

# A Dependent Type Theory for Meta-programming with Intensional Analysis

JASON Z. S. HU, McGill University, Canada

BRIGITTE PIENKA, McGill University, Canada

In this paper, we introduce DELAM, a dependent layered modal type theory which enables meta-programming in Martin-Löf type theory (MLTT) with recursion principles on open code. DELAM includes three layers: the layer of static syntax objects of MLTT without any computation, the layer of pure MLTT with the computational behaviors, and the meta-programming layer, which extends MLTT with support for quoting an open MLTT code object, composing, and analyzing open code using recursion. We can also execute a code object at the meta-programming layer. The expressive power strictly increases as we move up in a given layer. In particular, while code objects only describe static syntax, we allow computation at the MLTT and meta-programming layer. As a result, DELAM provides a dependently typed foundation for meta-programming that supports both type-safe code generation and code analysis. We prove the weak normalization of DELAM and the decidability of convertibility using Kripke logical relations.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → **Functional languages**.

Additional Key Words and Phrases: dependent types, logical relations, meta-programming, intensional analysis, modal type theory, contextual types

## ACM Reference Format:

Jason Z. S. Hu and Brigitte Pientka. 2025. A Dependent Type Theory for Meta-programming with Intensional Analysis. *Proc. ACM Program. Lang.* 9, POPL, Article 15 (January 2025), 30 pages. <https://doi.org/10.1145/3704851>

## 1 Introduction

Today, many proof assistants (PAs) support meta-programming in practice (e.g. Agda [van der Walt and Swierstra 2012], Idris [Christiansen and Brady 2016], MetaCoq [Anand et al. 2018; Sozeau et al. 2020], or Lean [Ebner et al. 2017]). This is done by reflecting the syntax of the type theory in the given PA, so that users can directly write and reason about meta-programs that construct and manipulate the syntactic objects in the dependent type theory itself. However, the reflected syntactic object only provides access to an untyped, low-level syntax tree, where variables are modeled by de Bruijn indices. Therefore, meta-programs generally do not guarantee the well-scopedness or the well-formedness of these syntax trees. As a consequence, errors in meta-programs can only be found when evaluating the syntactic objects for execution rather than when generating them. This is unfortunate, since the PA itself has the capability to express stronger properties.

To provide stronger static guarantees for meta-programs, Mtac [Kaiser et al. 2018; Ziliani et al. 2015] extends Coq with a monadic type  $M_A$ , representing the type of Mtac tactics that, when applied, may diverge or fail, but if they terminate successfully, will produce a Coq term of type  $A$ . Further, to support practical tactic development, Mtac provides support for capturing the proof

---

Authors' Contact Information: Jason Z. S. Hu, McGill University, Montréal, Canada, zhong.s.hu@mail.mcgill.ca; Brigitte Pientka, McGill University, Montréal, Canada, brigitte.pientka@mcgill.ca.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/1-ART15  
<https://doi.org/10.1145/3704851>

state and analyzing it via pattern matching. While this approach provides more static guarantees than reflection, it remains unclear how this extension fits within Coq's type theory, and it violates important type-theoretic properties such as confluence and normalization.

Despite the intense engineering effort invested in meta-programming in PAs, these tools do not have type-theoretic foundations. In this paper, we are interested in a more fundamental question:

**What is a suitable dependent type theory for meta-programming that allows us to quote objects as code, compose code, recursively analyze code, and evaluate code?**

We take as a starting point the work by [Davies and Pfenning \[2001\]](#); [Pfenning and Davies \[2001\]](#) which provides a simply-typed foundation for meta-programming based on the modal logic  $S4$ . In this work, the necessity modality  $\Box T$  is interpreted as closed code of type  $T$ . Meta-programs are then programs that generate objects of type  $\Box T$ . Subsequently, [Nanevski et al. \[2008\]](#) generalize this work to capture open code using the contextual type  $(\Box(\Gamma \vdash T))$  that describes open code of type  $T$  in a context  $\Gamma$ . For example, the *code object* `box`  $(x. x + 2)$  has the contextual type  $\Box(x:\text{Nat} \vdash \text{Nat})$ . This line of work has been, for example, extended to System F [\[Jang et al. 2022\]](#), but it has been challenging to extend this work to a dependent type theory that supports intensional code analysis.

