Implementing a Modal Dependent Type Theory

DANIEL GRATZER, Aarhus University, Denmark
JONATHAN STERLING, Carnegie Mellon University, United States
LARS BIRKEDAL, Aarhus University, Denmark

Modalities are everywhere in programming and mathematics! Despite this, however, there are still significant technical challenges in formulating a core dependent type theory with modalities. We present a dependent type theory MLTT$_{\text{🔒}}$ supporting the connectives of standard Martin-Löf Type Theory as well as an **S4**-style necessity operator. MLTT$_{\text{🔒}}$ supports a smooth interaction between modal and dependent types and provides a common basis for the use of modalities in programming and in synthetic mathematics. We design and prove the soundness and completeness of a type checking algorithm for MLTT$_{\text{🔒}}$, using a novel extension of normalization by evaluation. We have also implemented our algorithm in a prototype proof assistant for MLTT$_{\text{🔒}}$, demonstrating the ease of applying our techniques.

CCS Concepts: • **Theory of computation → Modal and temporal logics**; **Type theory**; **Proof theory**.

Additional Key Words and Phrases: Modal types, dependent types, normalization by evaluation, type-checking

ACM Reference Format:
Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a Modal Dependent Type Theory. *Proc. ACM Program. Lang.* 3, ICFP, Article 107 (August 2019), 29 pages. https://doi.org/10.1145/3341711

1 INTRODUCTION

Modalities have appeared as a powerful tool of abstraction in all corners of computer science and mathematics. In distributed computing, shareable values are naturally organized into a comonad [Epstein et al. 2011; Murphy 2008; Murphy et al. 2004]. In staged computation, each different stage for computation can be structured as another comonad [Davies and Pfenning 1999]. A wide variety of language-based security techniques are substantially based on modalities [Abadi et al. 1999]. In mathematics, modal type theory can be used to distill the situations of topological cohesion, differentiability, *etc.* into their algebraic essence [Schreiber 2013; Schreiber and Shulman 2014], promising new advances in the program initiated by Lawvere [1992]. Modal type theory also plays a critical role in the construction of classifying fibrations in cubical models of Homotopy Type Theory [Licata et al. 2018]. Similar ideas have been used with success in logical relations models of type systems for higher-order programming languages and in higher-order concurrent separation logics, where modalities, e.g., have been used to abstract guarded recursion [Birkedal et al. 2011; Bizjak and Birkedal 2018; Krebbers et al. 2017]. By abstracting the details of distributed computation, information flow, or topological spaces into a modal interface, we can program and prove domain-specific facts without recourse to low-level features of the situation.

Authors' addresses: Daniel Gratzer, Computer Science, Aarhus University, Aabogade 34, Aarhus N, 8200, Denmark, gratzer@cs.au.dk; Jonathan Sterling, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, 15213, United States, jmsterli@cs.cmu.edu; Lars Birkedal, Computer Science, Aarhus University, Aabogade 34, Aarhus N, 8200, Denmark, birkedal@cs.au.dk.

2475-1421/2019/8-ART107
https://doi.org/10.1145/3341711

Despite their ubiquity and obvious utility, modalities are notoriously difficult to incorporate into sophisticated type systems. While there have been considerable advances in the integration of modalities into simple type systems [Clouston 2018; Guatto 2018; Kavvos 2017; Licata et al. 2017; Pfenning and Davies 2000], it has remained a significant challenge to scale from simple type theory to "full-spectrum" dependent type theory in a way that preserves desirable syntactic properties (closure under substitution, canonicity, normalization, decidability of type checking, *etc.*).

Progress has been made in extending the ideas of Pfenning and Davies [2000] to richer type theories. For instance, in the context of logical frameworks, contextual modal type theory [Nanevski et al. 2008; Pientka et al. 2019] has been studied as a generalization of the necessity modality which allows for dependence on a specific set of local variables. Implementations of some dependent variants of Pfenning and Davies [2000] have also been explored in modern proof assistants [The Agda Development Team 2018]. Another recent advance is the development of Clocked Type Theory [Bahr et al. 2017], which extends the work of Clouston [2018] and applies it to guarded recursion.

We contribute $\mathsf{MLTT}_{\color{blue}\blacksquare}$, a core type theory which smoothly incorporates the comonadic necessity modality from **S4** into dependent type theory, while obtaining normalization and decidability of type checking for a (minimally) annotated version of $\mathsf{MLTT}_{\color{blue}\blacksquare}$. This type theory can be used simultaneously as a basis for next-generation programming languages and as a metalanguage for synthetic mathematics. We also show that $\mathsf{MLTT}_{\color{blue}\blacksquare}$ is immediately applicable by implementing our type checking algorithm in a prototype proof assistant.

Why is implementing dependent type theory difficult? In any typed language, it is necessary to decide when one type is equal to another. In a dependent type theory, however, a type may contain an arbitrary piece of code. Deciding the equality of types then entails deciding the equality of terms, a far more involved task. The equality of terms, moreover, should be as flexible as possible to ease the burden of proof, but if it is too flexible, type checking will become undecidable.

One robust approach to deciding equality in dependent type theory is *normalization by evaluation* (NbE) [Abel 2013; Berger and Schwichtenberg 1991; Martin-Löf 1975]. NbE is a type-directed procedure for reducing terms to a canonical representative of their equivalence class, scaling up to very sophisticated extensions of type theory [Abel 2009; Abel et al. 2009, 2017; Coquand 2018]. Coquand [1996] showed that NbE can also be used to implement an efficient bidirectional type checker for dependent type theory which avoids the use of substitution or De Bruijn shifting entirely during type checking.

We present a novel extension of NbE and semantic type checking to support modalities. This provides the template for an efficient implementation of a proof assistant for modal dependent type theory.

Why are modalities difficult to add to a type theory? In a type theory with robust syntactic properties, each connective arises from the judgmental structure [Martin-Löf 1996]. In standard programming languages or type theories there is simply no judgmental structure for a modality. Indeed, ordinary type theory provides a framework only for connectives which are closed under substitutions between local contexts $\Delta \to \Gamma$; most modalities found in mathematics (and, in fact, *all* non-trivial comonadic modalities) fail to be uniform in this sense. We then have to add a new judgmental structure to our programming language without disrupting any of the existing connectives — a delicate task.

For our modality, the new judgmental structure will enforce that a term in $\Box A$ only depends on variables which are themselves under the modality. A variety of judgmental frameworks have been proposed to address this challenge. In particular, Pfenning and Davies [2000] and Clouston [2018] have described calculi for the simply-typed case with good computational properties. For

full Martin-Löf Type Theory, there are proposals such as Shulman [2018] (which roughly follows Pfenning and Davies [2000]) and Clouston et al. [2018] (which extends Clouston [2018]). The previous work has largely focused on the models of each particular type theory and comparatively little time is spent on their syntax. Therefore, while experimental implementations for certain modal dependent type theories have been proposed, there are few proofs of decidability of type checking supporting them. The syntactic presentations of modal dependent type theories remains an interesting question.

A modal type theory. Our new calculus, $\mathsf{MLTT}_{\lock}$, isolates a particular modality and defines an implementable syntax specialized to it. Specifically, $\mathsf{MLTT}_{\lock}$ targets an idempotent comonad which is right adjoint to a monad. This situation may sound arcane, but in fact it is not overly strict and arises in several existing calculi. For instance, the $\Box$ modalities of Davies and Pfenning [1999], and Clouston et al. [2015] as well as the $\flat$ modality of Shulman [2018] satisfy all the required properties; intuitively, each of these modalities arise as a particular global sections functor and this class of modalities always satisfies our requirements.

Our work extends Clouston et al. [2018] by constructing a simpler syntax and proving normalization. Following the Fitch-style presentations of modal logic [Borghuis 1994; Martini and Masini 1996] and their recent adaptions [Clouston 2018], we extend contexts in $\mathsf{MLTT}_{\lock}$ with a *locking* operation, $\Gamma.\lock$; when a variable appears behind a lock, it becomes inaccessible. These locks enable a simple characterization of $\Box A$ by introduction and elimination rules: to construct a proof of $\Box A$, we lock away the entire context and continue by constructing a proof of A. On the other hand, whenever we are trying to prove A, we can delete all of the locks that occur in the context (written $\Gamma^{\unlock}$) and instead shift to proving $\Box A$. These two operations form the introduction and elimination rules for $\Box A$:

$$\textsc{tm/lock}\ \frac{\Gamma.\lock \vdash t : A}{\Gamma \vdash [t]_{\lock} : \Box A} \qquad \textsc{tm/unlock}\ \frac{\Gamma^{\unlock} \vdash t : \Box A}{\Gamma \vdash [t]_{\unlock} : A}$$

These two primitives suffice for deriving the operations of an **S4** necessity modality. For instance, extracting A from $\Box A$ can be done with $\lambda x.\, [x]_{\unlock} : \Box A \to A$. A slightly more complex property is $(A \to \Box B) \to \Box(A \to B)$, which can be proved by the following term: as well: $\lambda f.\, [\lambda a.\, [f(a)]_{\unlock}]_{\lock}$. Many additional properties have been mechanically checked in our implementation, including the constancy of natural numbers ($\mathsf{nat} \to \Box\mathsf{nat}$).

In addition to being convenient to program with, the new connectives have a strong but decidable equational theory. They admit both a β-rule, $[[t]_{\lock}]_{\unlock} = t$, and an η-rule, $[[t]_{\unlock}]_{\lock} = t$. Together they ensure that the equational laws for comonads hold *definitionally* for the modality.

The most subtle point of $\mathsf{MLTT}_{\lock}$ is the novel definition of *substitutions* and the rules associated with them. $\mathsf{MLTT}_{\lock}$ is structured so that, except for the $[-]_{\unlock}$ and $[-]_{\lock}$ operators, locks are entirely silent and substitutions commute with all modal operators. This simplified syntax is essential to formulating a proper normalization algorithm, which is the linchpin of any type-checking algorithm. Moreover, $\mathsf{MLTT}_{\lock}$ satisfies several admissible properties so that locks behave intuitively. For instance, any term which type checks in a locked context will type check in an unlocked context.

Contributions. In summary, we make the following contributions:

- A detailed and well-behaved syntactic presentation of $\mathsf{MLTT}_{\lock}$, a dependent type theory with all standard connectives *and* a necessity modality.
- An extension of normalization by evaluation to account for modalities as well as a proof that NbE is sound and complete for $\mathsf{MLTT}_{\lock}$.

- An extension of Coquand's semantic type-checking algorithm to modal dependent type theory as well as a proof of its soundness and completeness.
- An implementation of MLTT$_{\Box}$ which has been used to mechanize properties of the modality.

For reasons of space, the full proof of the correctness of normalization is given in the accompanying technical report. It includes the full MLTT$_{\Box}$ language, including an infinite hierarchy of cumulative universes.

2 DECLARATIVE SYNTAX OF MLTT$_{\Box}$

We begin by presenting our type theory MLTT$_{\Box}$ in declarative style, extending Martin-Löf's type theory (MLTT) with a necessity modality $\Box A$. Because it is better to walk before attempting to run, we first restrict our attention to the core MLTT language, and then proceed to its extension.

2.1 Introducing dependent type theory

The core MLTT language includes dependent functions and pairs, intensional identity types, and natural numbers. It also includes an infinite hierarchy of universes, but for reasons of space, we opted to omit cumulativity in this presentation, handling it in the accompanying technical report. The syntax of MLTT and some of its rules are presented in Figure 1. MLTT has seven separate forms of judgment:

$\Gamma\ ctx$	"Γ is a context"
$\Gamma \vdash A\ type$	"A is a type in context Γ"
$\Gamma \vdash t : A$	"t is a term of type A in context Γ"
$\Gamma \vdash \delta : \Delta$	"δ is a substitution from Γ to Δ"
$\Gamma \vdash A_0 = A_1\ type$	"A_0 and A_1 are definitionally equal types in context Γ"
$\Gamma \vdash t_0 = t_1 : A$	"t_0 and t_1 are definitionally equal terms of type A in context Γ"
$\Gamma \vdash \delta_0 = \delta_1 : \Delta$	"δ_0 and δ_1 are definitionally equal substitutions from Γ to Δ"

2.1.1 Explicit substitutions. Formally, MLTT and MLTT$_{\Box}$ are variants of Martin-Löf's substitution calculus [Granström 2013; Martin-Löf 1992]; rather than defining substitution as a meta-operation on untyped pre-terms and then establishing the admissibility of a substitution principle, substitution calculi add syntax and corresponding typing judgments for simultaneous substitutions $\Gamma \vdash \delta : \Delta$. Then, substitutions are enacted on terms not through a meta-operation, but rather through a new constructor in the syntax: if $\Gamma \vdash t : A$, then $\Delta \vdash t[\delta] : A[\delta]$.

Then, equational rules are added to the calculus which distribute substitutions through other constructors, mirroring the clauses of the more familiar definition of substitution as a meta-operation on pre-terms. At a high level, the different possibilities for presenting substitution are not substantive, but the use of explicit substitutions is essential for proving the correctness of our normalization algorithm.

Some researchers choose to prove that a calculus with traditional substitution is equivalent to the more mathematically well-behaved version with explicit substitution, but we observe that this is ultimately unnecessary: the use of explicit substitutions is totally transparent and undetectible for a *user* of type theory, since substitutions never appear in user-code.

