalization of a version of his type theories, and independently by Berger and Schwichtenberg [1991] for their proof checker MinLog. There are different possible choices for the computational domain. One popular choice is based on category theory [Altenkirch et al. 1995]. Such models are often organized as presheaf categories [Hu and Pientka 2022; Valliappan et al. 2022], which map the category of weakenings to the set of well-formed terms. Altenkirch and Kaposi [2016a,b, 2017] mechanize an NbE proof for a dependent type theory based on category with families [Dybjer 1995], a presheaf framework for modelling dependent types.

NbE algorithms are usually easier to understand, extend, and easier to implement if we choose an untyped domain model over a presheaf model [Abel 2013]. For example, Abel et al. [2017] and Gratzer et al. [2019] both extend this style of NbE proof to sized types and idempotent modal types

respectively. Most recently, Hu et al. [2023] mechanize an NbE algorithm for MLTT extended with a $\Box$ modality, based on Abel [2013]'s untyped domain model in just 11K lines of Agda code. This demonstrates the compactness and elegance of mechanizing NbE for dependent type theories, in particular compared to mechanizations of other styles of normalization proofs.

Mechanizing normalization proofs for type theory within Rocq poses its own challenge: in particular, we cannot rely on induction-recursion in the definition of our semantic model which is common in on-paper proofs as well as in Agda mechanizations. Existing mechanizations of type theory such as by Adjedj et al. [2024] or Wieczorek and Biernacki [2018] cap the number of universes. However, removing this restriction is not straightforward in Rocq. Our McTT infrastructure leverages impredicativity in Rocq together with our subtyping rule to provide a general principled way of developing proofs that usually require induction-recursion. This allows us to mechanize a larger fragment of MLTT than previously in Rocq. Choosing Rocq as our proof assistant also has a distinct advantage: we can leverage Rocq's excellent support for code extraction to derive an OCaml implementation of the verified normalization algorithms without extraneous noises from proof evidences.

7.2 Verified Proof Checking Kernels

In addition to the mechanization of CIC, Barras and Werner [1997] also made the first notable attempt to verify the kernel of a significant fragment of the Calculus of Inductive Construction and to extract a verified type-checker from the verification. More recently, an alternative approach has been pursued in the MetaRocq project to verify the formal semantics of Rocq's type theory within Rocq [Sozeau et al. 2020a,b, 2019]. In this line of work, the representation of Rocq terms in the implementation of Rocq is reflected into Rocq itself. Many properties about Rocq can then be proven and mechanized within Rocq about these representations. Since MetaRocq aims to capture the whole feature sets of Rocq, we cannot hope to prove the consistency of Rocq itself due to Gödel's incompleteness theorem. Hence, it is necessary to postulate normalization. The goal of McTT is to actually internalize the normalization proof for a type theory within Rocq. In this approach we will ultimately hit the normalization barrier, but we can still mechanize a significant, but strict subset of Rocq.

Compared to previous work, in McTT, we verify the meta-theory of MLTT hand-in-hand with the algorithmic implementation of a type checker and a normalizer for MLTT. Using Rocq's code extraction, we are able to synthesize an OCaml implementation that is readable and compact. This gives us a verified type-checker for a core fragment of MLTT.

8 Conclusion

We have described the McTT infrastructure to build a verified implementation of a normalizer and type-checker for core MLTT. Every step in the McTT pipeline is verified except for the lexer and pretty-printer. As a result, we have a fully verified kernel for core MLTT with the cumulative universe hierarchy.

In building McTT, we pushed our understanding on how to mechanize type theory in Rocq showing how to leverage impredicativity to deal with the more common induction-recursion definitions. Another important lesson we can draw from our work is that theory and verifiable algorithmic implementations of that theory should be developed hand in hand in a tight loop. In our case, this leads us to include universe subtyping.

Our McTT infrastructure also provides valuable lessons on how to arrive at a verified normalization algorithm within Rocq that allows us to extract readable OCaml code without extraneous noises due to proof witnesses. In particular, we identified two important steps: 1) Encoding partial

functions for evaluation and readback using functional relations and 2) Defining a call graph of each function to restrict the domain of the function following Bove [2009].