In this paper, we describe DELAM, a **Dependent Layered Modal** type theory, which enables meta-programming in Martin-Löf type theory (MLTT) with recursion principles on open code. Following [Hu and Pientka \[2024b\]](#), DELAM exploits *the matryoshka principle*: sub-languages at higher layers subsume those at lower layers. The principle is formally captured by two *guiding lemmas*: even though *the static code lemma* proves that code objects do not compute, via *the lifting lemma*, code objects may compute after being lifted to higher layers.

Specifically, DELAM includes three layers: code objects ( $\mathcal{C}$ ), the dependent type theory of MLTT ( $\mathcal{D}$ ), and meta-programs ( $\mathcal{M}$ ), such that  $\mathcal{C} < \mathcal{D} < \mathcal{M}$ . Syntactically, the layer  $\mathcal{C}$  describes static code objects of MLTT with no computation; contrarily, the layer  $\mathcal{D}$  corresponds to the dependent type theory of pure MLTT and allows computation. The meta-programming layer  $\mathcal{M}$  extends MLTT with contextual types, code objects and recursion principles over them. Hence the expressive power of sub-languages strictly increases as we move up one layer, as well as the computational power. As proved by the static code lemma, code objects at layer  $\mathcal{C}$  are static and do not compute, but their types do and live at the higher layer  $\mathcal{D}$ . To illustrate, consider the constructor for an empty list `nil`  $: (A : \mathbf{Ty}) \rightarrow \text{List } A$ , where  $\mathbf{Ty}$  is the type for the universe. Then `box`  $(\text{nil } ((\lambda x. x) \text{Nat}))$  and `box`  $(\text{nil } \text{Nat})$  represent two distinct code objects at layer  $\mathcal{C}$ , so they are not equivalent. In particular, the  $\beta$  redex in the first code object does not compute at layer  $\mathcal{C}$ . However, both contextual types  $\Box(\vdash \text{List } ((\lambda x. x) \text{Nat}))$  and  $\Box(\vdash \text{List } \text{Nat})$  are equivalent and are equally valid types for the code objects. The  $\beta$  redex in the first contextual type now appears on the type level due to dependent types and computes. This action of bringing code ( $\mathcal{C}$ ) to the type level ( $\mathcal{D}$ ) is a special instance of the lifting lemma, which we call *code promotion*.

Finally, the meta-programming layer  $\mathcal{M}$  extends MLTT with constructs to support an explicit way to execute code. Particularly, to execute the code object `box`  $((\lambda x. x) \emptyset)$ , we first use `letbox` to bind it to a *meta-variable*  $u$  and then refer to  $u$ : `letbox`  $u \leftarrow \text{box } ((\lambda x. x) \emptyset)$  `in`  $u \approx (\lambda x. x) \emptyset \approx \emptyset$ . The first equivalence substitutes  $u$  for the code object and is another instance of the lifting lemma, where the code object at layer  $\mathcal{C}$  is lifted to layer  $\mathcal{M}$ , which we refer to as *code execution*.

Layered type theories such as DELAM share many similarities with two-level type theories, where a more powerful language sits on top of weaker ones (e.g. [Allais \[2024\]](#); [Kovács \[2022\]](#); [Pientka et al. \[2019\]](#)). Nevertheless, there are also significant differences. For example, 2LTT by [Allais \[2024\]](#); [Kovács \[2022\]](#) lacks the ability to intensionally analyze code and the fine-grained control over computation of code types. In Cocon [\[Pientka et al. 2019\]](#), MLTT sits on top of the logical framework LF [\[Harper et al. 1993\]](#), where we define object languages using higher-order

abstract syntax (HOAS) [Pfenning and Elliott 1988]. The LF objects are wrapped in contextual types on the upper MLTT level, where we can write recursive programs to analyze them. However, the flexibility in defining object languages in LF comes at a cost: though we can write functions to evaluate LF objects in MLTT, such evaluation functions do not exist for all object languages. Moreover, Cocon also lacks the fine-grained control of equivalences that is built into DELAM. In Cocon, contextual objects and contextual types only describe static syntax. Type-level equivalence such as  $\Box (\vdash \text{List } ((\lambda x. x) \text{Nat}))$  and  $\Box (\vdash \text{List Nat})$  needs to be reasoned about explicitly. The ability to exploit equivalences at different layers is particularly important when working with rich type theories such as MLTT.