2.1.2 Binding and names. Rather than explicit variable names x, y, z, the MLTT/MLTT$_{\Box}$ calculi represent variables using De Bruijn indices. A De Bruijn index is a natural number n, which points "upward" to its binder; for instance, the De Bruijn form of the constant function $\lambda x.\, \lambda y.\, x$ is simply $\lambda(\lambda(\mathsf{var}_1))$, whereas the De Bruijn form of the identity function $\lambda x.\, x$ is $\lambda(\mathsf{var}_0)$. The major benefit of

$$
\begin{array}{llll}
\textit{(contexts)} & \Gamma, \Delta & ::= & \cdot \mid \Gamma.A \\
\textit{(types)} & A, B & ::= & t \mid \mathsf{nat} \mid \mathsf{U}_i \mid \Pi(A,B) \mid \Sigma(A,B) \mid \mathsf{Id}(A,t,t) \\
\textit{(terms)} & s, t & ::= & A \mid \mathsf{var}_n \mid \lambda(t) \mid t(t) \mid \langle t, t \rangle \mid \mathsf{fst}(t) \mid \mathsf{snd}(t) \mid \\
 & & & \mathsf{refl}(t) \mid \mathsf{J}(C,t,t) \mid \mathsf{zero} \mid \mathsf{succ}(t) \mid \mathsf{natrec}(A,t,t,t) \mid t[\delta] \\
\textit{(subst.)} & \gamma, \delta & ::= & \mathsf{id} \mid \delta.t \mid \delta \circ \delta \mid \mathsf{p}^k \mid \cdot
\end{array}
$$

$$\textsc{cx/emp}\ \frac{}{\cdot\ ctx} \qquad \textsc{cx/ext}\ \frac{\Gamma\ ctx \qquad \Gamma \vdash A\ type}{\Gamma.A\ ctx} \qquad \textsc{tp/pi-sig}\ \frac{\Gamma \vdash A\ type \qquad \Gamma.A \vdash B\ type}{\Gamma \vdash \Pi(A,B)\ type \qquad \Gamma \vdash \Sigma(A,B)\ type} \qquad \textsc{tp/russell}\ \frac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash A\ type}$$

$$\textsc{tp/esubst}\ \frac{\Gamma \vdash \delta : \Delta \qquad \Delta \vdash A\ type}{\Gamma \vdash A[\delta]\ type} \qquad \textsc{tm/var}\ \frac{\Gamma_0.A.\Gamma_1\ ctx \qquad k = \|\Gamma_1\|}{\Gamma_0.A.\Gamma_1 \vdash \mathsf{var}_k : A[\mathsf{p}^{k+1}]} \qquad \textsc{tm/lam}\ \frac{\Gamma \vdash A\ type \qquad \Gamma.A \vdash t : B}{\Gamma \vdash \lambda(t) : \Pi(A,B)}$$

$$\textsc{tm/pair}\ \frac{\Gamma \vdash t_0 : A \qquad \Gamma.A \vdash B\ type \qquad \Gamma \vdash t_1 : B[\mathsf{id}.t_0]}{\Gamma \vdash \langle t_0, t_1 \rangle : \Sigma(A,B)}$$

$$\textsc{tm/snd}\ \frac{\Gamma \vdash t : \Sigma(A,B) \qquad \Gamma \vdash A\ type \qquad \Gamma.A \vdash B\ type}{\Gamma \vdash \mathsf{snd}(t) : B[\mathsf{id}.\mathsf{fst}(t)]} \qquad \textsc{tm/esubst}\ \frac{\Gamma \vdash \delta : \Delta \qquad \Delta \vdash t : A}{\Gamma \vdash t[\delta] : A[\delta]}$$

$$\textsc{tm/conv}\ \frac{\Gamma \vdash A = B\ type \qquad \Gamma \vdash t : A}{\Gamma \vdash t : B} \qquad \textsc{sb/ext}\ \frac{\Delta \vdash A\ type \qquad \Gamma \vdash \delta : \Delta \qquad \Gamma \vdash t : A[\delta]}{\Gamma \vdash \delta.t : \Delta.A} \qquad \textsc{sb/weaken-1}\ \frac{\Gamma_0.\Gamma_1\ ctx \| \Gamma_1 \| = k}{\Gamma_0.\Gamma_1 \vdash \mathsf{p}^k : \Gamma_0}$$

$$\textsc{tmeq/pi}\ \frac{\Gamma \vdash A\ type \qquad \Gamma \vdash t : \Pi(A,B)}{\Gamma \vdash \lambda(t[\mathsf{p}^1](\mathsf{var}_0)) = t : \Pi(A,B)} \qquad \textsc{tmeq/sig}\ \frac{\Gamma \vdash A\ type \qquad \Gamma.A \vdash B\ type \qquad \Gamma \vdash t : \Sigma(A,B)}{\Gamma \vdash \langle \mathsf{fst}(t), \mathsf{snd}(t) \rangle = t : \Sigma(A,B)}$$

Fig. 1. Selected syntax and typing rules for MLTT; the extension to MLTT🔒 is presented in Figure 3.

De Bruijn indices, in both practical implementation and metatheory, is that they provide canonical representatives of α-equivalence classes, eliminating the paperwork of renaming bound variables.

The main forms of substitution are *weakening* p^n (which weakens the last n variables in the context) and *extension* $\gamma.t$ (which extends the substitution γ with the term t). We additionally have the identity substitution id (which does nothing), a terminal substitution $\cdot$ (which goes from any context to the empty context), and the composition of two substitutions $\delta \circ \gamma$. As an example, the substitution $\mathsf{id}.t$ substitutes the term t for the *last* variable in the context.

Notation 2.1 (Non-dependent function types)**.** When using De Bruijn indices, weakening a variable is explicit rather than silent. Using an explicit substitution, we can define a non-dependent function type $A \longrightarrow B$ in terms of the dependent function type $\Pi(A, B[\mathsf{p}^1])$.

Notation 2.2 (Named variables)**.** In our examples, we will feel free to use a notation with explicit names; likewise, our prototype implementation of MLTT🔒 uses standard named variables in its surface language, resolving these to De Bruijn indices during an elaboration step between parsing

and type checking. We will write $\lambda x.\, t$, $(x : A) \to B$ and $(x : A) \times B$ for the λ-abstraction, dependent function and dependent pair types.

2.1.3 Definitional equality. When are two terms of the same type definitionally equal to each other?[1] This is the fundamental decision for a designer of type theories, not least because adding and removing equations can drastically alter the set of terms which can be typed. Because type-theoretic languages are designed with ease of use in mind, it is generally desirable to include as many equations as possible without disrupting important properties of the language (such as decidability of type checking).

For example, if the equation $1 + 1 = 2$ was not definitional and required explicit proof, the size of proofs would quickly explode (and they would be nearly impossible to write!). On the other hand, if this equation holds definitionally, any conversion steps involving this equation are elided from the term; a consequence of this convenience is that the implementation of a type checker must correctly discharge such equations.

The class of equations which hold definitionally varies widely between type theories; choosing an appropriate notion of definitional equality is a matter of balancing trade-offs (simplicity, efficiency, usability, mathematical meaning), and has an empirical component. In $\mathsf{MLTT}_{\blacksquare}$, we have included η-rules for both dependent functions and dependent pairs, which express the equations $\lambda x.\, t(x) = t$ (when $x \notin t$) and $\langle \mathsf{fst}(t), \mathsf{snd}(t) \rangle = t$.

Deciding definitional equality in the presence of η-laws is challenging. A naïve approach for deciding definitional equality is to reduce each term as much as possible and then to compare for syntactic equality. This obvious way to extend reduction to the η-laws is already unwieldy for function types, and actually breaks down for product types, leading to a failure of confluence which disrupts the transitivity of the resulting algorithmic notion of equivalence. It is currently an open question whether reduction can be used to decide definitional equivalence for a version of type theory with dependent function and pair types; rather than attempt to answer this question, we will use a more streamlined *reduction-free* technique for deciding definitional equality: normalization by evaluation.

2.1.4 Presuppositions and admissible rules. In the *semantics* of Martin-Löf's type theory, a form of judgment like $\Gamma \vdash A\ \mathit{type}$ is explained by *first* specifying what are the meaningful instances; for instance $5 \vdash A\ \mathit{type}$ is never meaningful. The conditions under which a judgment is meaningful are referred to as its "presupposition" [Martin-Löf 1996; Schroeder-Heister 1987]; we would say, for instance, that $\Gamma \vdash A\ \mathit{type}$ presupposes $\Gamma\ \mathit{ctx}$.

On the other hand, when developing a *syntax* for type theory, it is often simplest to write the rules in such a way that the presuppositions become *closure conditions* of the logic, or admissible rules.

Theorem 2.3 (Presupposition).

(1) *If* $\Gamma \vdash A\ \mathit{type}$ *then* $\Gamma\ \mathit{ctx}$.
(2) *If* $\Gamma \vdash t : A$ *then* $\Gamma \vdash A\ \mathit{type}$.
(3) *If* $\Gamma_0 \vdash \gamma : \Gamma_1$ *then* $\Gamma_i\ \mathit{ctx}$.
(4) *If* $\Gamma \vdash A_0 = A_1\ \mathit{type}$ *then* $\Gamma \vdash A_i\ \mathit{type}$.
(5) *If* $\Gamma \vdash t_0 = t_1 : A$ *then* $\Gamma \vdash t_i : A$.
(6) *If* $\Gamma \vdash \delta_0 = \delta_1 : \Delta$ *then* $\Gamma \vdash \delta_i : \Delta$.

Note that the above is *not* saying that we have a rule which concludes $\Gamma\ \mathit{ctx}$ from $\Gamma \vdash A\ \mathit{type}$; it is an external statement about derivability in the formal system. To ensure that Theorem 2.3 holds,

[1] By *definitional equality*, we mean the equivalence relation which requires no proof; in many type theories, including $\mathsf{MLTT}_{\blacksquare}$, an identity type $\mathsf{Id}(A, M, N)$ is used to express equations which *do* require proof.

we must add some auxiliary premises to the rules of the type theory such as the $\Gamma_0.A.\Gamma_1\ ctx$ premise in TM/VAR or both of the type premises in TM/SND. From a normalization result, one can show that many of these premises are ultimately unnecessary, but in order to stage the metatheory properly, we must include them at first.

2.2 MLTT$_🔒$: a modal extension of MLTT

We want to extend MLTT with a *necessity modality* $\Box A$, which contains the elements of A which don't depend on any local variables; for this to make any sense, A must also be a type which doesn't depend on any local variables either. What do we mean by "local variables"? We are being intentionally vague at the moment, but an element $t : \Box A$ *should* be allowed to depend on some variable $x : \Box B$ (a "global variable"), but not on some arbitrary $x : B$.

We also expect that $\Box A$ should have the structure of a *comonad*: that is, we should be able to exhibit elements $\mathsf{extract} : \Box A \to A$ and $\mathsf{dup} : \Box A \to \Box\Box A$ which satisfy the comonad laws. Realizing all these goals in a way that preserves the crucial (and fragile!) syntactic properties of dependent type theory is extremely subtle. One might first attempt to explain $\Box$ using an introduction rule which simply forces the element to not use any variables which are not of the form $x : \Box A$:

$$\frac{\Gamma \vdash t : A}{\Box\Gamma \vdash \mathsf{lock}(t) : \Box A}\ \textsc{tm/lock/bad}^*$$

This attempt immediately fails to preserve the critical syntactic properties of MLTT (such as substitution), but one could consider increasingly sophisticated versions of the idea which, for instance, allowed assumptions like $x : \Box A \times \Box B$, *etc.* as in Prawitz's [1967] notion of "essentially modal" context. While any approach that restricts a context in the conclusion of a rule seems doomed to failure in the context of dependent type theory (in which substitution plays a critical role), there is a kernel of truth in the naïve rule which we intend to nurture.

Necessity: the view from the left. While the conclusion of TM/LOCK/BAD* attempts to strangle the left-hand side of the turnstile into submission, we adapt the Fitch-style approach [Clouston 2018] which achieves the same end for *arbitrary* Γ, by adjusting the context in the *premise* instead. This is achieved by extending our type theory with a new kind of assumption which records that we may not use the local assumptions of Γ, written $\Gamma.🔒$. Then, the variable rule is restricted so that it cannot see anything in the context to the left of a lock:

$$\frac{\Gamma_0.A.\Gamma_1\ ctx \qquad \|\Gamma_1\| = k \qquad 🔒 \notin \Gamma_1}{\Gamma_0.A.\Gamma_1 \vdash \mathsf{var}_k : A[\mathsf{p}^{k+1}]}$$

In our formulation, the locks do *not* count when determining the length of a context; so $\|\Gamma\| = \|\Gamma.🔒\|$.

Using this new form of context, the force of $\Box A$ can be made *conditional*: "Assuming we restrict access to local variables, then we have an element of A"; this is captured by the following introduction rule:

$$\frac{\Gamma.🔒 \vdash t : A}{\Gamma \vdash [t]_🔒 : \Box A}\ \textsc{tm/lock}$$

The TM/LOCK rule imposes no conditions on the shape that Γ must take; rather than being forced to search through the context to remove local variables, we now have the ability to tag them as inaccessible by hiding them behind a lock. This is crucial for obtaining a syntax which respects substitution.

Semantically, the appropriate elimination rule would simply invert TM/LOCK; since, however, we insist on closing $\mathsf{MLTT}_{🔒}$ under substitutions, we must not restrict the context in a conclusion of a rule in that way. A syntactically appropriate presentation would (equivalently) remove locks in the premise rather than adding them in the conclusion. Writing $\Gamma^{🔓}$ for a version of the context Γ with all locks removed, we add the following elimination rule:

$$\frac{\Gamma^{🔓} \vdash t : \Box A \qquad \Gamma \vdash A\ \mathit{type}}{\Gamma \vdash [t]_{🔓} : A}\ \textsc{tm/unlock}$$

In Clouston et al. [2018], a similar elimination rule was presented. In that type theory, however, $[-]_{🔓}$ was required to remove *precisely* one lock, while in $\mathsf{MLTT}_{🔒}$ we delete an arbitrary number. This difference means that in $\mathsf{MLTT}_{🔒}$, $\Box A$ behaves as a *comonad* instead of merely being equipped with the (<*>) operation of applicative functors [McBride and Paterson 2008].

2.3 Programming in $\mathsf{MLTT}_{🔒}$

Before getting deep into technical details (see Section 2.4), we explore some examples of programming in $\mathsf{MLTT}_{🔒}$ to get a feel for the language. To start with, we will exhibit the comonad structure of $\Box A$, fixing $\Gamma.🔒 \vdash A\ \mathit{type}$:

$$\mathsf{extract}_A : \Box A \to A \qquad \mathsf{dup}_A : \Box A \to \Box\Box A$$
$$\mathsf{extract}_A \triangleq \lambda x.\, [x]_{🔓} \qquad \mathsf{dup}_A \triangleq \lambda x.\, [[[x]_{🔓}]_{🔒}]_{🔒}$$

How do these operations work? For $\mathsf{extract}_A$, we are given $x : \Box A$ and wish to construct A. At this point, the only applicable move is use the elimination rule for $\Box A$, namely $[x]_{🔓}$. Notice that, while TM/UNLOCK removes all the locks from the context, there were no locks to remove; this operation is only definable because we have made "deleting no locks" a valid use of $[-]_{🔓}$.

When constructing the dup_A operation, we start by applying as many introduction rules as possible, leaving $\lambda x.\, [[?]_{🔒}]_{🔒}$. We need to construct some term of type A in a context with $x : \Box A$ behind two locks. At this point we cannot access x directly and we cannot apply any further introduction rules, so we must use TM/UNLOCK to clear away the locks. After this, we must construct $\Box A$, not just A, but we are free now to use the assumption $x : \Box A$. Just as $\mathsf{extract}_A$ relies on being able to delete no locks, dup_A is only possible to implement because TM/UNLOCK is able to delete multiple locks.