We regard McTT as a test-bed for two further directions: First, the theoretical component is a platform for formally verifying standard features like equality types and inductive types, to narrow the gap between the meta-theory and the implementation. Extending our meta-theoretic proofs follows the same principle that we have set up. While this is somewhat tedious, we believe that it is very feasible given the modularity of the McTT infrastructure. This would allow us to extract a verified type-checker to OCaml which can then be used in practice to certify significant parts of existing mechanizations.

Second, McTT can be used to study novel extensions to standard MLTT such as modalities or proof irrelevance. We believe that our theoretical component is sufficiently modular that others can build on it. Furthermore, we believe that leveraging impredicativity for modelling induction-recursion is a technique that is very relevant when mechanizing advanced type theories. More importantly, our McTT infrastructure not only allows us to study the meta-theory of these advanced type theories, but also provides a convenient way to extract a verified type-checker to OCaml, so that we can experiment with these advanced features and explore their potential impact in practice.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions and feedbacks. This work was funded in part by a Natural Sciences and Engineering Research Council of Canada (discovery grant 206263) and Fonds de recherche du Quebec - Nature et technologies (grant 253521). Junyoung Jang was funded by Fonds de recherche du Quebec - Nature et technologies (grant 333531). Jason Z. S. Hu was funded partly by Postgraduate Scholarship - Doctoral from the Natural Sciences and Engineering Research Council of Canada and partly by Doctoral (B2X) Research Scholarship from Fonds de recherche du Québec - Nature et technologies during his Ph.D. study.

Data-Availability Statement

This paper is accompanied by a software artifact [Jang et al. 2025] that embodies the key contribution of this paper, a verified kernel of MLTT. The installation instruction and browsable documentation are available in https://beluga-lang.github.io/McTT/icfp25/.

References

Andreas Abel. 2013. *Normalization by Evaluation: Dependent Types and Impredicativity.* Habilitation Thesis. Ludwig-Maximilians-Universität München, Munich, Germany. https://www.cse.chalmers.se/~abela/habil.pdf

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP (2023), 920–954. doi:10.1145/3607862

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL (2018), 23:1–23:29. doi:10.1145/3158111

Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by Evaluation for Sized Dependent Types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 33:1–33:30. doi:10.1145/3110277

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 230–245. doi:10.1145/3636501.3636951

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical Reconstruction of a Reduction Free Normalization Proof. In *Proceedings of the 6th International Conference on Category Theory and Computer Science, CTCS 1995, Cambridge, UK, August 7-11, 1995 (Lecture Notes in Computer Science, Vol. 953)*, David H. Pitt, David E. Rydeheard, and Peter T. Johnstone (Eds.). Springer, 182–199. doi:10.1007/3-540-60164-3_27

Thorsten Altenkirch and Ambrus Kaposi. 2016a. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, Porto, Portugal, June 22-26, 2016 (LIPIcs,*

Vol. 52), Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:16. doi:10.4230/LIPICS.FSCD.2016.6

Thorsten Altenkirch and Ambrus Kaposi. 2016b. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, Florida, USA, January 20-22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 18–29. doi:10.1145/2837614.2837638

Thorsten Altenkirch and Ambrus Kaposi. 2017. Normalisation by Evaluation for Type Theory, in Type Theory. *Log. Methods Comput. Sci.* 13, 4 (2017). doi:10.23638/LMCS-13(4:1)2017

Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. 2016. Encoding Proofs in Dedukti: the case of Coq proofs. In *Proceedings Hammers for Type Theories*. 1–6. https://inria.hal.science/hal-01330980

Jeremy Avigad and John Harrison. 2014. Formally verified mathematics. *Commun. ACM* 57, 4 (2014), 66–75.

Bruno Barras and Benjamin Werner. 1997. Coq in Coq. https://www.lix.polytechnique.fr/Labo/Bruno.Barras/publi/coqincoq.pdf Unpublished manuscript.

Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed Lambda-calculus. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science, LICS 1991, Amsterdam, the Netherlands, July 15-18, 1991*. IEEE Computer Society, 203–211. doi:10.1109/LICS.1991.151645

Mathieu Boespflug and Guillaume Burel. 2012. CoqInE: Translating the Calculus of Inductive Constructions into the λΠ-calculus Modulo. In *Workshop on Proof eXchange for Theorem Proving (PxTP) (CEUR Workshop Proceedings, Vol. 878)*. CEUR-WS.org, 44–50.