In this paper, we show that DELAM is a suitable dependent type theory for meta-programming that allows us to quote objects as code, compose code, recursively analyze code objects, and execute code. It is a significant step towards a type theory suitable for daily meta-programming in PAs and provides a fresh perspective of practical meta-programming systems in today's PAs. Concretely our contributions in this paper are:

- We describe a syntactic theory for DELAM (Sec. 3), a dependent layered modal type theory with a non-cumulative, Tarski-style universe hierarchy that supports execution of and recursion on code of MLTT. In DELAM, code objects are represented as static syntax trees at layer  $c$  and do not compute. Their types live at a higher layer  $D$ , which corresponds to pure MLTT and allows computation. The meta-programming layer  $M$  extends MLTT with the support for quoting, composing, and recursively analyzing code.
- We illustrate the power of DELAM with two common tactics: an equality checker and a general solver tactic (see Sec. 2). The generated code is guaranteed to be well-formed thanks to modelling code types as contextual types.
- We prove DELAM's weak normalization and the decidability of convertibility based on Kripke logical relations (Sec. 5 and 6) à la Abel et al. [2018]; We present a simpler form of the logical relations than those by Abel et al. [2018], based on partial equivalence relations (PERs), which cut the length of proofs by approximately half.

Due to space limitations, we focus on describing the main idea in this paper and readers can find many omitted details in our technical report [Hu and Pientka 2024a].

## 2 DELAM by Examples

DELAM supports quotation of static MLTT code objects in `box` and composition of code objects. These code objects can be further analyzed by recursion and eventually be executed to obtain MLTT programs. These features allow us to write widely used tactics, and the generated proofs are guaranteed well-formed as ascribed by their types.

### 2.1 Recursion on Code Objects describing MLTT Terms

As a first example, we implement a tactic, which checks whether two expressions are equal up to associativity and commutativity (AC). This functionality is frequently desired in a proof assistant. For example, both Isabelle/HOL [Wenzel et al. 2024, Sec. 9.3.3] and Lean provide such AC solvers.

To fit the tactic in the available space, we concentrate on an AC checker for summations of expressions of natural numbers. In addition, we assume that the addition operation  $+$ , its associativity and commutativity, and transitivity and Leibniz substitution of equality have been defined at the layer  $c$ , so we have access to these definitions in a code object. Finally, we assume that a summation is written in the right-associated form, i.e. of the form  $a_1 + (a_2 + \dots + (a_n + a_{n+1}))$  and  $a_i$  is an object of type  $\text{Nat}$ . In practice, this can always be achieved using a preprocessor.

```

ac-check : (g : Ctx) =>
  (a b : □ (g ⊢ Nat)) →
  letbox a' ← a; b' ← b in
  Option (□ (g ⊢ Eq Nat a' b'))
ac-check g (box (a1 + a2)) b =
  letbox b' ← b in
  search g (box a1) (box b')
  >>= λ (c, pf1).
  letbox c' ← c in
  ac-check g (box a2) (box c')
  >>= λ pf2.
  letbox pf1' ← pf1;
        pf2' ← pf2 in
  Some (box (trans
    (cong (a1 +_) pf2')
    (sym pf1')))
ac-check g (box a') b =
  letbox b' ← b in
  if eq? (box a') (box b')
  then Some (box refl) else None

search : (g : Ctx) =>
  (a b : □ (g ⊢ Nat)) →
  letbox a' ← a; b' ← b in
  Option (Σ (c : □ (g ⊢ Nat)).
    letbox c' ← c in
    □ (g ⊢ Eq Nat b' (a' + c'))))
search g a (box (b1 + b2)) =
  letbox a' ← a in
  if eq? (box b1) (box a')
  then Some (box b2, box refl)
  else if eq? (box b2) (box a')
  then Some (box b1, box (comm b1 b2))
  else search g (box a') (box b2)
  >>= λ (c, pf).
  letbox c' ← c; pf' ← pf in
  Some ( box (b1 + c')
        , box (trans (cong (b1' +_) pf')
                  (pull b1' a' c'))))
search g a (box b') = letbox a' ← a in None

```

Fig. 1. An implementation of an equality checker modulo associativity and commutativity (AC)

The main idea of this tactic is simple: to compare  $a_1 + \dots + (a_n + a_{n+1})$  with  $b$ , we match all  $a_i$ 's with addends in  $b$  until  $a_{n+1}$ , and then we just compare  $a_{n+1}$  with the rest of  $b$  for syntactic equality. The tactic is implemented as the `ac-check` function in Fig. 1 in an Agda-like surface syntax. In the type of `ac-check`, a fat arrow  $\Rightarrow$  denotes a *meta-function*. The `ac-check` function is polymorphic in the context  $g$ , so it applies for any regular context. Next, it takes two pieces of code  $a$  and  $b$  as inputs, with their open variables in  $g$ . The return type should express the equality between  $a$  and  $b$ .