In addition to being a comonad, $\Box A$ satisfies Axiom **K** from modal logic (the (<*>) operation of an applicative functor [McBride and Paterson 2008]):

$$\circledast_{A,B} : \Box(A \to B) \to \Box A \to \Box B$$
$$\circledast_{A,B} \triangleq \lambda f.\, \lambda x.\, [[f]_{🔓}([x]_{🔓})]_{🔒}$$

Dependent types: boxing the universe. In fact, rather than working schematically with types $\Gamma.🔒 \vdash A\ \mathit{type}$, we can define the operations above in greater generality using type-theoretic universes U_i underneath the modality:

$$\mathsf{extract} : (A : \Box \mathsf{U}_i) \to \Box[A]_{🔓} \to [A]_{🔓} \qquad \mathsf{dup} : (A : \Box \mathsf{U}_i) \to \Box[A]_{🔓} \to \Box\Box[A]_{🔓}$$
$$\mathsf{extract} \triangleq \lambda A.\, \lambda x.\, [x]_{🔓} \qquad \mathsf{dup} \triangleq \lambda A.\, \lambda x.\, [[[x]_{🔓}]_{🔒}]_{🔒}$$

$$\circledast : (A : \Box \mathsf{U}_i) \to (B : \Box \mathsf{U}_i) \to \Box([A]_{🔓} \to [B]_{🔓}) \to \Box[A]_{🔓} \to \Box[B]_{🔓}$$
$$\circledast = \lambda A.\, \lambda B.\, \lambda f.\, \lambda x.\, [[f]_{🔓}([x]_{🔓})]_{🔒}$$

In the above, we used the elimination form to obtain types (elements of the universes) from *modal assumptions* of type $\Box \mathsf{U}_i$. The unity between the treatment of modal types and their elements is one of the main advantages of the calculus we present here.

Equational theory. What equations hold for terms that use the necessity modality? Many modern type theories include both β-rules (eliminating an introduction form is an identity) and η-rules (every element is equal to an introduction form):

$$\textsc{tm/unlock-lock}\ \frac{\Gamma^{\unlock,\lock} \vdash t : A}{\Gamma \vdash [[t]_{\lock}]_{\unlock} = t : A} \qquad \textsc{tm/lock-unlock}\ \frac{\Gamma \vdash t : \Box A}{\Gamma \vdash t = [[t]_{\unlock}]_{\lock} : \Box A}$$

The TM/UNLOCK-LOCK and TM/LOCK-UNLOCK rules are in fact sufficient to establish the comonad laws for $\Box A$. For instance, to verify the law that $\mathsf{extract}_{\Box A}(\mathsf{dup}_A(t)) = t : \Box A$, we calculate:

$$\begin{aligned}
\mathsf{extract}_{\Box A}(\mathsf{dup}_A(t)) &= (\lambda x.\, [x]_{\unlock})((\lambda x.\, [[[x]_{\unlock}]_{\lock}]_{\lock})(t)) \\
&= [(\lambda x.\, [[[x]_{\unlock}]_{\lock}]_{\lock})(t)]_{\unlock} \\
&= [[[[t]_{\unlock}]_{\lock}]_{\lock}]_{\unlock} \\
&= [[t]_{\unlock}]_{\lock} \\
&= t
\end{aligned}$$

Remark 2.4. One subtlety in the equations for $\Box A$ is that the premise of TM/LOCK-UNLOCK is more restrictive than it appears at first. It is not always valid to η-reduce a term, *e.g.*, $[[t]_{\unlock}]_{\lock}$ to t. In order for this reduction to be well-typed, the term t must be well-typed under all the locks in the ambient context, *i.e., t only* relies on $[-]_{\unlock}$ to remove the single lock introduced by $[-]_{\lock}$. For instance, in the definition of dup_A, there seems to be an η-contractible expression, but the contracted term $\mathsf{dup}_A = \lambda x.\, [x]_{\lock}$ would be ill-typed.

This side-condition is the reason why it is simpler to decide definitional equality for $\mathsf{MLTT}_{\lock}$ using NbE than using rewriting. It is always valid to η-expand a term, and so the normalization procedure can be relatively simple-minded for $\Box A$. On the other hand, any rewriting system which seeks to η-reduce terms must carefully maintain the invariant that it never apply an η-reduction leading to an ill-typed term.

For a final example, we will demonstrate that natural numbers are a *constant type*, that is, that there is a function $\mathsf{nat} \to \Box\mathsf{nat}$. Since $A \to \Box A$ is not generally inhabited, we must rely on the particulars of nat:

$$\begin{aligned}
&\mathsf{con}_{\mathsf{nat}} : \mathsf{nat} \to \Box\mathsf{nat} \\
&\mathsf{con}_{\mathsf{nat}} \triangleq \lambda n.\, \mathsf{natrec}(_.\, \Box\mathsf{nat}, [\mathsf{zero}]_{\lock}, _, p.\, [\mathsf{succ}([p]_{\unlock})]_{\lock}, n)
\end{aligned}$$

This function proceeds by recursion on the argument. We cannot construct $[n]_{\lock}$, but if we know that n is zero, we can construct $[\mathsf{zero}]_{\lock}$. For the inductive case, if we know that n is of the form $\mathsf{succ}(n')$, and that $\mathsf{con}_{\mathsf{nat}}(n') = p$, we can construct the successor of p as a constant term: $[\mathsf{succ}([p]_{\unlock})]_{\lock}$. While the notation for the recursor on natural numbers is compact in the calculus, in the implementation we use a more traditional notation. The machine-checkable version of $\mathsf{con}_{\mathsf{nat}}$ is presented in Figure 2.

A final summary of the changes from MLTT to $\mathsf{MLTT}_{\lock}$ is presented in Figure 3.

```
let con : nat → [box nat] =
  fun n →
  rec n at _ → [box nat] with
  | zero → [lock zero]
  | suc _, p → [lock suc [unlock p]]
```

Fig. 2. The code of $\mathsf{con}_{\mathsf{nat}}$ in our experimental implementation of MLTT🔒.

$$\begin{array}{llll} \textit{(contexts)} & \Gamma, \Delta & ::= & \cdots \mid \Gamma.🔒 \\ \textit{(types)} & A, B & ::= & \cdots \mid \Box A \\ \textit{(terms)} & s, t & ::= & \cdots \mid [t]_🔒 \mid [t]_{🔓} \end{array}$$

$$\textsc{cx/lock}\ \frac{\Gamma\ ctx}{\Gamma.🔒\ ctx} \qquad \textsc{tp/box}\ \frac{\Gamma.🔒 \vdash A\ type}{\Gamma \vdash \Box A\ type} \qquad \textsc{tm/box}\ \frac{\Gamma.🔒 \vdash A : \mathsf{U}_i}{\Gamma \vdash \Box A : \mathsf{U}_i} \qquad \textsc{tm/lock}\ \frac{\Gamma.🔒 \vdash t : A}{\Gamma \vdash [t]_🔒 : \Box A} \qquad \textsc{tm/unlock}\ \frac{\Gamma^{🔓} \vdash t : \Box A \qquad \Gamma \vdash A\ type}{\Gamma \vdash [t]_{🔓} : A}$$

$$\textsc{tm/unlock-lock}\ \frac{\Gamma^{🔓}.🔒 \vdash t : A}{\Gamma \vdash [[t]_🔒]_{🔓} = t : A} \qquad \textsc{tm/lock-unlock}\ \frac{\Gamma \vdash t : \Box A}{\Gamma \vdash t = [[t]_{🔓}]_🔒 : \Box A} \qquad \textsc{tp/box-subst}\ \frac{\Gamma \vdash \delta : \Delta \qquad \Delta.🔒 \vdash A\ type}{\Gamma \vdash (\Box A)[\delta] = \Box(A[\delta])\ type}$$

$$\textsc{tm/lock-subst}\ \frac{\Gamma \vdash \delta : \Delta \qquad \Delta.🔒 \vdash t : T}{\Gamma \vdash [t]_🔒[\delta] = [t[\delta]]_🔒 : (\Box T)[\delta]} \qquad \textsc{tm/unlock-subst}\ \frac{\Gamma \vdash \delta : \Delta \qquad \Delta^{🔓} \vdash t : \Box T}{\Gamma \vdash [t]_{🔓}[\delta] = [t[\delta]]_{🔓} : T[\delta]}$$

Fig. 3. Selected new rules of MLTT🔒.

2.4 Sweating the details: admissibilities and substitutions

In order to validate the admissibilities which we required in Section 2.1.4, we must impose some additional closure conditions having to do with locks. Our first admissible rule expresses the intuition that having a lock in the context only makes it *harder* to prove something, never easier:

Theorem 2.5 (Lock Strengthening). *Letting $\mathcal{J}$ range over any judgment, if $\Gamma_0.🔒.\Gamma_1 \vdash \mathcal{J}$, then $\Gamma_0.\Gamma_1 \vdash \mathcal{J}$.*

A related principle is that a judgment which holds with a lock in one position should also hold if the lock is moved leftward in the context; note that this principle only makes sense if Theorem 2.5 holds.

Theorem 2.6 (Lock-variable exchange). *Suppose that $\Gamma_0.🔒 \vdash A\ type$ holds. If $\Gamma_0.A.🔒.\Gamma_1 \vdash \mathcal{J}$ then $\Gamma_0.🔒.A.\Gamma_1 \vdash \mathcal{J}$.*

Finally, we have an admissible rule which allows locks to be *duplicated* (or contracted, depending on perspective). The admissiblity of this rule ensures that the concrete number of locks in front of a variable is not significant, beyond whether it is non-zero; when programming in MLTT🔒 the only question that matters is whether a variable is behind any locks at all.

Theorem 2.7 (Lock contraction). *If $\Gamma_0.🔒.\Gamma_1 \vdash \mathcal{J}$ then $\Gamma_0.🔒.🔒.\Gamma_1 \vdash \mathcal{J}$.*

Corollary 2.8. *If $\Delta \vdash \gamma : \Gamma$ then $\Delta^{🔓} \vdash \gamma : \Gamma^{🔓}$.*

Satisfying these admissibilities is crucial for proving Theorem 2.3 and, in particular, justifying TP/BOX-SUBST, TM/LOCK-SUBST, and TM/UNLOCK-SUBST. Imagine, for instance, that Corollary 2.8 failed to hold; then when we attempt to show that TM/UNLOCK-SUBST satisfied Theorem 2.3, we would have to deduce that $\Gamma\unlock \vdash t[\delta] : A[\delta]$ from only $\Delta\unlock \vdash t : A$ and $\Gamma \vdash \delta : \Delta$, which is unlikely to work. Moreover, without rules like TM/LOCK-SUBST, we would lose the ability to push the explicit substitutions to the leaves of a term, a crucial property for normalization.

Satisfying all of these admissibilities, moreover, requires changes to some of the standard rules of type theory. It ought to be the case that because $\Gamma.\lock.\mathsf{nat} \vdash \mathsf{id} : \Gamma.\lock.\mathsf{nat}$ holds, if Theorem 2.5 is true then there is a derivation of $\Gamma.\mathsf{nat} \vdash \mathsf{id} : \Gamma.\lock.\mathsf{nat}$. Such a derivation does not exist, however, with the existing rule id from MLTT; a similar problem arises for weakenings p^k. To resolve our difficulties, we must generalize the rules for id and p^k in order to allow locks to be silently introduced in an appropriate way.

We introduce an auxiliary judgment $\Gamma \rhd_\lock \Delta$ relating two contexts if Γ arises from Δ through lock strengthenings, contractions, or exchanges; with this in hand, we can define stronger rules for id and p^k:

$$\textsc{sb/id}\ \frac{\Delta \rhd_\lock \Gamma}{\Delta \vdash \mathsf{id} : \Gamma} \qquad \textsc{sb/weaken}\ \frac{\Gamma_0.\Gamma_1\ ctx \qquad \Gamma_0 \rhd_\lock \Gamma_0' \qquad k = \|\Gamma_1\| \qquad \lock \notin \Gamma_1}{\Gamma_0.\Gamma_1 \vdash \mathsf{p}^k : \Gamma_0'}$$

This maneuver trivializes the proofs of Theorems 2.5 to 2.7 for id and p^n but it does not disrupt the rest of the system. The necessity of these technical changes was only apparent after several failed attempts to prove the correctness of the naïve rules; while these changes are small, they are essential for formulating a syntactic account of any modality enjoying admissibilities like Theorems 2.5 to 2.7. Type theorists know from experience that when it comes to syntax, nothing is "obvious".

3 NORMALIZATION BY EVALUATION FOR MLTT$\lock$

Normalization by evaluation (NbE) is a *reduction-free* technique for obtaining normal forms, or canonical representatives of equivalence classes — inducing an algorithm to decide definitional equivalence and thence typing.[2] Like many modern type theories, MLTT$\lock$ has both β-rules and η-rules for the dependent function and dependent pair types. On top of this, MLTT$\lock$ adds the β- and η-rules for $\Box A$. These η-rules force one to consider a *type-sensitive* notion of normal form, singling out the terms which contain no β-redexes and are maximally η-expanded.

Before explaining the algorithm, we will *specify* it. In particular, we will have a partial operation $\underline{\mathbf{nbe}}^{\mathsf{tp}}_\Gamma(A)$ which gives the normal form of the type A in context Γ; and a partial operation $\underline{\mathbf{nbe}}^{A}_\Gamma(t)$ which gives the normal form of the term t at type A in context Γ. Next, we state a *completeness* theorem which, in essence, says that these partial operations are total *functions* on definitional equivalence–classes of well-formed types and terms:

THEOREM 3.1 (COMPLETENESS).

(1) *If* $\Gamma \vdash A_0 = A_1$ *type, then there is exactly one term* A *such that* $\underline{\mathbf{nbe}}^{\mathsf{tp}}_\Gamma(A_0) = A$ *and* $\underline{\mathbf{nbe}}^{\mathsf{tp}}_\Gamma(A_1) = A$.
(2) *If* $\Gamma \vdash t_0 = t_1 : A$, *then there is exactly one term* t *such that* $\underline{\mathbf{nbe}}^{A}_\Gamma(t_0) = t$ *and* $\underline{\mathbf{nbe}}^{A}_\Gamma(t_1) = t$.

[2] In contrast to properties like *strong normalization* (SN) with respect to an abstract rewriting system, NbE is compatible with an intrinsic view of typed terms quotiented by definitional equivalence. The normalization result is therefore a structure on the type theory itself, equipping each definitional equivalence class with a canonical representative. SN, on the contrary, is a property of a rewriting system on the pre-terms of the type theory: every reduction chain in the system terminates. While SN enables crucial lemmas when proving confluence and other properties of a rewriting system, we do not require SN here because MLTT$\lock$ does not define equality based on reduction.