Ana Bove. 2009. Another Look at Function Domains. *Electronic Notes in Theoretical Computer Science* 249 (2009), 61–74. doi:10.1016/j.entcs.2009.07.084 Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708. doi:10.1017/S0960129505004822

Kevin Buzzard, Johan Commelin, and Patrick Massot. 2020. Formalising perfectoid spaces. 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, 299–312. doi:10.1145/3372885.3373830

James Chapman. 2008. Type Theory Should Eat Itself. In *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice, LFMTP@LICS 2008, Pittsburgh, Pennsylvania, USA, June 23, 2008 (Electronic Notes in Theoretical Computer Science, Vol. 228)*, Andreas Abel and Christian Urban (Eds.). Elsevier, 21–36. doi:10.1016/J.ENTCS.2008.12.114

Thierry Coquand. 2018. Canonicity and normalisation for Dependent Type Theory. *CoRR* abs/1810.09367 (2018). arXiv:1810.09367 http://arxiv.org/abs/1810.09367

Thierry Coquand and Peter Dybjer. 1997. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science* 7, 1 (1997), 75–94. doi:10.1017/S0960129596002150

Thierry Coquand and Christine Paulin. 1988. Inductively Defined Types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings (Lecture Notes in Computer Science, Vol. 417)*, Per Martin-Löf and Grigori Mints (Eds.). Springer, 50–66. doi:10.1007/3-540-52335-9_47

Peter Dybjer. 1995. Internal Type Theory. In *International Workshop on Types for Proofs and Programs, TYPES 1995, Torino, Italy, June 5-8, 1995, Selected Papers (Lecture Notes in Computer Science, Vol. 1158)*, Stefano Berardi and Mario Coppo (Eds.). Springer, 120–134. doi:10.1007/3-540-61780-9_66

Peter Dybjer and Anton Setzer. 2003. Induction-recursion and Initial Algebras. *Ann. Pure Appl. Log.* 124, 1-3 (2003), 1–47. doi:10.1016/S0168-0072(02)00096-9

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing A Modal Dependent Type Theory. *Proc. ACM Program. Lang.* 3, ICFP (2019), 107:1–107:29. doi:10.1145/3341711

Robert Harper and Frank Pfenning. 2005. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log.* 6, 1 (2005), 61–101. doi:10.1145/1042038.1042041

Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. 2023. Normalization by Evaluation for Modal Dependent Type Theory. *J. Funct. Program.* 33 (2023). doi:10.1017/S0956796823000060

Jason Z. S. Hu and Brigitte Pientka. 2022. A Categorical Normalization Proof for the Modal Lambda-Calculus. In *Proceedings of the 38th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2022, Cornell University, Ithaca, New York, USA, with a satellite event at IRIF, Denis Diderot University, Paris, France, and online, July 11-13, 2022 (EPTICS, Vol. 1)*, Justin Hsu and Christine Tasson (Eds.). EpiSciences. doi:10.46298/ENTICS.10360

Junyoung Jang, Antoine Gaulin, Jason Z. S. Hu, and Brigitte Pientka. 2025. *McTT: A Verified Kernel for a Proof Assistant*. doi:10.5281/zenodo.15712175

Dominique Larchey-Wendling and Jean-François Monin. 2018. Simulating Induction-Recursion for Partial Algorithms. In *24th International Conference on Types for Proofs and Programs,TYPES 2018*. Braga, Portugal. https://hal.science/hal-02333374

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Xavier Leroy. 2009b. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446.

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

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Studies in proof theory, Vol. 1. Bibliopolis.

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium 1973*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73–118. doi:10.1016/S0049-237X(08)71945-1

Christine Paulin-Mohring. 1993. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, the Netherlands, March 16-18, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 664)*, Marc Bezem and Jan Friso Groote (Eds.). Springer, 328–345. doi:10.1007/BFB0037116

Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now for Good. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. doi:10.1145/3498693

Loïc Pujet and Nicolas Tabareau. 2023. Impredicative Observational Equality. *Proc. ACM Program. Lang.* 7, POPL (2023), 2171–2196. doi:10.1145/3571739

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.* 11, 4 (1998), 363–397. doi:10.1023/A:1010027404223

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020a. The MetaCoq Project. *J. Autom. Reason.* 64, 5 (2020), 947–999. https://doi.org/10.1007/s10817-019-09540-0