To do that, we first use `letbox` to unpack  $a$  and  $b$  to obtain meta-variables that we can subsequently use in the contextual type  $(\square (g \vdash \text{Eq Nat } a' b'))^1$ . This contextual type describes the proof that  $a'$  and  $b'$  are equal where `Eq` refers to propositional equality. The actual return type of `ac-check` is an `Option`, as we might not be able to prove the equality.

The `ac-check` function is defined by recursion on the input  $a$ .

(1) If  $a$  is `box (a1 + a2)`, then  $a_1$  and  $a_2$  are pattern variables representing open code of addends of  $a$  in the parametric context  $g$ . We use the `search` function to look for  $a_1$  in  $b$ . It optionally returns code  $c$  with an equality proof of  $b = a_1 + c$ . Hence  $c$  describes the remainder of  $b$  excluding  $a_1$  and we recursively compare  $a_2$  with  $c$ . We use the bind operation (`>>=`) to short-circuit the uninteresting `None` case. If the recursion is successful, then we obtain two proofs:  $pf_1$  for  $b = a_1 + c$  and  $pf_2$  for  $a_2 = c$ . The proof obligation of `ac-check` requires the code that proves  $a_1 + a_2 = b$ , i.e.  $\square (g \vdash \text{Eq Nat } (a_1 + a_2) b')$ . This obligation can be filled in by following a sequence of equalities:  $a_1 + a_2 = a_1 + c = b$ . More concretely, we first use `letbox` to obtain meta-variables  $pf_1'$ , which represents code of  $g \vdash \text{Eq Nat } b' (a_1 + c')$ , and  $pf_2'$ , which represents code of  $g \vdash \text{Eq Nat } a_2 c'$ . To simulate the sequence of equalities above, we begin with transitivity of equality, which chains two equality proofs. In the first equality proof, we apply the congruence of addition to obtain  $g \vdash \text{Eq Nat } (a_1 + a_2) (a_1 + c')$  from  $pf_2'$ . The second equality proof requires  $g \vdash \text{Eq Nat } (a_1 + c') b'$ , which is just a symmetrized version of  $pf_1'$ .

<sup>1</sup>A practical front-end would insert `letbox`'s smartly, but in this paper, we would like to make the mechanism explicit.

Agda users might find the last step familiar. Indeed, providing the proof obligations in DELAM resembles filling in holes manually in Agda in many aspects. The critical difference however, is that in DELAM, proof obligations are fulfilled by (meta-)programs.

(2) If  $a$  is not an addition at all, then the only possibility for it to be equal to  $b$  is that both syntactically describe the same code. The function `eq?` tests the syntactic equality between two code objects by recursively comparing sub-structures.

The search function is the main driver of the algorithm. It recurses on  $b$  to look for an addend syntactically identical to  $a$ . If  $b$  has no addition, then `search` fails and returns `None`. If  $b$  is `box`  $(b_1 + b_2)$ , the first two `ifs` compare  $a$  with  $b_1$  and  $b_2$ . If either comparison succeeds, then we have found this addend. Otherwise, in the last `else` branch, the recursive search continues to look for  $a$  in  $b_2$ . If successful, we obtain some  $c$  and a proof `pf` of  $b_2 = a + c$ . Finally, we should return  $b_1 + c$  and a proof of  $b_1 + b_2 = a + (b_1 + c)$ . Similar to `ac-check`, we also begin with a `letbox` to obtain meta-variables  $c'$  and  $pf'$ , which represents code of  $g \vdash \text{Eq Nat } b_2 (a' + c')$ . The proof obligation is established by again a transitivity. First, by the congruence of addition, the first equality proves  $g \vdash \text{Eq Nat } (b_1 + b_2) (b_1 + (a' + c'))$ . Now the second equality requires a proof of  $g \vdash \text{Eq Nat } (b_1 + (a' + c')) (a' + (b_1 + c'))$ . This equality is established by an invocation of `pull`: `pull b1' a' c'`, which swaps  $b_1$  and  $a'$ . This property is called *left commutativity*, and is often required in an AC solver, e.g. in Isabelle/HOL and in Lean. Left commutativity can be proved by a sequence of associativity, commutativity and again associativity in MLTT.