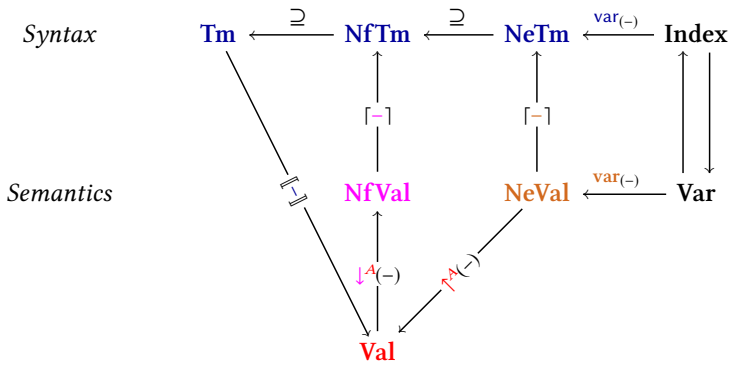


Fig. 4. A bird's eye view of the syntactic and semantic domains involved in NbE, inspired by Abel [2013].

Theorem 3.1 is a basic well-definedness property for the normalization algorithm, but it is surprisingly involved to prove. The essence of the correctness of normalization lies in the soundness theorem below:

Theorem 3.2 (Soundness).

(1) *If* $\Gamma \vdash A$ *type, then* $\Gamma \vdash A = \underline{\mathbf{nbe}}^{\mathrm{tp}}_{\Gamma}(A)$ *type.*
(2) *If* $\Gamma \vdash t : A$*, then* $\Gamma \vdash t = \underline{\mathbf{nbe}}^{A}_{\Gamma}(t) : A$.

The proofs of the two preceeding theorems are carried out in painstaking detail in our accompanying technical report, and treated at a high level here in Sections 5 and 6.

3.1 Warming up to defunctionalized NbE

In the literature, many distinct approaches are described as normalization by evaluation, but in the final analysis they can all be brought back to a single fundamental idea: evaluate syntax into a computational domain, and then quote normal forms back from it. In our presentation of NbE, one works with a variety of domains, summarized schematically in Figure 4.

At a high level, we have domains of values, normal values and neutral values; terms t can be evaluated to values, $[\![t]\!]$. A neutral value e is a variable or some kind of elimination form that is stuck on a variable, and is *reflected* into the values together with its type by the operation $\uparrow^A e$. A value v can be *reified* into a normal value together with its type by the operation $\downarrow^A v$. Finally, both neutral values e and normal values d can be *quoted* into neutral and normal terms $\lceil e \rceil$ and $\lceil d \rceil$ respectively. Then, one obtains the normal form of a closed term $t : A$ by first evaluating, and then reifying, and then quoting: roughly, the normal form of t is $\lceil \downarrow^{[\![A]\!]} [\![t]\!] \rceil$. To normalize an open term, we must consider environments, but for now we are content to convey only the main intuitions.

Representing variables. We have presented the high-level interface to NbE, but there are a number of options available for instantiating it concretely. For instance, the domain of semantic variables **Var** could be instantiated with an inexhaustible set of names, or with De Bruijn indices (matching the syntax); following Abel [2013], we choose to use De Bruijn *levels* in the semantic domain. De Bruijn levels are like indices except that they count from the opposite side of the context; the reason for this somewhat peculiar choice is that it enables normalization and type checking algorithms which never execute a De Bruijn lifting, a critical optimization to enable tractable type checking.

Defunctionalizing NbE. Classical versions of NbE (such as Abel et al. [2007]) often conflate reification /reflection with quotation/evaluation, resulting in the collapse of several of the domains in Figure 4; in these algorithms, the reification operation must eagerly perform η-expansion. The main semantic domain in classical NbE treats binding in a "higher-order" way and therefore must be obtained from a mixed-variance fixed point in some algebraically complete category [Freyd 1991] (such as the category of Scott domains). Implementations of NbE then combined several steps of the algorithm into single operations: $[\![-]\!]_\rho$ would contain the code implementing $\downarrow^A -$ and $\lceil - \rceil_k$ would include $\uparrow^A -$. By requiring a higher-order representation of binders, moreover, it became impossible to formalize the algorithm in a straightforward way in a proof assistant, where such negative occurrences are forbidden.

This view of NbE was refined by Coquand to distinguish reification/reflection from quotation/evaluation, as in Abel et al. [2009]. A final refinement, presented in Abel [2013], involves replacing the higher-order interpretation of binders with syntactic closures, and defunctionalizing the reification and reflection operators. This step unravels enough knots that the use of domain theory can be abandoned, obtaining ordinary *sets* of values, normal values and neutral values. When we say that reification/reflection are "defunctionalized" we mean that they are no longer partial operations which perform η-expansion, but instead are inert *constructors*; the η-expansion is then distributed lazily across the rest of the algorithm in a straightfoward way, and is forced only during quotation.

3.2 The semantic domains

The complete specification of the semantic domains for MLTT$_{\text{🔒}}$ is given informally below:

$$
\begin{array}{lllll}
\textit{(values)} & A, u & ::= & \uparrow^A e \mid \mathbf{\Pi}(A, F) \mid \mathbf{\Sigma}(A, B) \mid \mathbf{Id}(A, u, v) \mid \Box A \mid \mathbf{U}_i \mid \mathbf{nat} \\
 & & & \lambda(f) \mid \langle u, v \rangle \mid \mathbf{refl}(v) \mid \mathbf{lock}(v) \mid \mathbf{zero} \mid \mathbf{succ}(v) \\
\textit{(neutrals)} & e & ::= & \mathbf{var}_k \mid e.\mathbf{app}(d) \mid e.\mathbf{fst} \mid e.\mathbf{snd} \mid e.\mathbf{unlock} \mid e.\mathbf{natrec}(F, v, f) \\
 & & & e.\mathbf{J}(F, f, A, v_1, v_2) \\
\textit{(environments)} & \rho & ::= & \cdot \mid \rho.v \\
\textit{(closures)} & F, f & ::= & t \triangleleft \rho \\
\textit{(normals)} & d & ::= & \downarrow^A v
\end{array}
$$

The values contain constructors for each introduction form, as well as the reflection (suspended η-expansion) $\uparrow^A e$ of a neutral e of type A; a sequence of values forms an *environment*. Binders are represented using a syntactic closure $t \triangleleft \rho$, where ρ provides a value for each free variable in the term t except the variables the abstraction itself binds. A neutral is a variable possibly followed by a spine of stuck elimination forms; finally, a normal value is just a value together with its type annotation (see Section 3.1). It is worth noting that we do not need the semantic domains to be closed under *any* substitution or renaming principle. The only *new* parts of our semantic domain are $\Box -$, $\mathbf{lock}(-)$ and $-.\mathbf{unlock}$, which interpret $\Box -$, $[-]_{\text{🔒}}$ and $[-]_{\text{🔓}}$ respectively.

3.3 Evaluation: from syntax to semantics

A term t with n free variables is evaluated with respect to a semantic environment ρ of length n, written $[\![t]\!]_\rho$ when it is defined. We present a selection of the clauses for the partial evaluation operation in Figure 5, and describe in more detail a few illustrative cases below.

Evaluating functions. To warm up, we consider the case for evaluating the introduction and elimination forms of the dependent function type. Given an environment ρ, we evaluate the λ-abstraction $\lambda(t)$ by constructing a closure and wrapping it in the semantic λ-abstraction, $\lambda(t \triangleleft \rho)$. Evaluating the syntactic application $s(t)$ is more subtle: first we evaluate the function term s with respect to ρ, and then we must proceed by case on the result:

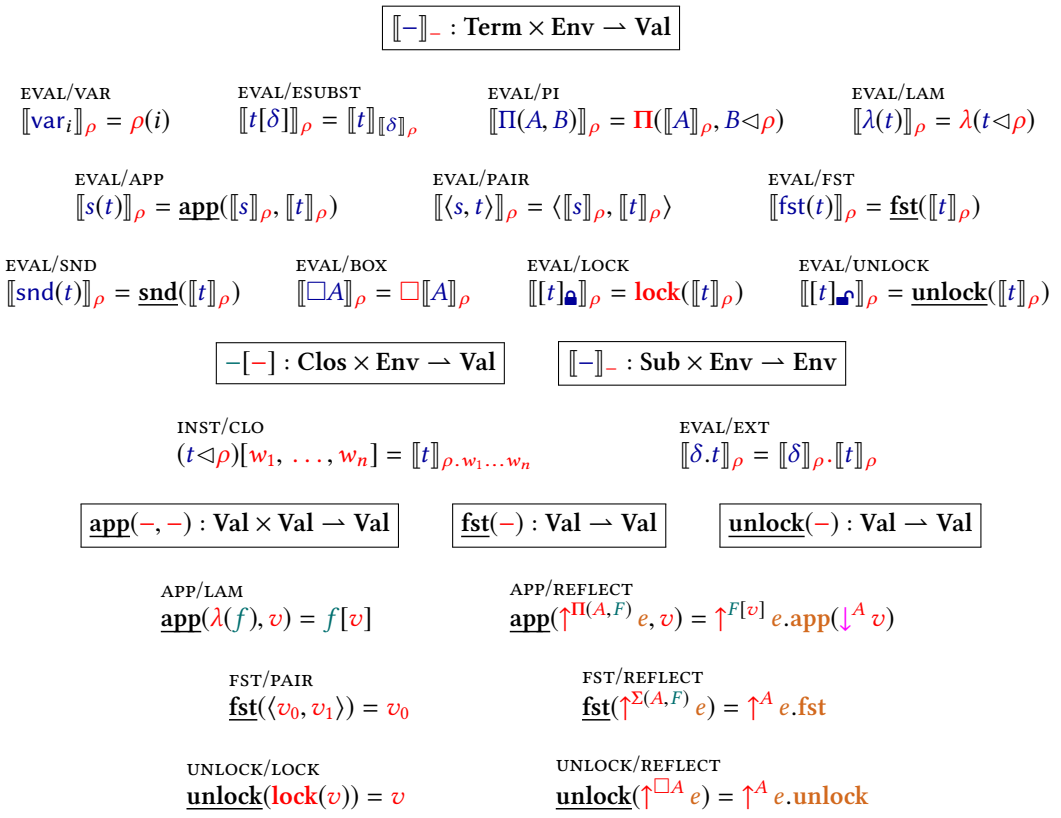


Fig. 5. Selected rules of evaluation.

(1) If the result is a semantic λ-abstraction $\lambda(s' \triangleleft \rho')$, we discard ρ and evaluate s' in the environment ρ' extended by the value of t, returning $[\![s']\!]_{\rho' . [\![t]\!]_\rho}$.
(2) On the other hand, it is possible that $[\![s]\!]_\rho$ is the reflection of a neutral function, $\uparrow^{\Pi(A, B \triangleleft \rho')} e$. In this case, we extend the spine by a neutral application frame, $e.\mathbf{app}(\downarrow^A [\![t]\!]_\rho)$. To reflect this neutral application, we obtain its type by instantiating the closure $B \triangleleft \rho'$, finally returning $\uparrow^{[\![B]\!]_{\rho' . [\![t]\!]_\rho}} e.\mathbf{app}(\downarrow^A [\![t]\!]_\rho)$.

To simplify the procedure above, we factor evaluation into several other partial operations which "enact" each elimination form ($\underline{\mathbf{app}}(-,-)$, $\underline{\mathbf{fst}}(-)$, $\underline{\mathbf{unlock}}(-)$, *etc.*), enabling us to evaluate $s(t)$ as simply $\underline{\mathbf{app}}([\![s]\!]_\rho, [\![t]\!]_\rho)$. We also factor out the instantiation of a closure $F[v]$.

Evaluating the modality. Evaluation for the introduction and elimination forms of the modality is even simpler; the value of $[t]_🔒$ with respect to ρ is just $\mathbf{lock}([\![t]\!]_\rho)$; and the value of $[t]_🔓$ is $\underline{\mathbf{unlock}}([\![t]\!]_\rho)$, where $\underline{\mathbf{unlock}}(\mathbf{lock}(v)) = v$ and $\underline{\mathbf{unlock}}(\uparrow^{\Box A} e) = \uparrow^A e.\mathbf{unlock}$.

3.4 Quotation: from semantics to syntax

Quotation is where the "real work" happens in the defunctionalized version of NbE. We will have three quotation operations, each parameterized by the length of the context: $\lceil A \rceil^{\mathsf{ty}}_n$ quotes a semantic type value A to a term, $\lceil d \rceil_n$ quotes a normal value d to a term, and $\lceil e \rceil_n$ quotes a neutral value

QUO/VAR
$$\lceil \mathsf{var}_k \rceil_n = \mathsf{var}_{n-(k+1)}$$

QUO/REFLECT
$$\lceil \downarrow^{\uparrow^{C} e'} \uparrow^{A} e \rceil_n = \lceil e \rceil_n$$

QUO/TY
$$\lceil \downarrow^{\mathsf{U}_i} v \rceil_n = \lceil v \rceil_n^{\mathsf{ty}}$$

QUO/PI-TP
$$\lceil \Pi(A, F) \rceil_n^{\mathsf{ty}} = \Pi(\lceil A \rceil_n^{\mathsf{ty}}, \lceil F[\uparrow^{A} \mathsf{var}_n] \rceil_{n+1}^{\mathsf{ty}})$$

QUO/PI-EL
$$\lceil \downarrow^{\Pi(A,F)} v \rceil_n = \lambda(\lceil \downarrow^{F[\uparrow^{A} \mathsf{var}_n]} \underline{\mathbf{app}}(v, \uparrow^{A} \mathsf{var}_n) \rceil_{n+1})$$

QUO/APP
$$\lceil e.\mathbf{app}(d) \rceil_n = (\lceil e \rceil_n)(\lceil d \rceil_n)$$

QUO/SG-EL
$$\lceil \downarrow^{\Sigma(A,B)} v \rceil_n = \langle \lceil \downarrow^{A} \underline{\mathbf{fst}}(v) \rceil_n, \lceil \downarrow^{B[\underline{\mathbf{fst}}(v)]} \underline{\mathbf{snd}}(v) \rceil_n \rangle$$

QUO/FST
$$\lceil e.\mathbf{fst} \rceil_n = \mathsf{fst}(\lceil e \rceil_n)$$

QUO/BOX-TP
$$\lceil \Box A \rceil_n^{\mathsf{ty}} = \Box \lceil A \rceil_n^{\mathsf{ty}}$$

QUO/BOX-EL
$$\lceil \downarrow^{\Box A} v \rceil_n = [\lceil \downarrow^{A} \underline{\mathbf{unlock}}(v) \rceil_n]_{🔒}$$

QUO/OPEN
$$\lceil e.\mathbf{unlock} \rceil_n = [\lceil e \rceil_n]_{🔓}$$

Fig. 6. Selected rules of quotation.

e to a term. We present a fragment of the quotation algorithm in Figure 6. It is simple to see by inspection that the image of each quotation function is precisely the β-short/η-long normal forms.