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020b. The MetaCoq Project. *Journal of Automated Reasoning* (Feb. 2020). doi:10.1007/s10817-019-09540-0

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (Dec. 2019), 28 pages. doi:10.1145/3371076

Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP (2019), 86:1–86:29. doi:10.1145/3341690

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. doi:10.2307/2271658

The Mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, Louisiana, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. doi:10.1145/3372885.3373824

Nachiappan Valliappan, Fabian Ruch, and Carlos Tomé Cortiñas. 2022. Normalization for Fitch-style Modal Calculi. *Proc. ACM Program. Lang.* 6, ICFP (2022), 772–798. https://doi.org/10.1145/3547649

Pawel Wieczorek and Dariusz Biernacki. 2018. A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, California, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 266–279. doi:10.1145/3167091

Théo Winterhalter. 2023. Composable partial functions in Coq, totally for free. In *29th International Conference on Types for Proofs and Programs TYPES 2023 — Abstracts*. 208.

A Definition of Martin-Löf type theory

A.1 Syntactic definition

A.1.1 Syntax. We recall the syntax of McTT for convenience:

Variables			x, y, r		
De Bruijn indices	$\mathbb{N}$	$\ni$	d		
Universe levels	$\mathbb{N}$	$\ni$	i, j		
Contexts	Ctx	$\ni$	Γ, Δ	$::=$	$\cdot \mid \Gamma, x : A$
Expressions	Exp	$\ni$	A, B, C, M, N	$::=$	$x_d \mid \mathsf{Nat} \mid \mathsf{Type@}i \mid \Pi(x{:}A).B$
				$\mid$	$\mathsf{zero} \mid \mathsf{succ}\ M \mid \mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ N$
				$\mid$	$\lambda(x{:}A).M \mid M\ N \mid M[\sigma]$
Substitutions	Subst	$\ni$	σ, δ	$::=$	$\mathsf{id} \mid \mathsf{wk} \mid \sigma \circ \delta \mid \sigma, M/x_0$

A.1.2 Judgments. We give a more exhaustive definition of the declarative judgements of McTT. Congruence rules are omitted from the equivalence judgments.

$\boxed{\vdash \Gamma}$ Γ is a well-formed context.

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathsf{Type@}i}{\vdash \Gamma, x{:}A}$$

$\boxed{\Gamma \vdash t : T}$ Term t has type T in Γ.

$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash A : \mathsf{Type@}i}{\Gamma \vdash A : \mathsf{Type@}1 + i} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat} : \mathsf{Type@}0} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\ M : \mathsf{Nat}} \qquad \frac{\begin{array}{c}\Gamma, x : \mathsf{Nat} \vdash A : \mathsf{Type@}i \qquad \Gamma \vdash M_{\mathsf{zero}} : A[\mathsf{id}, \mathsf{zero}/x] \\ \Gamma, y : \mathsf{Nat}, r : A \vdash M_{\mathsf{succ}} : A[\mathsf{wk} \circ \mathsf{wk}, \mathsf{succ}\ y/x] \qquad \Gamma \vdash N : \mathsf{Nat}\end{array}}{\Gamma \vdash \mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ N : A[\mathsf{id}, N/x]}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i}{\Gamma \vdash \Pi(x : A).B : \mathsf{Type@}i} \qquad \frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda(x : A).M : \Pi(x : A).B}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i \qquad \Gamma \vdash M : \Pi(x : A).B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[\mathsf{id}, N/x]} \qquad \frac{\Delta \vdash M : A \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash M[\sigma] : A[\sigma]}$$

$$\frac{\Gamma \vdash A' : \mathsf{Type@}i \qquad \Gamma \vdash M : A \qquad \Gamma \vdash A \subseteq A'}{\Gamma \vdash M : A'}$$

$\boxed{\Gamma \vdash \sigma : \Delta}$ σ is a well-formed substitution from Δ to Γ.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{id} : \Gamma} \qquad \frac{\vdash \Gamma, x : A}{\Gamma, x : A \vdash \mathsf{wk} : \Gamma} \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash A : \mathsf{Type@}i \qquad \Gamma \vdash M : A[\sigma]}{\Gamma \vdash \sigma, M/x_0 : \Delta, x : A}$$

$$\frac{\Gamma' \vdash \sigma : \Gamma'' \qquad \Gamma \vdash \delta : \Gamma'}{\Gamma \vdash \sigma \circ \delta : \Gamma''} \qquad \frac{\vdash \Delta' \qquad \Gamma \vdash \sigma : \Delta \qquad \vdash \Delta \subseteq \Delta'}{\Gamma \vdash \sigma : \Delta'}$$