We can use `ac-check` to algorithmically derive proofs for many tedious equations about natural numbers. The following lemma is one example which we would like to avoid proving manually:

```

lem : (x y z : Nat) → Eq Nat (x + (y + z)) (y + (z + x))
lem x y z = let Some pf ← ac-check (a : Nat, b : Nat, c : Nat) -- context
            (box (a + (b + c))) (box (b + (c + a)))
            in letbox u ← pf in u[x/a,y/b,z/c]

```

The first argument to `ac-check` is the ambient context  $a : \text{Nat}, b : \text{Nat}, c : \text{Nat}$  and unrelated to the function arguments  $x, y$  and  $z$ . The next two arguments are code objects describing both sides of the equation in the ambient context above. Since the invocation of `ac-check` is closed, it will return `Some pf` where `pf` is a code object denoting the equality proof of the code type  $\square(a:\text{Nat},b:\text{Nat},c:\text{Nat} \vdash \text{Eq Nat } (a + (b + c)) (b + (c + a)))$ . Note that `pf` is guaranteed to be a well-formed equality proof of the given contextual type. To use the equality proof `pf` at the layer  $M$ , we use `letbox` to bind it to the meta-variable  $u$ . Subsequently, in the body of `letbox`, we use  $u$  with a substitution to substitute  $x, y$  and  $z$  for  $a, b$  and  $c$ , respectively. This pattern of extracting a proof from code is an instance of code execution, which is justified by the lifting lemma.

Finally, definitions like `Option`, `Eq` and `>>=` are defined in MLTT, but the meta-programming layer  $M$  also has access to them, due to the uniform syntax of DELAM for all layers and the lifting lemma. Hence, users of DELAM only need to learn one language for proving and meta-programming.

## 2.2 Recursion on Code Objects describing MLTT Types

In DELAM, code objects describe not only MLTT terms, but also MLTT types. A code object representing a MLTT type has a contextual type  $\square (g \vdash @1)$  where  $@1$  denotes the universe level 1 of the type. Hence, DELAM supports recursion on the shape of a code object of types. Being able to write meta-programs that recursively analyze the code describing MLTT types is key to implementing tactics.

To illustrate, consider a goal of the following form: it is either a universally quantified formula  $(x : \text{Nat}) \rightarrow F'$ , a conjunction  $F_1 \wedge F_2$ , or an equality between arithmetic expressions. When quoting such a goal, we obtain a contextual code object  $F : \square (g \vdash @0)$ . Below we implement the `crush`

tactic which takes in  $F$  as an input and constructs a code object describing the proof for  $F$ , if it finds it. The tactic is implemented as a meta-program in DELAM by pattern matching on the shape of  $F$  and leverages the previous AC checker to crush equalities between arithmetic expressions.

```

crush : (g : Ctx) => (F : □ (g ⊢ @0)) → letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b)) = ac-check g (box a) (box b)
crush g (box (F1 ∧ F2)) = crush g (box F1) >>= λ (r1 : □ (g ⊢ F1)).
                             crush g (box F2) >>= λ (r2 : □ (g ⊢ F2)).
                             letbox pf1 ← r1 ; pf2 ← r2 in Some (box (pf1, pf2))
crush g (box ((x:Nat) → F)) = crush (g, x:Nat) (box F) >>= λ (r : □ (g, x:Nat ⊢ F)).
                             letbox pf ← r in Some (box (λ x. pf))
crush g (box _) = None

```

In the first case, if the goal formula is simply the equality between arithmetic expressions  $a$  and  $b$ , we call `ac-check`. Otherwise, if the goal is a conjunction, then we crush both components and compose their proofs at the end. We again use the bind operation (`>>=`) for convenience. If the goal is a universal quantification, then we extend the regular context with the parameter  $x:Nat$  and recurse on `box F`. In general, abstracting over contexts is crucial for recursion on binders that extend regular contexts (see also [Pientka et al. 2019]). When we have found a proof for  $F$ , we return a proof for  $(x : Nat) \rightarrow F$  by embedding it in a  $\lambda$ . The last case captures all other shapes of formulas and returns `None`. We can now use this meta-program to solve more complex goals such as the following:

```

lem2 : (x y : Nat) → Eq Nat (x + y) (y + x) ∧
        ((z : Nat) → Eq Nat (x + (y + z)) (y + (z + x)))
lem2 = let Some pf ←
        crush () (box ((x y : Nat) → Eq Nat (x + y) (y + x) ∧
                      ((z : Nat) → Eq Nat (x + (y + z)) (y + (z + x)))))
        in letbox u ← pf in u

```

This tactic follows the same pattern as in the last example.