The quotation algorithm for elements of types that have η-laws (such as dependent function, dependent pair and modal types) implements η-expansion by treating the provided value as a black box. For instance, to quote a value v of type $\Box A$, one does not inspect v but instead *unlocks* it, quotes the result in A, and wraps it in the syntactic introduction form for the modality, returning $[\lceil \downarrow^{A} \underline{\mathbf{unlock}}(v) \rceil_n]_{🔒}$. Function and pair types work in an analogous way: first, you apply the semantic elimination operation, then quote, and then wrap in the syntactic introduction form.

A subtle case is the quotation of a semantic variable var_k in a context of length n. Here, k is a De Bruijn *level*, but in the syntax we use De Bruijn *indices*; therefore, we must split the difference using a bit of arithmetic, returning $\mathsf{var}_{n-(k+1)}$.

3.5 Normalization by evaluation

Given a $\Gamma \vdash A$ *type* and $\Gamma \vdash t : A$, what are the normal forms of A and t? We are now nearly equipped to define the normalization operations for types and terms; all that remains is to define an operation to reflect a syntactic context Γ to into a semantic environment, written $\uparrow\Gamma$:

REFLECT/EMP
$$\uparrow\cdot = \cdot$$

REFLECT/SNOC
$$\uparrow\Gamma.A = \uparrow\Gamma.\uparrow^{[\![A]\!]_{\uparrow\Gamma}} \mathsf{var}_{\|\Gamma\|}$$

The reflected context is nothing more than a sequence of semantic variables. Now, we can define normalization for types and for their terms using context reflection, evaluation, reification and quotation:

$$\underline{\mathbf{nbe}}_{\Gamma}^{\mathsf{tp}}(A) = \lceil [\![A]\!]_{\uparrow\Gamma} \rceil_{\|\Gamma\|}^{\mathsf{ty}} \qquad \underline{\mathbf{nbe}}_{\Gamma}^{A}(t) = \lceil \downarrow^{[\![A]\!]_{\uparrow\Gamma}} [\![t]\!]_{\uparrow\Gamma} \rceil_{\|\Gamma\|}$$

We defer the *correctness* of this algorithm to Sections 5 and 6.

4 SEMANTIC TYPE CHECKING

Using our normalization result, it is now possible to define an algorithm to type check a suitably annotated version of MLTT$_{🔒}$. It is unlikely that the terms of MLTT$_{🔒}$ as presented in Section 2 have decidable type checking, but by passing to a version of the calculus $\mathrm{MLTT}_{🔒}^{\leftrightarrows}$ which annotates β-redexes with a type, we do obtain a total algorithm; moreover, we will see that MLTT$_{🔒}$ and

$\mathsf{MLTT}^{\leftrightarrows}_{\lock}$ coincide on their β-normal fragments. Then, using our normalization theorem, we will show in Section 7 that $\mathsf{MLTT}^{\leftrightarrows}_{\lock}$ is *adequate* in a technical sense.

The simplest way to type check dependent types involves splitting the algorithm into two stages: type checking and type synthesis. A type checking problem is to determine whether a term has a given type in a given context, whereas a type synthesis problem is to *infer* a type for a term in a given context. The resulting mutually recursive algorithm is called *bidirectional type checking*, invented by Coquand in 1996 and "broken in" by Pierce and Turner in 2000.

4.1 Bidirectional syntax

The bidirectional type checking algorithm works best when the terms being checked are split into two syntactic categories, depending on whether they support checking or synthesis; we give the grammar of $\mathsf{MLTT}^{\leftrightarrows}_{\lock}$ below:

$$\begin{array}{llll} \textit{(checking)} & A, M, N & ::= & R \mid \Pi(A,B) \mid \Sigma(A,B) \mid \mathsf{Id}(A,M,M) \mid \Box A \mid \mathsf{nat} \mid \mathsf{U}_i \\ & & & \lambda(M) \mid \langle M, N\rangle \mid \mathsf{refl}(M) \mid [M]_{\lock} \mid \mathsf{zero} \mid \mathsf{succ}(M) \\ \textit{(synthesis)} & R, S & ::= & (M : A) \mid \mathsf{var}_n \mid R(M) \mid \mathsf{fst}(R) \mid \mathsf{snd}(R) \mid \mathsf{J}(C,M,R) \mid [R]_{\unlock} \mid \\ & & & \mathsf{natrec}(C,R,M,N) \end{array}$$

Note that we do not include explicit substitutions; as with our semantic domains, we do not in fact require any substitution closure properties at all for the syntax of $\mathsf{MLTT}^{\leftrightarrows}_{\lock}$. A term M of $\mathsf{MLTT}^{\leftrightarrows}_{\lock}$ can be trivially erased to a term M° in $\mathsf{MLTT}_{\lock}$; the only interesting case is $(M : A)^\circ = M^\circ$.

Following Coquand [1996], we define the bidirectional type checking algorithm relative *not* to syntactic contexts Γ and syntactic types A, but rather with respect to a *semantic* kind of context Ξ (defined below) and semantic type values A. This yields a much more efficient algorithm than usual, avoiding the need for expensive De Bruijn liftings; but the algorithm can be lifted to syntactic types and contexts using evaluation and reflection.

4.2 Semantic contexts

Semantic contexts Ξ are annotated versions of environments ρ, storing type information and locks:

$$\textit{(semantic contexts)} \quad \Xi ::= \cdot \mid \Xi.\lock \mid \Xi.\downarrow^A v$$

We write $\Xi.A$ to abbreviate the extension of a semantic context with a new variable, $\Xi.\downarrow^A \uparrow^A \mathbf{var}_{\|\Xi\|}$, where we write $\|\Xi\|$ to mean the length of Ξ (ignoring locks). As was the case for syntactic contexts, we can define an operation which deletes all locks from a semantic context Ξ, written $\Xi^{\unlock}$. These semantic contexts have an evident projection to semantic environments which ignores locks and drops the type annotation on normals, which we write as $|\Xi|$. Syntactic contexts Γ can be transformed into semantic contexts $\Updownarrow\Gamma$ easily. We set $\Updownarrow\cdot = \cdot$ and $\Updownarrow(\Gamma.\lock) = (\Updownarrow\Gamma).\lock$ and $\Updownarrow(\Gamma.A) = (\Updownarrow\Gamma).[\![A]\!]_{\Updownarrow\Gamma}$.

4.3 Checking and synthesis

We will define three mutually recursive algorithmic judgments for bidirectional terms relative to semantic contexts and types:

(1) The judgment $\boxed{\Xi \vdash A \Leftarrow \mathit{type}}$ checks that A is a type in semantic context Ξ.
(2) The judgment $\boxed{\Xi \vdash M \Leftarrow A}$ checks that M is a term of type A in semantic context Ξ.
(3) The judgment $\boxed{\Xi \vdash R \Rightarrow A}$ synthesizes the semantic type A of the term R in context Ξ. It is important to note that A is an *output* of this judgment and not an input.

$$\textsc{sem-synth/var}\ \frac{\Xi(k) = \downarrow^A v}{\Xi \vdash \mathsf{var}_k \Rightarrow A} \qquad \textsc{sem-check/synth}\ \frac{\Xi \vdash R \Rightarrow A \qquad \lceil A \rceil^{\mathsf{ty}}_{\|\Xi\|} = \lceil B \rceil^{\mathsf{ty}}_{\|\Xi\|}}{\Xi \vdash R \Leftarrow B} \qquad \textsc{sem-check-tp/synth}\ \frac{\Xi \vdash R \Rightarrow \mathbf{U}_i}{\Xi \vdash R \Leftarrow \mathit{type}}$$

$$\textsc{sem-synth/check}\ \frac{\Xi \vdash A \Leftarrow \mathit{type} \qquad [\![A^\circ]\!]_{|\Xi|} = A_\Xi \qquad \Xi \vdash t \Leftarrow A_\Xi}{\Xi \vdash (t : A) \Rightarrow A_\Xi}$$

$$\textsc{sem-check-tp/pi}\ \frac{\Xi \vdash A \Leftarrow \mathit{type} \qquad \Xi.[\![A^\circ]\!]_{|\Xi|} \vdash B \Leftarrow \mathit{type}}{\Xi \vdash \Pi(A, B) \Leftarrow \mathit{type}} \qquad \textsc{sem-check/pi}\ \frac{\Xi \vdash A \Leftarrow \mathbf{U}_i \qquad \Xi.[\![A^\circ]\!]_{|\Xi|} \vdash B \Leftarrow \mathbf{U}_i}{\Xi \vdash \Pi(A, B) \Leftarrow \mathbf{U}_i}$$

$$\textsc{sem-check/lam}\ \frac{\Xi.A \vdash M \Leftarrow F[\uparrow^A \mathbf{var}_{\|\Xi\|}]}{\Xi \vdash \lambda(M) \Leftarrow \mathbf{\Pi}(A, F)} \qquad \textsc{sem-synth/app}\ \frac{\Xi \vdash R \Rightarrow \mathbf{\Pi}(A, F) \qquad \Xi \vdash M \Leftarrow A}{\Xi \vdash R(M) \Rightarrow F[[\![M^\circ]\!]_{|\Xi|}]}$$

$$\textsc{sem-check-tp/sg}\ \frac{\Xi \vdash A \Leftarrow \mathit{type} \qquad \Xi.[\![A^\circ]\!]_{|\Xi|} \vdash B \Leftarrow \mathit{type}}{\Xi \vdash \Sigma(A, B) \Leftarrow \mathit{type}} \qquad \textsc{sem-check/sg}\ \frac{\Xi \vdash A \Leftarrow \mathbf{U}_i \qquad \Xi.[\![A^\circ]\!]_{|\Xi|} \vdash B \Leftarrow \mathbf{U}_i}{\Xi \vdash \Sigma(A, B) \Leftarrow \mathbf{U}_i}$$

$$\textsc{sem-check/pair}\ \frac{\Xi \vdash M \Leftarrow A \qquad \Xi \vdash N \Leftarrow F[[\![M^\circ]\!]_{|\Xi|}]}{\Xi \vdash \langle M, N \rangle \Leftarrow \mathbf{\Sigma}(A, F)} \qquad \textsc{sem-synth/fst}\ \frac{\Xi \vdash R \Rightarrow \mathbf{\Sigma}(A, F)}{\Xi \vdash \mathsf{fst}(R) \Rightarrow A} \qquad \textsc{sem-synth/snd}\ \frac{\Xi \vdash R \Rightarrow \mathbf{\Sigma}(A, F)}{\Xi \vdash \mathsf{snd}(R) \Rightarrow F[\underline{\mathbf{fst}}([\![R^\circ]\!]_{|\Xi|})]}$$

$$\textsc{sem-check-tp/box}\ \frac{\Xi.\text{🔒} \vdash A \Leftarrow \mathit{type}}{\Xi \vdash \Box A \Leftarrow \mathit{type}} \qquad \textsc{sem-check/box}\ \frac{\Xi.\text{🔒} \vdash A \Leftarrow \mathbf{U}_i}{\Xi \vdash \Box A \Leftarrow \mathbf{U}_i} \qquad \textsc{sem-check/lock}\ \frac{\Xi.\text{🔒} \vdash M \Leftarrow A}{\Xi \vdash [M]_{\text{🔒}} \Leftarrow \Box A} \qquad \textsc{sem-synth/unlock}\ \frac{\Xi^{\text{🔓}} \vdash R \Rightarrow \Box A}{\Xi \vdash [R]_{\text{🔓}} \Rightarrow A}$$

Fig. 7. Selected semantic type checking rules. Note that $\Xi(k) = \downarrow^A v$ is undefined if $\downarrow^A v$ appears behind a lock in Ξ.

For each type constructor, such as $\Pi(A, B)$, we need a clause to check both $\Xi \vdash \Pi(A, B) \Leftarrow \mathit{type}$ and $\Xi \vdash \Pi(A, B) \Leftarrow \mathbf{U}_i$. It is possible to factor these into the same routine, but we keep them separate for the sake of simplicity.

These judgments are rendered in our implementation as OCaml functions of the following types respectively:

```
val check_tp : sem_ctx → term → bool
val check : sem_ctx → term → value → bool
val synth : sem_ctx → term → value option
```

A selection of clauses from the type checking algorithm is presented in Figure 7, but we will step through some examples to cultivate intuition.

Example 4.1 (SEM-CHECK-TP/PI). Suppose we are trying to check that $\Pi(A, B)$ is a type in context Ξ; first, we check must check that A is a type at Ξ, and then we must do something about B, which

should be a type in an extended context. To extend the context Ξ, we must obtain the value of A; erasing A to an $\mathsf{MLTT}_{\text{🔒}}$-term A°, we can evaluate it with respect to the environment determined by Ξ; we therefore check that B is a type in the context $\Xi.[\![A^\circ]\!]_{|\Xi|}$. This case can be implemented in OCaml as follows:[3]

```
let check_tp ctx ty =
  match ty with
  | Pi (dom, cod) →
    check_tp ctx dom && begin
      let vdom = eval (proj_env ctx) (erase dom) in
      check_tp (ext_ctx ctx vdom) cod
    end
  (* ... *)
```

5 THE COMPLETENESS OF NORMALIZATION BY EVALUATION

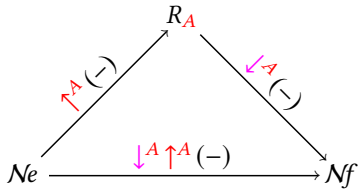
Prior to certifying the type-checking algorithm, we must prove the normalization algorithm correct. The first and easiest correctness condition for a normalization algorithm is *completeness*; roughly, a normalization algorithm is called complete when any two equal terms are taken to *exactly* the same normal form. We recall the full statement of completeness below:

(1) If $\Gamma \vdash A_0 = A_1$ *type* then there is exactly one term A such that $\underline{\mathbf{nbe}}^{\mathsf{tp}}_\Gamma(A_0) = A$ and $\underline{\mathbf{nbe}}^{\mathsf{tp}}_\Gamma(A_1) = A$.
(2) If $\Gamma \vdash t_0 = t_1 : A$ then there is exactly one term t such that $\underline{\mathbf{nbe}}^{A}_\Gamma(t_0) = t$ and $\underline{\mathbf{nbe}}^{A}_\Gamma(t_1) = t$.