$\boxed{\Gamma \vdash M \approx N : A}$ M and N of type A are equivalent.

$$\frac{\Gamma, x : \mathsf{Nat} \vdash A : \mathsf{Type@}i \qquad \Gamma \vdash M_{\mathsf{zero}} : A[\mathsf{id}, \mathsf{zero}/x_0] \qquad \Gamma, y : \mathsf{Nat}, r : A \vdash M_{\mathsf{succ}} : A[\mathsf{wk} \circ \mathsf{wk}, \mathsf{succ}\ y/x_0]}{\Gamma \vdash \mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ \mathsf{zero} \approx M_{\mathsf{zero}} : A[\mathsf{id}, \mathsf{zero}/x_0]}$$

$$\frac{\Gamma, x : \mathsf{Nat} \vdash A : \mathsf{Type@}i \qquad \Gamma \vdash M_{\mathsf{zero}} : A[\mathsf{id}, \mathsf{zero}/x_0] \qquad \Gamma, y : \mathsf{Nat}, r : A \vdash M_{\mathsf{succ}} : A[\mathsf{wk} \circ \mathsf{wk}, \mathsf{succ}\ y/x_0] \qquad \Gamma \vdash N : \mathsf{Nat} \qquad M_r = \mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ N}{\Gamma \vdash \mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ (\mathsf{succ}\ N) \approx M_{\mathsf{succ}}[\mathsf{id}, N/y_1, M_r/r_0] : A[\mathsf{id}, \mathsf{succ}\ N/x_0]}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i \qquad \Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda(x : A).M)\ N \approx M[\mathsf{id}, N/x_0] : B[\mathsf{id}, N/x_0]}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i \qquad \Gamma \vdash M : \Pi(x : A).B}{\Gamma \vdash M \approx \lambda(x : A).(M[\mathsf{wk}])\ x : \Pi(x : A).B}$$

$$\frac{\Gamma \vdash A' : \mathsf{Type@}i \qquad \Gamma \vdash M \approx N : A \qquad \Gamma \vdash A \subseteq A'}{\Gamma \vdash M \approx N : A'}$$

$\boxed{\Gamma \vdash \sigma \approx \delta : \Delta}$ σ and δ are equivalent substitutions from Δ to Γ.

$$\frac{\Gamma' \vdash \sigma : \Gamma'' \qquad \Gamma'' \vdash A : \mathsf{Type@}i \qquad \Gamma' \vdash M : A[\sigma] \qquad \Gamma \vdash \delta : \Gamma'}{\Gamma \vdash (\sigma, M/x_0) \circ \delta \approx (\sigma \circ \delta), M[\delta]/x_0 : \Gamma'', x : A} \qquad \frac{\vdash \Delta' \qquad \Gamma \vdash \sigma \approx \sigma' : \Delta \qquad \vdash \Delta \subseteq \Delta'}{\Gamma \vdash \sigma \approx \sigma' : \Delta'}$$

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Delta \vdash A : \mathsf{Type@}i \qquad \Gamma \vdash M : A[\sigma]}{\Gamma \vdash \mathsf{wk} \circ (\sigma, M/x_0) \approx \sigma : \Delta} \qquad \frac{\Gamma \vdash \sigma : \Gamma', x : A}{\Gamma \vdash \sigma \approx (\mathsf{wk} \circ \sigma), x[\sigma]/x_0 : \Gamma, x : A}$$

$\boxed{\Gamma \vdash A \subseteq B}$ A is a subtype of B.