### 3 Syntax of DELAM

Starting this section, we define DELAM formally. DELAM includes multiple layers: the layer `c` accommodates static code objects, whereas the layer `D` corresponds to the dependent type theory of MLTT. The topmost layer `M` extends MLTT with contextual types and other constructs for composition, execution, and recursion on code. While the syntax of DELAM (see Fig. 2) is uniform and thus users only need to learn one language, layers are distinguished on the judgmental level, where we define well-formed objects and valid computational behaviors. We will dissect the syntax gradually and discuss our design decisions. Readers might find hyperlinks in the text convenient.

#### 3.1 Explicit Universe Polymorphism

DELAM supports universe polymorphism following Bezem et al. [2022]. Universes ( $l$ ) form an idempotent commutative monoid, where  $l \sqcup l'$  takes the maximum of  $l$  and  $l'$ . The  $\omega$  level is added to support universe-polymorphic functions. A universe level  $l$  is well-formed if all variables in  $l$  appear in  $L$  and  $l$  contains no  $\omega$ . Similar to Agda, universes respect a number of equalities: identity ( $0 \sqcup l = l$ ), distributivity ( $1 + (l \sqcup l') = (1 + l) \sqcup (1 + l')$ ), absorption ( $l \sqcup (1 + l) = 1 + l$ ), commutativity, associativity and idempotence. Given two well-formed universes  $l$  and  $l'$ , whether  $l = l'$  is decidable as implemented in Agda. One possible algorithm is to compare the universe level associated with each universe variable in  $l$  and  $l'$ . Moreover, universes form a partial order ( $l \leq l' := l \sqcup l' = l'$ ) and a strict order ( $l < l' := 1 + l \leq l'$ ). The strict order is well-founded. This fact will be used to define the logical relations to prove weak normalization and decidability of convertibility.

$i, j, k \in \{v, c, d, m\}$	(Layers, where $v < c < d < m$ )
$x, u, \ell$	(Regular, meta-, universe variables, resp.)
$l := \ell \mid 0 \mid 1 + l \mid l \sqcup l' \mid \omega$	(Universe levels)
$L := \cdot \mid L, \ell$	(Universe contexts)
$\Phi, \Psi := \cdot \mid \Phi, u : E$	(Meta-contexts)
$\Gamma, \Delta := \cdot \mid u \mid \Gamma, x : T @ l$	(Regular contexts)
$\delta := \cdot \mid \text{wk} \mid \delta, t/x$	(Regular substitutions)
$E := \text{Ctx} \mid (\Gamma \vdash_i @ l) \mid (\Gamma \vdash_i T @ l)$	(Contextual kinds)
$e := \Gamma \mid T \mid t$	(Contextual objects)
$M, S, T := \text{Ty}_l \mid \text{Nat} \mid \Pi^{l,l'}(x : S).T \mid u^\delta \mid \vec{\ell} \Rightarrow^l T \mid (u : E) \Rightarrow^l T \mid \square E \mid \text{El}^l t$	(Types)
$s, t := \text{Ty}_l \mid \text{Nat} \mid \Pi^{l,l'}(x : s).t \mid x \mid u^\delta \mid \text{zero} \mid \text{succ } t$	(Terms)
$\quad \mid \lambda^{l,l'}(x : S).t \mid (t : \Pi^{l,l'}(x : S).T) s \mid \Lambda^l u.t \mid t \$ e \mid \Lambda^l \vec{\ell}.t \mid t \$ \vec{l}$	
$\quad \mid \text{box } e \mid \text{letbox}_{x,M}^l u \leftarrow (s : \square E) \text{ in } t \mid \text{elim}^{l_1, l_2} \vec{M} \vec{b} (t : \square E)$	
$\vec{M} := (\ell, u_\Gamma, x_T.M_{\text{Typ}}) (\ell, u_\Gamma, u_T, x_t.M_{\text{Trm}})$	(Two motives for recursion on code)
$\vec{b} := \vec{b}_{\text{Typ}} \vec{b}_{\text{Trm}}$	(Branches for recursion on code)
$b_{\text{Typ}} := (\ell, u_\Gamma.t_{\text{Ty}}) \mid (u_\Gamma.t_{\text{Nat}}) \mid (\ell, \ell', u_\Gamma, u_S, u_T, x_S, x_T.t_\Pi) \mid (\ell, u_\Gamma, u_t, x_t.t_{\text{El}})$	(Branches for code of MLTT types)
$b_{\text{Trm}} := (\ell, u_\Gamma.t'_{\text{Ty}}) \mid (u_\Gamma.t'_{\text{Nat}}) \mid (\ell, \ell', u_\Gamma, u_S, u_t, x_S, x_t.t'_\Pi) \mid (\ell, u_\Gamma, u_T, u_x.t_x) \mid (u_\Gamma.t_{\text{zero}})$	
$\quad \mid (u_\Gamma, u_t, x_t.t_{\text{succ}}) \mid (\ell, \ell', u_\Gamma, u_S, u_T, u_t, x_S, x_t.t_\lambda) \mid (\ell, \ell', u_\Gamma, u_S, u_T, u_t, u_s, x_S, x_T, x_t, x_s.t_{\text{app}})$	(Branches for code of MLTT terms)