The completeness of normalization for dependent type theory can be proved using a semantic model in which every type is interpreted as a *partial equivalence relation* (PER) on semantic values. A partial equivalence relation is a binary relation which is both symmetric and transitive, but not necessarily reflexive; equivalently, one can consider equivalence relations on subsets of the collection of values.

To understand why this works, we must first construct the two fundamental partial equivalence relations on which everything will hinge, namely the PER of neutral values $\mathcal{N}e$ and the PER of (defunctionalized) normal values $\mathcal{N}f$. These relations distinguish the pairs of neutral values (resp. normal values) which are quoted to *the exact same* piece of syntax. For instance, two neutrals e_0, e_1 are related in $\mathcal{N}e$ exactly when we have $\lceil e_0 \rceil_n = \lceil e_1 \rceil_n$ for all de Bruijn levels n.

We then require a critical closure condition, that every type A's PER R_A embeds all of $\mathcal{N}e$, and is embedded in $\mathcal{N}f$; we call this closure condition *saturation*:



(1)

Completeness is obtained immediately from (a) the interpretation of the syntax of $\mathsf{MLTT}_{\text{🔒}}$ into the PER model (equal terms get evaluated to equal elements of the PER), and (b) the fact that every PER in the model is saturated (equal elements in the PER will be quoted to the exact same piece of syntax).

[3]Our actual code is more abstracted than this; we present an elementary version here for intuition.

Scaling PERs to modalities. In ordinary PER models of type theory, each type is given meaning through a partial equivalence relation of its elements. While such an approach could be used to develop the syntactic metatheory of $\mathsf{MLTT}_{\color{blue}\blacksquare}$ (canonicity, normalization, decidability), the proof would not support the extension of $\mathsf{MLTT}_{\color{blue}\blacksquare}$ with any type A for which $A \to \Box A$ is refuted. Because such non-constant types are the *raison d'être* for modal extensions of type theory, we develop a modular proof of normalization using Kripke PERs over an arbitrary non-empty partial order $\mathbb{P}$, enabling extension by non-constant types. The use of Kripke PERs mirrors the semantic situation for modal type theory, in which categories of presheaves play a central role.

A $\mathbb{P}$-PER is a family R of partial equivalence relations indexed in $p : \mathbb{P}$ which is monotone in the sense that if $(u_0, u_1) \in R_p$ and $q \leq p$, then $(u_0, u_1) \in R_q$. Borrowing notation from Kripke forcing, we write $p \Vdash u_0 \sim u_1 \in R$ for $(u_0, u_1) \in R_p$.

Scaling PERs to dependent type theory. The idea of constructing a PER model to prove completeness is simple enough in concept, but to scale the construction to dependent type theory with universes is quite involved. Because types can be computed from terms, we can't define the partial equivalence relations for types "by induction on the type structure", which is the way that relational models are usually defined for simpler programming languages. One needs to specify when two types are equal simultaneously with when two values are equal; this style of definition is "inductive-recursive", and can be constructed concretely in ordinary mathematical foundations using a fixed point on the complete lattice of relations [Allen 1987; Angiuli 2019].

Kripke type systems. Writing $\mathbf{Val}$ for the set of values u, we define the set $\mathbf{Rel}$ of *indexed relations* to be the powerset of $\mathbb{P} \times \mathbf{Val} \times \mathbf{Val}$. In order to simultaneously interpret typehood and type equality with type membership and member equality, we will work with an indexed notion of *type system*; a type system is a relation $\tau \subseteq \mathbb{P} \times \mathbf{Val} \times \mathbf{Val} \times \mathbf{Rel}$. Writing $\tau \models_p A \sim B \downarrow R$ for $(p, A, B, R) \in \tau$, we mean that at stage p, the type system τ regards A, B as equal types with relational interpretation R. We will write $\tau \models_p A \sim B$ for the existential quantification $\exists R.\, \tau \models_p A \sim B \downarrow R$.

At this stage in the construction, we do not place any constraints on indexed relations or type systems: later, after performing a somewhat involved fixed point construction to obtain a cumulative hierachy τ_α for $\alpha \in \mathbb{N} \cup \{\omega\}$ of type systems which model all of $\mathsf{MLTT}_{\color{blue}\blacksquare}$, we prove by induction that our type systems have the following properties:

(1) Each τ_α forms the graph of a partial function $\mathbb{P} \times \mathbf{Val} \times \mathbf{Val} \rightharpoonup \mathbf{Rel}$.
(2) Each relation $\{(p, A, B) \mid \tau_\alpha \models_p A \sim B\}$ is a saturated $\mathbb{P}$-PER.
(3) Whenever $\tau_\alpha \models_p A \sim B \downarrow R$, the relation R is a saturated $\mathbb{P}$-PER.

The type system hierarchy. The type systems τ_α will explain what types are equal at level α, and what their elements are; the first infinite type system τ_ω contains all the types, including every finite universe $\mathbf{U}_i$. Each type system τ_α is constructed using an *inductive definition* which closes under all the connectives and base types of $\mathsf{MLTT}_{\color{blue}\blacksquare}$; a fragment of this definition is presented in Figure 8. Of particular note is the clause for the necessity modality:

$$\frac{\forall q.\, \tau_\alpha \models_q A_0 \sim A_1 \downarrow R(q)}{\tau_\alpha \models_p \Box A_0 \sim \Box A_1 \downarrow \{(q, u_0, u_1) \mid q \Vdash \underline{\mathbf{unlock}}(u_0) \sim \underline{\mathbf{unlock}}(u_1) \in R(q)\}}$$

This clause says that two instances of the necessity modality $\Box A_0, \Box A_1$ are equal at stage p when the types A_0, A_1 are equal *at all stages* q; the $\mathbb{P}$-PER assigned to the modality likewise quantifies over all stages, and implicitly implements the η-rule of the modality by way of $\underline{\mathbf{unlock}}(-)$. This universal quantification over stages reflects the concrete interpretation of the necessity modality in semantic models, such as the topos of trees [Birkedal et al. 2011; Clouston et al. 2015].

$$\frac{}{\tau_\alpha \models_p \mathbf{nat} \sim \mathbf{nat} \downarrow [\![\mathbb{N}]\!]} \qquad \frac{(j < \alpha)}{\tau_\alpha \models_p \mathbf{U}_j \sim \mathbf{U}_j \downarrow \{(q, A_0, A_1) \mid \tau_j \models_q A_0 \sim A_1\}}$$

$$\frac{\tau_\alpha \models_p A_0 \sim A_1 \downarrow R \qquad \tau_\alpha \models_q R \gg B_0 \sim B_1 \downarrow S}{\tau_\alpha \models_p \mathbf{\Pi}(A_0, B_0) \sim \mathbf{\Pi}(A_1, B_1) \downarrow [\![\Pi]\!](R, S)}$$

$$\frac{\forall q.\, \tau_\alpha \models_q A_0 \sim A_1 \downarrow R(q)}{\tau_\alpha \models_p \Box A_0 \sim \Box A_1 \downarrow \{(q, u_0, u_1) \mid q \Vdash \underline{\mathbf{unlock}}(u_0) \sim \underline{\mathbf{unlock}}(u_1) \in R(q)\}}$$

$$\frac{\forall q \leq p.\, \forall (q \Vdash u_0 \sim u_1 \in R).\, \tau \models_q B_0[u_0] \sim B_1[u_1] \downarrow S(u_0, u_1)}{\tau \models_p R \gg B_0 \sim B_1 \downarrow S}$$

$$\frac{}{p \Vdash \mathbf{zero} \sim \mathbf{zero} \in [\![\mathbb{N}]\!]} \qquad \frac{p \Vdash u_0 \sim u_1 \in [\![\mathbb{N}]\!]}{p \Vdash \mathbf{succ}(u_0) \sim \mathbf{succ}(u_1) \in [\![\mathbb{N}]\!]} \qquad \frac{e_0 \sim e_1 \in \mathcal{N}e}{p \Vdash \uparrow^{\mathbf{nat}} e_0 \sim \uparrow^{\mathbf{nat}} e_1 \in [\![\mathbb{N}]\!]}$$

$$\frac{\forall q \leq p.\, \forall (q \Vdash v_0 \sim v_1 \in R).\, q \Vdash \underline{\mathbf{app}}(u_0, v_0) \sim \underline{\mathbf{app}}(u_1, v_1) \in S(v_0, v_1)}{p \Vdash u_0 \sim u_1 \in [\![\Pi]\!](R, S)}$$

Fig. 8. A fragment of the inductive definition of the type system hierarchy τ_α.

Interpreting the judgments of MLTT$_{\text{🔒}}$. The validity of each formal judgment $\Gamma \vdash \mathcal{J}$ is interpreted as a statement $\Gamma \vDash \mathcal{J}$ about the ultimate type system τ_ω. Hypothetical judgments are intepreted by quantifying over equal semantic environments $p \Vdash \rho_0 = \rho_1 : \Gamma$ (a relation which we omit for reasons of space). The validity conditions for the judgments of MLTT$_{\text{🔒}}$ is specified below:

(1) $\Gamma \vDash A_0 = A_1$ *type* holds when for all $p : \mathbb{P}$ and $p \Vdash \rho_0 = \rho_1 : \Gamma$, we have $\tau_\omega \models_p [\![A_0]\!]_{\rho_0} \sim [\![A_1]\!]_{\rho_1}$.
(2) $\Gamma \vDash t_0 = t_1 : A$ holds when for all $p : \mathbb{P}$ and $p \Vdash \rho_0 = \rho_1 : \Gamma$, we have both $\tau_\omega \models_p [\![A_0]\!]_{\rho_0} \sim [\![A_1]\!]_{\rho_1} \downarrow R$ and $p \Vdash [\![t_0]\!]_{\rho_0} \sim [\![t_1]\!]_{\rho_1} \in R$ for some R.
(3) $\Gamma \vDash \delta_0 = \delta_1 : \Delta$ holds when for all $p : \mathbb{P}$ and $p \Vdash \rho_0 = \rho_1 : \Gamma$, we have $p \Vdash [\![\delta_0]\!]_{\rho_0} = [\![\delta_1]\!]_{\rho_1} : \Delta$
(4) $\Gamma \vDash A$ *type* holds iff $\Gamma \vDash A = A$ *type*.
(5) $\Gamma \vDash t : A$ holds iff $\Gamma \vDash t = t : A$.
(6) $\Gamma \vDash \delta : \Delta$ holds iff $\Gamma \vDash \delta = \delta : \Delta$.

The fundamental theorem of the PER model is to show that τ_ω is closed under all the rules of MLTT$_{\text{🔒}}$ in the sense of Theorem 5.1 below.

Theorem 5.1 (Fundamental Theorem). *If* $\Gamma \vdash \mathcal{J}$, *then* $\Gamma \vDash \mathcal{J}$.

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{J}$. □

Remark 5.2. Explicit substitutions play an important role in the proof of Theorem 5.1; without them, this model would refute many β-equalities![4] Consider the β-rule for functions $(\lambda(t_0))(t_1) = t_0[\mathrm{id}.t_1]$. In our type theory with explicit substitutions, in the environment ρ both of these will

[4]The authors would like to acknowledge that Abel explained this point in 2013, but that they unwisely chose to ignore it in a first attempt at this proof.

compute to $[\![t_0]\!]_{\rho.[\![t_1]\!]_\rho}$. In particular, in evaluating $t_0[\mathsf{id}.t_1]$, we first use the explicit substitution to modify the environment: $[\![\mathsf{id}.t_1]\!]_\rho = \rho.[\![t_1]\!]_\rho$.

In contrast, if substitution were a meta-operation, we would only have that the right-hand side of the equation computes as t_0', where t_0' is the result of substituting t_1' for var_0 in t_0. In a defunctionalized NbE algorithm, these will not evaluate to the same result. We may see the difference if there are any closures in the results of the evaluations: in the left-hand side the substitution will not have taken place under any closures, but in the right-hand side the substitution will have propagated through.

Completeness of normalization for definitional equality is a corollary of the fundamental theorem of the PER model, using the fact that τ_ω is valued in saturated $\mathbb{P}$-PERs. We can show for any Γ *ctx*, $p \Vdash \uparrow\Gamma = \uparrow\Gamma : \Gamma$. Therefore, if we are have $\Gamma \vdash t_0 = t_1 : A$, we then must have $p \Vdash [\![t_0]\!]_{\uparrow\Gamma} \sim [\![t_1]\!]_{\uparrow\Gamma} \in [\![A]\!]_{\uparrow\Gamma}$ by the fundamental theorem. Finally, the saturation of $[\![A]\!]_\rho$ then tells us that the quotations of $[\![t_0]\!]_{\uparrow\Gamma}, [\![t_1]\!]_{\uparrow\Gamma}$ are identical, whereby $\underline{\mathbf{nbe}}^A_\Gamma(t_0) = \underline{\mathbf{nbe}}^A_\Gamma(t_1)$.

6 THE SOUNDNESS OF NORMALIZATION BY EVALUATION

Through completeness, we have shown that the normalization algorithm lifts to a total function on definitional equivalence–classes of MLTT$_{🔒}$ terms; but even the constant function which returns the same "normal form" for all terms would have this property. We additionally need to see that normalization is faithful, or *sound*:

(1) If $\Gamma \vdash A$ *type* and $\underline{\mathbf{nbe}}^{\mathsf{tp}}_\Gamma(A) = A'$ then $\Gamma \vdash A = A'$ *type*.
(2) If $\Gamma \vdash t : A$ and $\underline{\mathbf{nbe}}^A_\Gamma(t) = t'$ then $\Gamma \vdash t = t' : A$.

The soundness of normalization for definitional equality, like completeness, cannot be proved naïvely by induction on derivations. Instead, it is necessary to employ a further model construction which glues the syntax of MLTT$_{🔒}$ together with its computational model; this is a *cross-language Kripke logical relation* between syntax and semantics, indexed in the category of syntactic contexts and weakenings. The fundamental judgments of the logical relations model are the following:

(1) $\Gamma \vdash_p A \mathrel{®} A' \ \mathsf{type}_\alpha$ relates a syntactic type $\Gamma \vdash A$ *type* in universe level α to the semantic type value A' at stage p.
(2) $\Gamma \vdash_p t : A \mathrel{®} v \in_\alpha A'$ relates a syntactic term $\Gamma \vdash t : A$ to the semantic value v in the type value A' in universe level α at stage p.
(3) $\Gamma \vdash_p \delta : \Delta \mathrel{®} \rho$ relates a syntactic substitution $\Gamma \vdash \delta : \Delta$ to a semantic environment ρ at stage p.