$$\frac{\Gamma \vdash B : \mathsf{Type@}i \qquad \Gamma \vdash A \approx B : \mathsf{Type@}i}{\Gamma \vdash A \subseteq B} \qquad \frac{\Gamma \vdash A \subseteq A' \qquad \Gamma \vdash A' \subseteq A''}{\Gamma \vdash A \subseteq A''} \qquad \frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \mathsf{Type@}i \subseteq \mathsf{Type@}j}$$

$$\frac{\Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma \vdash A' : \mathsf{Type@}i \qquad \Gamma \vdash A \approx A' : \mathsf{Type@}i \qquad \Gamma, x : A \vdash B : \mathsf{Type@}i \qquad \Gamma, x : A' \vdash B' : \mathsf{Type@}i \qquad \Gamma, x : A' \vdash B \subseteq B'}{\Gamma \vdash \Pi(x : A).B \subseteq \Pi(x : A').B'}$$

$\boxed{\vdash \Gamma \subseteq \Delta}$ Γ is a sub-context of Δ.

$$\frac{}{\vdash \cdot \subseteq \cdot} \qquad \frac{\vdash \Gamma \subseteq \Gamma' \qquad \Gamma \vdash A : \mathsf{Type@}i \qquad \Gamma' \vdash A' : \mathsf{Type@}i \qquad \Gamma \vdash A \subseteq A'}{\vdash \Gamma, x : A \subseteq \Gamma', x : A'}$$

A.2 Semantic definition

A.2.1 Domain. We recall the definition of the domain for convenience:

$$
\begin{array}{llllll}
\text{De Bruijn levels} & \mathbb{N} & \ni & z & & \\
\text{Neutral domain values} & \mathsf{D}^{\mathsf{Ne}} & \ni & v & ::= & l_z \mid \mathsf{rec}(A, b, M_{\mathsf{succ}}, \rho, v) \mid v\ w \\
\text{Domain values} & \mathsf{D} & \ni & a, b, m, n & ::= & \uparrow^a (v) \mid \mathsf{N} \mid \mathsf{U}@i \mid \Pi(a, B, \rho) \\
 & & & & \mid & \mathsf{ze} \mid \mathsf{su}(a) \mid \Lambda(M, \rho) \\
\text{Normal domain values} & \mathsf{D}^{\mathsf{Nf}} & \ni & w & ::= & \downarrow^a (m) \\
\text{Evaluation environments} & \mathsf{Env} & \ni & \rho & &
\end{array}
$$

A.2.2 Evaluation.

$$
\begin{aligned}
[\![\_]\!](\_) &:: \mathsf{Exp} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{D} \\
[\![x_d]\!](\rho) &:= \rho(d) \\
[\![\Pi(x : A).B]\!](\rho) &:= \Pi([\![A]\!](\rho), B, \rho) \\
[\![\mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ N]\!](\rho) &:= \mathsf{rec} \cdot (A, [\![M_{\mathsf{zero}}]\!](\rho), M_{\mathsf{succ}}, \rho, [\![N]\!](\rho)) \\
[\![\lambda(x : A).M]\!](\rho) &:= \Lambda(M, \rho) \\
[\![M\ N]\!](\rho) &:= [\![M]\!](\rho) \cdot [\![N]\!](\rho) \\
[\![t[\sigma]]\!](\rho) &:= [\![t]\!]([\![\sigma]\!](\rho))
\end{aligned}
$$

$$
\begin{aligned}
[\![\_]\!](\_) &:: \mathsf{Subst} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{Env} \\
[\![\mathsf{id}]\!](\rho) &:= \rho \\
[\![\mathsf{wk}]\!](\rho) &:= \mathsf{drop}(\rho) \\
[\![\sigma, M/x_0]\!](\rho) &:= \mathtt{ext}([\![\sigma]\!](\rho), [\![M]\!](\rho)) \\
[\![\sigma \circ \delta]\!](\rho) &:= [\![\sigma]\!]([\![\delta]\!](\rho))
\end{aligned}
$$