Fig. 2. Syntax of DELAM

DELAM supports a universe polymorphic function space ( $\vec{\ell} \Rightarrow^l T$ ); this type is introduced by abstractions ( $\Lambda^l \vec{\ell}.t$ ) and used by applications ( $t \$ \vec{l}$ ).

### 3.2 Variables, Contexts and Substitutions

DELAM distinguishes regular regular variables ( $x$ ) and meta-variables ( $u$ ), which represent holes in code. Meta-variables for types and terms are associated with a regular substitution  $\delta$ . We may omit writing the regular substitution if it is the identity  $\text{id}$ . To support universe polymorphism, we use  $\ell$  to range over all universe variables. Three kinds of variables are stored in three kinds of contexts: regular contexts ( $\Gamma$ ), meta-contexts ( $\Psi$ ), and universe contexts ( $L$ ). Universe contexts are just collections of universe variables. Meta-contexts bind meta-variables  $u$  to a contextual kind ( $E$ ) which describes either a regular context or a contextual object for types or for terms. Due to context polymorphism, there are two base cases for a regular context: empty ( $\cdot$ ) or a context variable ( $u : \text{Ctx}$ ) bound in meta-contexts. Correspondingly, there are also two base cases for a regular substitution. The empty substitution  $\cdot$  maps to an empty context, while the weakening  $\text{wk}$  is the base case for context variables (c.f. Sec. 4.1).

### 3.3 Tarski-style Universes and Types

Instead of a more common Russell-style universe hierarchy, DELAM employs a Tarski-style hierarchy [Palmgren 1998]. The Tarski style brings the syntax closer to the semantics than the Russell style and simplifies our semantic development. In the Tarski-style formulation, types and terms belong to two different but mutually defined grammars. Dependent types are achieved by introducing a decoder  $\text{El}$ , which decodes an encoding of a type. For example,  $\text{El}^0 \text{Nat}$  decodes to the type  $\text{Nat}$ . As a consequence,  $\text{Ty}$ ,  $\text{Nat}$ , and  $\Pi$  types are present both as types and as terms and

we are overloading their syntax. In addition, types also include universe-polymorphic functions ( $\vec{\ell} \Rightarrow^I T$ ), meta-functions ( $(u : E) \Rightarrow^I T$ ), and contextual types ( $\square E$ ) describing well-typed open code. As shown in Sec. 2, meta-functions are typically used to set up preliminaries in the typing context in order to describe contextual types, which denotes the actual code to manipulate.

Since we employ a non-cumulative universe hierarchy, following Pujet and Tabareau [2023], we use  $@i$  to mark the universe levels in contextual kinds and other places explicitly.

### 3.4 Dissecting Types and Terms of DELAM

In the grammar of DELAM (see Fig. 2) we distinguish between three different contextual objects  $e$ : MLTT contexts  $\Gamma$ , MLTT terms  $t$ , and MLTT types  $T$ . Contextual objects of MLTT types and terms are only meaningful w.r.t. a regular context. Unlike in prior work (e.g. [Cave and Pientka 2012; Pientka et al. 2019]), we omit the regular contexts associated with contextual objects as they are uniquely determined by the types of the contextual objects. Though we use abstract names in the presentation, under the hood, we assume de Bruijn indices for variables, so we avoid issues of  $\alpha$ -renaming.

Disregarding universe levels for a moment, terms in DELAM include MLTT objects like the encodings of types (e.g. natural numbers ( $\text{Nat}$ ) and function types ( $\Pi(x : S).t$ )), function abstractions ( $\lambda(x : S).t$ ), and function applications ( $(t : \Pi(x : S).T) s$ ). Note that in a function application we include the type annotation of  $t$ . This is necessary for quotation of terms and obtaining sufficient typing information of  $t$ . Without this annotation, we cannot generally derive what  $T$  is given only the overall type of the application. A recursor for natural numbers can be added in the usual way, but we omit it here for a more compact presentation.