Saturation condition. The definition of these logical relations is somewhat technical (see Figure 9), but the main objective is to ensure that they all exhibit the following saturation conditions (from which soundness will follow):

(1) If $\Gamma \vdash_p A \mathrel{®} A' \ \mathsf{type}_\alpha$, then for any weakening substitution $\Delta \vdash \gamma : \Gamma$, we have the equation $\Delta \vdash A[\gamma] = \lceil A' \rceil^{\mathsf{ty}}_{\|\Delta\|}$ *type*.
(2) If $\Gamma \vdash_p t : A \mathrel{®} v \in_\alpha A'$, then for any weakening substitution $\Delta \vdash \gamma : \Gamma$, we have the equation $\Delta \vdash t[\gamma] = \lceil \downarrow^{A'} v \rceil_{\|\Delta\|} : A[\gamma]$
(3) If $\Gamma \vdash_p A \mathrel{®} A' \ \mathsf{type}_\alpha$ and $\Gamma \vdash t : A$ and for all weakening substitutions $\Delta \vdash \gamma : \Gamma$ we have $\Delta \vdash t[\gamma] = \lceil e \rceil_{\|\Delta\|} : T[\gamma]$, then $\Gamma \vdash_p t : A \mathrel{®} \uparrow^{A'} e \in_\alpha A'$.
(4) We furthermore require that the identity substitution is related to the reflection of its context, $\Gamma \vdash_p \mathsf{id} : \Gamma \mathrel{®} \uparrow\Gamma$.

The fundamental theorem. After defining the logical relations and showing that they are saturated, we show that they interpret the rules of MLTT$_{🔒}$ in a suitable way:

(1) If $\Gamma \vdash A\ \mathit{type}$, then for all $p : \mathbb{P}$ and $\Delta \vdash_p \gamma : \Gamma \mathbin{®} \rho$, we have $\Delta \vdash_p A[\gamma] \mathbin{®} [\![A]\!]_\rho\ \mathrm{type}_\omega$.
(2) If $\Gamma \vdash t : A$, then for all $p : \mathbb{P}$ and $\Delta \vdash_p \gamma : \Gamma \mathbin{®} \rho$, we have $\Delta \vdash_p t[\gamma] : A[\gamma] \mathbin{®} [\![t]\!]_\rho \in_\omega [\![A]\!]_\rho$.

The soundness of normalization follows immediately from the fundamental theorem of the logical relations model, using the fact that every logical relation is saturated: if $\Gamma \vdash A\ \mathit{type}$, then (picking arbitrary $p : \mathbb{P}$) we have $\Gamma \vdash_p A \mathbin{®} [\![A]\!]_{\uparrow\Gamma}\ \mathrm{type}_\omega$ (using the identity weakening); by saturation, we furthermore have $\Gamma \vdash A = \ulcorner [\![A]\!]_{\uparrow\Gamma} \urcorner^{\mathsf{ty}}_{\|\Gamma\|}\ \mathit{type}$, and by definition we have $\underline{\mathbf{nbe}}^{\mathsf{tp}}_\Gamma(A) = \ulcorner [\![A]\!]_{\uparrow\Gamma} \urcorner^{\mathsf{ty}}_{\|\Gamma\|}$

Constructing the logical relations. We can explicitly construct a hierarchy of Kripke logical relations which has the properties described in the preceeding paragraphs, but it is somewhat subtle. We need to define $\Gamma \vdash_p A \mathbin{®} A'\ \mathrm{type}_\alpha$ by induction on the value A', but the induction is not obviously structural. For instance, we intend that $\Gamma \vdash_p C \mathbin{®} \mathbf{\Pi}(A, B)\ \mathrm{type}_\alpha$ shall hold iff the following hold:

- $\Gamma \vdash C = \Pi(A', B')\ \mathit{type}$ for some A', B';
- $\Gamma \vdash_p A' \mathbin{®} A\ \mathrm{type}_\alpha$;
- if $q \le p$ and $\Delta \vdash \gamma : \Gamma$ is a weakening, then $\Delta \vdash_q t : A'[\gamma] \mathbin{®} v \in_\alpha A$ implies $\Delta \vdash_q B'[r.t] \mathbin{®} B[v]\ \mathrm{type}_\alpha$.

The problem lies in the final clause above: the closure instantiation $B[v]$ is not structurally smaller than the semantic type $\mathbf{\Pi}(A, B)$. To resolve this problem, we define a well-ordering on semantic types $\sigma \models_p A < B$ relative to a Kripke type system σ and stage $p : \mathbb{P}$, in which (for instance) a dependent function is strictly larger than all well-typed instantiations of its closure; then, the definition of the logical relations can proceed by well-founded induction. This well-founded ordering is latent in Wieczorek and Biernacki [2018]. We exhibit a fragment of this definition in Figure 9.

Remark 6.1. The logical relation for soundness is by far the subtlest portion of the normalization proof, and one particularly slippery detail is the injectivity of type-constructors. During the course of the proof of the fundamental lemma for this logical relation, it is not known whether $\Gamma \vdash \Sigma(A_0, B_0) = \Sigma(A_1, B_1)\ \mathit{type}$ implies that $\Gamma \vdash A_0 = A_1\ \mathit{type}$ and $\Gamma.A_0 \vdash B_0 = B_1\ \mathit{type}$ (indeed, this is commonly proved as a corollary of normalization). This fact, however, seems needed during the proof of the fundamental lemma. This particular Gordian knot is cut through the extra premises placed on elimination rules in the declarative syntax (for instance, the requirement of $\Gamma.A \vdash B\ \mathit{type}$ in TM/SND). These premises give us a sufficiently strong induction hypothesis to push the proof through, and their redundancy can then be observed after the fact.

7 THE CORRECTNESS OF SEMANTIC TYPE-CHECKING

We wish to show that our semantic type-checking algorithm is equivalent to the declarative system for which we have proven NbE sound and complete. It is not immediately clear how to formulate this statement, however, because the semantic type-checking algorithm operates on terms of $\mathsf{MLTT}^{\leftrightarrows}_{\text{🔒}}$, not $\mathsf{MLTT}_{\text{🔒}}$. Instead, we prove an *adequacy* theorem for $\mathsf{MLTT}^{\leftrightarrows}_{\text{🔒}}$: every typeable term in the declarative system is equal (in the declarative system) to a term which is well-formed in the bidirectional syntax and appropriately typed by our algorithm.

Theorem 7.1 (Adequacy: soundness). *If $\Gamma \vdash A\ \mathit{type}$ and $\Updownarrow\Gamma \vdash M \Leftarrow [\![A]\!]_{\uparrow\Gamma}$, then $\Gamma \vdash M^\circ : A$.*

A crucial difficulty in showing that semantic type-checking is complete as well as sound is the conversion rule. In the declarative system any type could be replaced with an equal one during the process of type-checking but the semantic type-checking algorithm is far more rigid. The type-checking algorithm, however, is stable under the PER equality defined in Section 5.

Lemma 7.2. *If $\tau_\omega \models_p A \sim B$ and $\Xi \vdash M \Leftarrow A$, then $\Xi \vdash M \Leftarrow B$.*

- $\Gamma \vdash_p C \mathrel{®} \mathbf{\Pi}(A, B)\ \mathrm{type}_\alpha$ if:
  - $\Gamma \vdash C = \Pi(A', B')$ *type* for some A', B';
  - $\Gamma \vdash_p A' \mathrel{®} A\ \mathrm{type}_\alpha$;
  - if $q \le p$ and $\Delta \vdash \gamma : \Gamma$ is a weakening, then $\Delta \vdash_q t : A'[\gamma] \mathrel{®} v \in_\alpha A$ implies $\Delta \vdash_q B'[r.t] \mathrel{®} B[v]\ \mathrm{type}_\alpha$.
- $\Gamma \vdash_p C \mathrel{®} \Box A\ \mathrm{type}_\alpha$ if:
  - $\Gamma \vdash C = \Box A'$ *type* for some A';
  - for all q, $\Gamma.🔒 \vdash_q A' \mathrel{®} A\ \mathrm{type}_\alpha$.
- $\Gamma \vdash_p C \mathrel{®} \uparrow^A e\ \mathrm{type}_\alpha$ if, when $\Delta \vdash \gamma : \Gamma$ is a weakening, then $\Delta \vdash C[\gamma] = \lceil e \rceil_{\|\Delta\|}$ *type*.
- $\Gamma \vdash_p C \mathrel{®} \mathbf{U}_j\ \mathrm{type}_\alpha$ if $j < \alpha$ and $\Gamma \vdash C = \mathrm{U}_j$ *type*.

- $\Gamma \vdash_p t : C \mathrel{®} v \in_\alpha \mathbf{\Pi}(A, B)$ if:
  - $p \Vdash v \sim v \in R$ and $\Gamma \vdash t : C$;
  - $\Gamma \vdash C = \Pi(A', B')$ *type* for some A', B';
  - $\Gamma \vdash_p A' \mathrel{®} A\ \mathrm{type}_\alpha$;
  - if $q \le n$ and $\Delta \vdash \gamma : \Gamma$ is a weakening, then $\Delta \vdash_q s : A'[\gamma] \mathrel{®} u \in_\alpha A$ implies $\Delta \vdash_q t[\gamma](s) : B'[\gamma.s] \mathrel{®} \underline{\mathbf{app}}(v, u) \in_\alpha B[u]$.
- $\Gamma \vdash_p t : C \mathrel{®} v \in_\alpha \Box A$ if:
  - $p \Vdash v \sim v \in R$ and $\Gamma \vdash t : C$;
  - $\Gamma \vdash C = \Box A'$ *type* for some A';
  - for all q, $\Gamma.🔒 \vdash_q [t]_🔓 : A' \mathrel{®} \underline{\mathbf{unlock}}(v) \in_\alpha A$
- $\Gamma \vdash_p t : C \mathrel{®} \uparrow^{A_0} e_0 \in_\alpha \uparrow^{A_1} e_1$ if, when $\Delta \vdash \gamma : \Gamma$ is a weakening, then $\Delta \vdash C[\gamma] = \lceil e_1 \rceil_{\|\Delta\|}$ *type* and $\Delta \vdash t[\gamma] = \lceil e_0 \rceil_{\|\Delta\|} : C[\gamma]$.
- $\Gamma \vdash_p t : C \mathrel{®} v \in_\alpha \mathbf{U}_i$ if:
  - $i < \alpha$;
  - $p \Vdash v \sim v \in R$;
  - $\Gamma \vdash t : C$ and $\Gamma \vdash C = \mathrm{U}_i$ *type*;
  - $\Gamma \vdash_p t \mathrel{®} v\ \mathrm{type}_i$.

Fig. 9. A fragment of the definition of the logical relations for types and terms.

This lemma in turn ensures that the semantic type-checking algorithm is complete for the conversion rule of the declarative syntax; by completeness, if $\Gamma \vdash A = B$ *type* holds then $\tau_\omega \models_p [\![A]\!]_{\uparrow\Gamma} \sim [\![B]\!]_{\uparrow\Gamma}$. But Lemma 7.2 then tells us that if $\Xi \vdash M \Leftarrow [\![A]\!]_{\uparrow\Gamma}$, then $\Xi \vdash M \Leftarrow [\![B]\!]_{\uparrow\Gamma}$, precisely as the conversion rule would require.

Stability relies on the fact that the semantic type-checking algorithm inspects only the outermost constructor of the value when checking a term against a type – no equal semantic types have different head constructors so it is safe to inspect these.

It is now possible to prove the completeness of the type-checking algorithm.

Theorem 7.3 (Adequacy: completeness). *If $\Gamma \vdash A$ type and $\Gamma \vdash t : A$, then there exists a bidirectional term M such that $\Gamma \vdash M^\circ = t : A$ and $\Updownarrow\Gamma \vdash M \Leftarrow [\![A]\!]_{\uparrow\Gamma}$.*

Proof. By soundness and completeness of normalization, we have $\Gamma \vdash t = \underline{\mathbf{nbe}}^A_\Gamma(t) : A$; but the declarative terms and the checkable terms coincide for normal forms, so we simply choose $M \triangleq \underline{\mathbf{nbe}}^A_\Gamma(t)$. □

8 RELATED WORK

Many previous variants of modal simply-type calculi have been presented. One notable such calculus is Pfenning and Davies [2000], which structures the judgments differently than $\mathsf{MLTT}_{\color{blue}\blacksquare}$. In Pfenning and Davies, contexts are split into *true* and *necessary* hypotheses.

Cohesive type theory. The dual context approach is used in, for instance, Shulman's [2018] *cohesive type theory*, another proposed modal dependent type theory. Cohesive type theory has focused on providing a type theory for abstract spaces [Lawvere 1992, 2007] using a chain of interacting modalities: $\int \dashv \flat \dashv \sharp$. Indeed, cohesive type theory has been successfully used on paper to prove Brouwer's fixed point theorem [Shulman 2018] within type theory and without recourse to low-level manipulations of topological spaces.

There is, however, enormous syntactic complexity that results from the non-trivial interactions of multiple modalities. In particular, the $\flat$ modality (which corresponds to $\Box$) uses an *open-scope* or *positive* eliminator. This prevents cohesive type theory from adding all of the equations of the $\flat$ modality without introducing *commuting conversions*,[5] which render decision procedures for definitional equality intractable. There is ongoing work to study syntactic properties of systems like cohesive type theory with multiple modalities; so far, however, this work has focused on restricted, simply-typed instances [Licata et al. 2017].

Agda-flat and crisp type theory. Separately, *crisp type theory* (a fragment of cohesive type theory which contains only the $\flat$ modality) has been implemented experimentally in a fork of *Agda* [The Agda Development Team 2018]. Crisp type theory is close to our own type theory, supporting a single **S4**-style comonadic modality. Unlike $\mathsf{MLTT}_{\color{blue}\blacksquare}$, crisp type theory, like cohesive type theory, uses an positive eliminator for its modality, and so it fails to satisfy several definitional equations that $\mathsf{MLTT}_{\color{blue}\blacksquare}$ enjoys. Additionally, while there is an experimental implementation of crisp type theory, there has not been work on proving any metatheoretic properties of the system, and, in particular, there is no proof of correctness for the implementation.

Contextual modal type theory. Another dual-context modal type theory is contextual modal type theory (CMTT) [Boespflug and Pientka 2011; Brottveit Bock and Schürmann 2015; Nanevski et al. 2008; Pientka et al. 2019]. CMTT has been studied in the context of *logical frameworks*, type theories specifically designed to study other type theories. Contextual modalities allow a type theory to internally specify that a term depends on a designated set of local variables. This generalizes the necessity modality, where a term can depend on either any local variables or none. CMTT provides an ideal setting for reasoning about higher-order abstract syntax, which demands the ability to manipulate open terms as first-class objects.