$$
\begin{aligned}
\_ \cdot \_ &:: \mathsf{D} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{D} \\
(\Lambda(M, \rho)) \cdot a &:= [\![M]\!](\mathtt{ext}(\rho, a)) \\
(\uparrow^{\Pi(a,B,\rho)} (v)) \cdot m &:= \uparrow^{[\![B]\!](\mathtt{ext}(\rho,m))} (v\ \downarrow^a (m))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{rec} \cdot (\_, \_, \_, \_, \_) &:: \mathsf{Exp} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{Exp} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{D} \\
\mathsf{rec} \cdot (A, m_{\mathsf{zero}}, M_{\mathsf{succ}}, \rho, \mathsf{ze}) &:= m_{\mathsf{zero}} \\
\mathsf{rec} \cdot (A, m_{\mathsf{zero}}, M_{\mathsf{succ}}, \rho, \mathsf{su}(m)) &:= [\![M_{\mathsf{succ}}]\!](\mathtt{ext}(\rho, m, \mathsf{rec} \cdot (A, m_{\mathsf{zero}}, M_{\mathsf{succ}}, \rho, m))) \\
\mathsf{rec} \cdot (A, m_{\mathsf{zero}}, M_{\mathsf{succ}}, \rho, \uparrow^b (v)) &:= \uparrow^{[\![A]\!](\mathtt{ext}(\rho, \uparrow^b (v)))} (\mathsf{rec}(A, m_{\mathsf{zero}}, M_{\mathsf{succ}}, \rho, v))
\end{aligned}
$$

A.2.3 Readback.

$$
\begin{aligned}
\mathsf{R}^{\mathsf{Nf}} &:: \mathbb{N} \rightharpoonup \mathsf{D}^{\mathsf{Nf}} \rightharpoonup \mathsf{Nf} \\
\mathsf{R}^{\mathsf{Nf}}_z(\downarrow^a (\uparrow^b (v))) &:= \mathsf{R}^{\mathsf{Ne}}_z(v) && \text{(where } a = \uparrow^{a'} (c) \text{ or } \mathsf{N}) \\
\mathsf{R}^{\mathsf{Nf}}_z(\downarrow^{\mathsf{U}@i} (a)) &:= \mathsf{R}^{\mathsf{Ty}}_z(a) \\
\mathsf{R}^{\mathsf{Nf}}_z(\downarrow^{\Pi(a,B,\rho)} (m)) &:= \lambda(x : \mathsf{R}^{\mathsf{Ty}}_z(a)).\mathsf{R}^{\mathsf{Nf}}_{1+z}(\downarrow^{[\![B]\!](\mathtt{ext}(\rho, \uparrow^a (l_z)))} (m \cdot \uparrow^a (l_z))) \\
\mathsf{R}^{\mathsf{Ty}} &:: \mathbb{N} \rightharpoonup \mathsf{D} \rightharpoonup \mathsf{Nf} \\
\mathsf{R}^{\mathsf{Ty}}_z(\Pi(a, B, \rho)) &:= \Pi(x : \mathsf{R}^{\mathsf{Ty}}_z(a)).\mathsf{R}^{\mathsf{Ty}}_{1+z}([\![B]\!](\mathtt{ext}(\rho, \uparrow^a (l_z)))) \\
\mathsf{R}^{\mathsf{Ty}}_z(\uparrow^a (v)) &:= \mathsf{R}^{\mathsf{Ne}}_z(v) \\
\mathsf{R}^{\mathsf{Ne}} &:: \mathbb{N} \rightharpoonup \mathsf{D}^{\mathsf{Ne}} \rightharpoonup \mathsf{Ne} \\
\mathsf{R}^{\mathsf{Ne}}_{z'}(l_z) &:= x_{\max(z'-z-1,0)} \\
\mathsf{R}^{\mathsf{Ne}}_z(\mathsf{rec}(A, b, M_{\mathsf{succ}}, \rho, v)) &:= \mathsf{rec}_{x.W}\ \mathsf{R}^{\mathsf{Nf}}_z(\downarrow^{[\![A]\!](\mathtt{ext}(\rho, \mathsf{ze}))} (b))\ (x, y.W_{\mathsf{succ}})\ \mathsf{R}^{\mathsf{Ne}}_z(v) \\
& \qquad \text{(where } a := [\![A]\!](\mathtt{ext}(\rho, \uparrow^{\mathsf{N}} (l_z))),\ W := \mathsf{R}^{\mathsf{Ty}}_{1+z}(a), \text{ and)} \\
& (a' := [\![A]\!](\mathtt{ext}(\rho, \mathsf{su}(\uparrow^{\mathsf{N}} (l_z)))),\ m := [\![M_{\mathsf{succ}}]\!](\mathtt{ext}(\rho, \uparrow^{\mathsf{N}} (l_z), \uparrow^a (l_{1+z}))),\ W_{\mathsf{succ}} := \mathsf{R}^{\mathsf{Nf}}_{2+z}(\downarrow^{a'} (m))) \\
\mathsf{R}^{\mathsf{Ne}}_z(v\ w) &:= \mathsf{R}^{\mathsf{Ne}}_z(v)\ \mathsf{R}^{\mathsf{Nf}}_z(w)
\end{aligned}
$$

A.3 Algorithmic judgements

$\boxed{\vdash^{\mathsf{Nf}}_A W \subseteq W'}$ Algorithmic subtyping between normal types

$$\frac{W \text{ is not a universe or a } \Pi \text{ type}}{\vdash^{\mathsf{Nf}}_A W \subseteq W} \qquad \frac{i \leq j}{\vdash^{\mathsf{Nf}}_A \mathsf{Type@}i \subseteq \mathsf{Type@}j}$$

$$\frac{\vdash^{\mathsf{Nf}}_A W_2 \subseteq W_2'}{\vdash^{\mathsf{Nf}}_A \Pi(x : W_1).W_2 \subseteq \Pi(x : W_1).W_2'}$$

$\boxed{\Gamma \vdash_A A \subseteq B}$ Algorithmic subtyping between well-formed types

$$\frac{\vdash^{\mathsf{Nf}}_A \mathsf{nbe}_\Gamma(A) \subseteq \mathsf{nbe}_\Gamma(B)}{\Gamma \vdash_A A \subseteq B}$$

$\boxed{\Gamma \vdash_A M \Leftarrow A}$ Term M is checked against type A

$$\frac{\Gamma \vdash_A M \Rightarrow W \qquad \Gamma \vdash_A A \subseteq B}{\Gamma \vdash_A M \Leftarrow B}$$

$\boxed{\Gamma \vdash_A M \Rightarrow W}$ W is an inferred normal type of term M

$$\frac{x : A \in \Gamma}{\Gamma \vdash_A x \Rightarrow \mathsf{nbe}_\Gamma(A)} \qquad \frac{}{\Gamma \vdash_A \mathsf{Type@}i \Rightarrow \mathsf{Type@}(1+i)} \qquad \frac{}{\Gamma \vdash_A \mathsf{Nat} \Rightarrow \mathsf{Type@}0}$$

$$\frac{}{\Gamma \vdash_A \mathsf{zero} \Rightarrow \mathsf{Nat}} \qquad \frac{\Gamma \vdash_A M \Leftarrow \mathsf{Nat}}{\Gamma \vdash_A \mathsf{succ}\ M \Rightarrow \mathsf{Nat}}$$

$$\frac{\begin{array}{c}\Gamma, x : \mathsf{Nat} \vdash_A A \Rightarrow \mathsf{Type@}i \qquad \Gamma \vdash_A M_{\mathsf{zero}} \Leftarrow A[\mathsf{id}, \mathsf{zero}/x] \\ \Gamma, y : \mathsf{Nat}, r : A \vdash_A M_{\mathsf{succ}} \Leftarrow A[(\mathsf{wk} \circ \mathsf{wk}), \mathsf{succ}\ y/x] \qquad \Gamma \vdash_A N \Leftarrow \mathsf{Nat}\end{array}}{\Gamma \vdash_A \mathsf{rec}_{x.A}\ M_{\mathsf{zero}}\ (y, r.M_{\mathsf{succ}})\ N \Rightarrow \mathsf{nbe}_\Gamma(A[\mathsf{id}, N/x])}$$

$$\frac{\Gamma \vdash_A A \Rightarrow \mathsf{Type@}i \qquad \Gamma, x : A \vdash_A B \Rightarrow \mathsf{Type@}j}{\Gamma \vdash_A \Pi(x : A).B \Rightarrow \mathsf{Type@max}(i, j)} \qquad \frac{\Gamma \vdash_A A \Rightarrow \mathsf{Type@}i \qquad \Gamma, x : A \vdash_A M \Rightarrow W}{\Gamma \vdash_A \lambda(x : A).M \Rightarrow \Pi(x : \mathsf{nbe}_\Gamma(A)).W}$$

$$\frac{\Gamma \vdash_A M \Rightarrow \Pi(x : W).W' \qquad \Gamma \vdash_A N \Leftarrow W}{\Gamma \vdash_A M\ N \Rightarrow \mathsf{nbe}_\Gamma(W'[\mathsf{id}, N/x])}$$

Received 2025-02-27; accepted 2025-06-27