Generalizing previous work on extending logical frameworks with contextual modalities, Pientka et al. [2019] have developed a version of full Martin-Löf Type Theory equipped with a contextual modality ranging over the contexts and types of a substrate logical framework. This work promises to bridge the gap between the convenience of LF-style mechanized metatheory [Harper and Licata 2007] and the generality available in modern proof assistants such as Agda [Norell 2007] and Coq [Coq Development Team 2016], a long-standing goal within the community.

Visible on the horizon is a type theory which collapses the distinction between *object* and *meta*, thereby obtaining the ability to nest the contextual modality multiple times. The modality considered in this paper would then arise as a special case.

[5] In some (homotopy-theoretic) models of cohesive type theory, one expects these commuting conversions to hold only up to a path. They do, however, hold strictly in 1-categorical models.

Guarded recursion. Recent investigations of modal dependent type theory have focused on guarded recursion [Bahr et al. 2017; Birkedal et al. 2016; Bizjak et al. 2016; Bizjak and Møgelberg 2015; Mannaa and Møgelberg 2018]. While the semantics of guarded recursion is relatively well-understood, the syntax presents certain difficulties pertaining to the admissibility of substitution. Early work in this area has resolved this problem using *delayed substitutions*, whose complicated equational theory obstructs the development of the syntactic metatheory, including normalization and decidability of type-checking.

Clocked Type Theory [Bahr et al. 2017; Mannaa and Møgelberg 2018] pioneered the use of *Fitch-style* syntax for modalities in a dependently typed setting, generalizing the work of Clouston [2018] from simple types and avoiding the use of delayed substitutions. In light of the improved syntax, Bahr et al. [2017] have been able to equip Clocked Type Theory with an operational semantics (a significant advance over prior work) and proved that it enjoys strong normalization. The sophistication of Clocked Type Theory's modal apparatus seemingly necessitates rules which invert a substitution, presenting certain challenges in passing from a strong normalization result to an algorithm for checking types.

Dependent right adjoints. Expanding of the ideas of Clocked Type Theory, Clouston et al. [2018] abstracted the particulars of guarded recursion into a general discipline for dependently typed modalities, generalizing the Fitch-style calculus [Clouston 2018] to a full dependent type theory.

MLTT$_🔒$ strengthens the syntactic properties of the modality in *loc. cit.* and, additionally, simplifies the rules of the type theory. As a result of these simplifications, our type theory validates the appropriate admissible rules and supports normalization. This was crucial for producing an implementation.

9 FUTURE WORK AND CONCLUSIONS

We have contributed MLTT$_🔒$, a core calculus for a dependently typed programming language with a necessity modality, together with a sound and complete type checking algorithm based on normalization by evaluation. To demonstrate that the MLTT$_🔒$ approach is ready for real-world applications, we have implemented a prototype proof assistant based on the calculus and algorithms which we have proved correct here. The core of the implementation is just 500 lines of code and transcribes the rules almost directly.

Categorical semantics. In this paper we have focused on syntactic properties (*e.g.,* normalization and the decidability of type-checking) and so we have not given much consideration to the categorical semantics of MLTT$_🔒$. In the future, we hope to specialize the semantics described in Clouston et al. [2018] to our modality. Unlike *loc. cit.* we will require stronger conditions on our dependent right adjoint. For instance, we will certainly need to require that it is an idempotent comonad. In addition, we will require the left adjoint to form a monad. With these modifications, we believe that the theory of Clouston et al. can be used to describe the categorical models of MLTT$_🔒$.

Extending to multiple modalities. A significant challenge for the future is the extension of MLTT$_🔒$ with more than one modality; different modalities have different effects on the proper design of substitution principles and structural rules, and these effects are not a priori local. Negotiating the emergent interactions between different modalities in a dependently typed setting is a critical area of future research that we hope to pursue; in the much more restricted simply typed setting, progress has been made by Licata et al. [2017] to this end. This line of work is important for incorporating existing modal type theories into our framework; multiple interacting modalities are crucial, for instance, in guarded type theory.

Semantic normalization. Normalization by evaluation, which we have presented in a highly syntactic way, corresponds to an instance of the *categorical gluing* technique [Altenkirch et al. 1995; Coquand 2018; Fiore 2002; Streicher 1998]. In semantic proofs of normalization, one glues a category of Kripke predicates along the nerve induced by a subcategory of substitutions which normal forms are closed under.

Instead of proving that a concretely-given normalization algorithm is correct through a PER model and logical relations (see Sections 5 and 6), these semantic proofs employ a single model and, using the *initiality of syntax*, induce a normalization algorithm abstractly. There has already been considerable work in Clouston et al. [2018] on the structure of models of modal dependent type theory, so we conjecture that the proof of normalization for $\mathsf{MLTT}_{\color{blue}\blacksquare}$ could be streamlined by adapting categorical gluing to our situation. It remains to be seen, however, what changes are required to the syntactic presentation of $\mathsf{MLTT}_{\color{blue}\blacksquare}$ in order to connect the abstract (algebra) with the concrete (implementation).

Applications in mathematics. A current goal within the scientific community is to maximize the amount of mathematics which can be formalized synthetically inside type theory [Shulman 2018; Univalent Foundations Program 2013], avoiding low-level analytic details. A recent success story involving the necessity modality in dependent type theory can be found in the *internal* construction of classifying objects for fibrations in models of homotopy type theory based on a "tiny interval" [Angiuli et al. 2019; Licata et al. 2018]. This construction uses a right adjoint to the path space functor $(-)^{\mathbb{I}}$ in a critical way, but this functor cannot be internalized. Using the necessity modality of $\mathsf{MLTT}_{\color{blue}\blacksquare}$, it is easy to axiomatize this adjunction and use it to construct the classifying fibration. Licata et al. [2018] have presented a formalization in *agda-flat*, and we conjecture that it should be possible to formalize the same in $\mathsf{MLTT}_{\color{blue}\blacksquare}$. We hope that such a formalization may benefit from the additional definitional equalities enabled by our treatment of the modality.

More generally, the necessity modality of $\mathsf{MLTT}_{\color{blue}\blacksquare}$ enables the type-theoretic axiomatization of global operations (such as comonadic modalities) which are not closed under substitution. This technique makes it possible to formalize a great deal of mathematics inside type theory, and the implementability of $\mathsf{MLTT}_{\color{blue}\blacksquare}$ promises the benefits of machine-checking for these formalizations.

ACKNOWLEDGMENTS

We are very thankful for productive conversations with Andreas Abel, Carlo Angiuli, Ranald Clouston, Robert Harper, Daniel R. Licata, Rasmus Ejlers Møgelberg, Brigitte Pientka, Urs Schreiber, Bas Spitters, and Felix Wellen. We also gratefully acknowledge our reviewers for their insightful comments, suggestions, and questions.

The authors gratefully acknowledge the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU) and in part by a research grant (12386, Guarded Homotopy Type Theory) from VILLUM FONDEN.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 147–160. https://doi.org/10.1145/292540.292555

Andreas Abel. 2009. Extensional normalization in the logical framework with proof irrelevant equality. In *2009 Workshop on Normalization by Evaluation.*

Andreas Abel. 2013. Normalization by Evaluation: Dependent Types and Impredicativity.

Andreas Abel, Klaus Aehlig, and Peter Dybjer. 2007. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. *Electron. Notes Theor. Comput. Sci.* 173 (April 2007), 17–39. https://doi.org/10.1016/j.entcs.2007.02.025

Andreas Abel, Thierry Coquand, and Miguel Pagano. 2009. A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance. In *Typed Lambda Calculi and Applications*, Pierre-Louis Curien (Ed.). Springer Berlin Heidelberg, 5–19.

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

Stuart Frazier Allen. 1987. A non-type-theoretic semantics for type-theoretic language.

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, David Pitt, David E. Rydeheard, and Peter Johnstone (Eds.). Springer Berlin Heidelberg, 182–199.

Carlo Angiuli. 2019. *Computational Semantics of Cartesian Cubical Type Theory.* Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA. To appear.

Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. 2019. Cartesian Cubical Type Theory. (Feb. 2019). https://github.com/dlicata335/cart-cube Preprint.

P. Bahr, H. B. Grathwohl, and R. E. Møgelberg. 2017. The clocks are ticking: No more delays!. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–12.

Ulrich Berger and H Schwichtenberg. 1991. An inverse of the evaluation functional for typed λ-calculus. *Proceedings - Symposium on Logic in Computer Science*, 203–211. https://doi.org/10.1109/LICS.1991.151645

Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Jean-Marc Talbot and Laurent Regnier (Eds.), Vol. 62. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 23:1–23:17.

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Stovring. 2011. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. IEEE Computer Society, 55–64.

Aleš Bizjak and Lars Birkedal. 2018. On Models of Higher-Order Separation Logic. *Electr. Notes Theor. Comput. Sci.* 336 (2018), 57–78. https://doi.org/10.1016/j.entcs.2018.03.016

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures: 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, 20–35.

Aleš Bizjak and Rasmus Ejlers Møgelberg. 2015. A Model of Guarded Recursion With Clock Synchronisation. *Electron. Notes Theor. Comput. Sci.* 319, C (Dec. 2015), 83–101.

Mathieu Boespflug and Brigitte Pientka. 2011. Multi-level Contextual Type Theory. *Electronic Proceedings in Theoretical Computer Science* 71 (Oct. 2011). https://doi.org/10.4204/EPTCS.71.3

V. A. J. Borghuis. 1994. Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus. https://doi.org/10.6100/IR427575

Peter Brottveit Bock and Carsten Schürmann. 2015. A Contextual Logical Framework, Vol. 9450. 402–417. https://doi.org/10.1007/978-3-662-48899-7_28

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, 258–275.

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, 407–421.

Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2018. Modal Dependent Type Theory and Dependent Right Adjoints. (2018). https://arxiv.org/abs/1804.05236

The Coq Development Team. 2016. The Coq Proof Assistant Reference Manual.

Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1 (1996), 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

Thierry Coquand. 2018. Canonicity and normalization for Dependent Type Theory. https://arxiv.org/abs/1810.09367

Rowan Davies and Frank Pfenning. 1999. A Modal Analysis of Staged Computation. *J. ACM* 48 (Sept. 1999). https://doi.org/10.1145/382780.382785

Jeff Epstein, Andrew Black, and Simon Peyton Jones. 2011. Towards Haskell in the cloud. https://www.microsoft.com/en-us/research/publication/towards-haskell-cloud/

Marcelo Fiore. 2002. Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '02)*. ACM, 26–37. https://doi.org/10.1145/571157.571161

Peter Freyd. 1991. Algebraically complete categories. In *Category Theory*, Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini (Eds.). Springer Berlin Heidelberg, 95–104.

Johan G. Granström. 2013. *Treatise on Intuitionistic Type Theory*. Springer Publishing Company, Incorporated.

Adrien Guatto. 2018. A Generalized Modality for Recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. 482–491. https://doi.org/10.1145/3209108.3209148

Robert Harper and Daniel R. Licata. 2007. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming* 17, 4-5 (July 2007), 613–673. https://doi.org/10.1017/S0956796807006430

G. A. Kavvos. 2017. Dual-Context Calculi for Modal Logic. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. http://arxiv.org/abs/1602.04860

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *European Symposium on Programming*.

F. William Lawvere. 1992. Categories of Space and of Quantity. In *The Space of Mathematics*, Javier Echeverria, Andoni Ibarra, and Thomas Mormann (Eds.). De Gruyter, 14–30.

F. William Lawvere. 2007. Axiomatic Cohesion. *Theory and Applications of Categories* 19 (June 2007).

Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. 22:1–22:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.22

Daniel R. Licata, Michael Shulman, and Mitchell Riley. 2017. A Fibrational Framework for Substructural and Modal Logics. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Dale Miller (Ed.), Vol. 84. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 25:1–25:22. https://doi.org/10.4230/LIPIcs.FSCD.2017.25

Bassel Mannaa and Rasmus Ejlers Møgelberg. 2018. The Clocks They Are Adjunctions Denotational Semantics for Clocked Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Hélène Kirchner (Ed.), Vol. 108. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23:1–23:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.23

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

Per Martin-Löf. 1992. Substitution calculus. Notes from a lecture given in Göteborg.

Per Martin-Löf. 1996. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic* 1, 1 (1996), 11–60.

Simone Martini and Andrea Masini. 1996. *A Computational Interpretation of Modal Proofs*. Springer Netherlands, Dordrecht, 213–241. https://doi.org/10.1007/978-94-017-2798-3_12

Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13. https://doi.org/10.1017/S0956796807006326

Tom Murphy, VII. 2008. Modal Types for Mobile Code. http://tom7.org/papers/ Available as technical report CMU-CS-08-126.

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. 2004. A Symmetric Modal Lambda Calculus for Distributed Computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Press.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions Computational Logic* 9 (June 2008). https://doi.org/10.1145/1352582.1352591

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology.

Frank Pfenning and Rowan Davies. 2000. A Judgmental Reconstruction of Modal Logic. *Mathematical Structures in Computer Science* 11 (Feb. 2000). https://doi.org/10.1017/S0960129501003322

Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rébecca Zucchini. 2019. A Type Theory for Defining Logics and Proofs. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*.

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions Programming Language and Systems* 22, 1 (2000), 1–44.

Dag Prawitz. 1967. Natural Deduction. A Proof-Theoretical Study. *Journal of Symbolic Logic* 32, 2 (1967), 255–256.

Urs Schreiber. 2013. Differential cohomology in a cohesive infinity-topos. *arXiv e-prints*, Article arXiv:1310.7930 (Oct 2013), arXiv:1310.7930 pages. arXiv:math-ph/1310.7930

Urs Schreiber and Michael Shulman. 2014. Quantum Gauge Field Theory in Cohesive Homotopy Type Theory. In *Proceedings 9th Workshop on Quantum Physics and Logic Brussels, Belgium, 10-12 October 2012*. 109–126. https://doi.org/10.4204/EPTCS.158.8

Peter Schroeder-Heister. 1987. Structural Frameworks with Higher-level Rules: Philosophical Investigations on the Foundations of Formal Reasoning. Habilitation thesis.

Michael Shulman. 2018. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science* 28, 6 (2018), 856–941. https://doi.org/10.1017/S0960129517000147

Thomas Streicher. 1998. Categorical intuitions underlying semantic normalisation proofs. In *Preliminary Proceedings of the APPSEM Workshop on Normalisation by Evaluation*, O. Danvy and P. Dybjer (Eds.). Department of Computer Science, Aarhus University.

The Agda Development Team. 2018. `agda-flat`. https://github.com/agda/agda/tree/flat

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book.

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