Inspired by Cocon [Pientka et al. 2019], we include abstractions over contextual objects ( $\Lambda u.t$ ) to introduce meta-functions and applications of meta-functions ( $t \$ e$ ). In addition, following Hu and Pientka [2024b], we add the capability to quote code (box  $e$ ), to compose and execute code using letbox, and to recurse over code objects using elim.

Both letbox and the recursor are defined to account for possible type dependencies. As a consequence, we annotate both constructs with the overall type of the expression (a.k.a. the motive  $M$ ). In particular, letbox carries a motive annotation  $x.M$ . In the case of elim, we need two motive annotations  $\vec{M}$ , as it defines a mutual recursion principle over code objects of MLTT types and terms at the same time. The mutual recursion also leads to two sets of branches  $\vec{b}_{\text{Typ}}$  and  $\vec{b}_{\text{Tm}}$ . It may be surprising to see that we define one recursor which includes two mutually defined recursion principles: one for types and one for terms. This comes from the fact that types and terms in DELAM are also mutually defined due to the Tarski-style universe hierarchy. Hence, when we analyze a function application  $(t : \Pi(x : S).T) s$ , we may recursively analyze not only the terms  $t$  and  $s$ , but also the types  $T$  and  $S$ . Similarly, when we analyze a type of the form  $\text{El}^I t$ , we may recursively analyze the term  $t$ . This mutual dependency is the source of the complication in the recursor.

Intuitively, the branches in  $\vec{b}_{\text{Tm}}$  cover all possible MLTT terms and those in  $\vec{b}_{\text{Typ}}$  cover all possible MLTT types.  $\vec{b}$  then collects both kinds of branches. In  $\vec{b}_{\text{Typ}}$ , we cover the following cases: when we encounter the code of a universe, we choose branch  $t_{\text{Y}}$ ; when we encounter the type  $\text{Nat}$ , we choose branch  $t_{\text{Nat}}$ ; when we encounter a  $\Pi$  type, we choose the branch  $t_{\Pi}$ ; and when we encounter a decoder  $\text{El}$ , we choose the branch  $t_{\text{El}}$ . In  $\vec{b}_{\text{Tm}}$ , we have cases for variables ( $t_x$ ), natural numbers ( $t_{\text{zero}}$  and  $t_{\text{succ}}$ ), function abstractions ( $t_\lambda$ ), function applications ( $t_{\text{app}}$ ), and the encodings of types ( $t'_{\text{Y}}$ ,  $t'_{\text{Nat}}$  and  $t'_{\Pi}$ ). In each branch, we list the pattern variables  $u$  of the pattern being considered. For example, in the branch  $t_{\text{succ}}$ , the pattern variable  $u_t$  describes the pattern variable in the pattern  $\text{succ } u_t$ . Similarly, in the branch  $t_{\text{app}}$ , the pattern would be  $(u_t : \Pi(x : u_S).u_T) u_s$ , so we



have pattern variables for the function  $u_t$  and the argument  $u_s$ , as well as for the type annotations  $u_S$  and  $u_T$ . Since the regular context might grow in the  $\Pi$  case and in the  $\lambda$  case, each branch maintains a context variable  $u_\Gamma$ . Finally, each branch includes recursion variables  $x$  for recursive calls. In the succ case,  $x_t$  refers to the recursive call over  $u_t$  when we encounter the pattern  $\text{succ } u_t$ . In the application case, we have four recursive calls:  $x_t$  corresponds to the recursive call on the function  $u_t$ ;  $x_s$  corresponds to the recursive call on the argument  $u_s$ ; and  $x_T$  and  $x_S$  correspond to the recursive calls on the types  $u_S$  and  $u_T$  respectively. Last, each branch introduces universe variables to quantify the universe levels of pattern variables. The variables in the branches  $\vec{b}_{\text{Typ}}$  are organized in a similar principle.

#### 4 Syntactic Judgments in DELAM

Though layers do not impact the uniform syntax of DELAM, they do make a significant difference in the judgments. Most syntactic judgments in DELAM are parameterized by a layer  $i$ . Through the layer  $i$ , we can define rules that generically hold at multiple layers and rules that only exist at a specific layer. For each layer, we control not only the validity of objects, but also the computational behaviors. In this way, we cleanly distinguish the layer  $c$  for code objects, the layer  $d$  for pure MLTT, and the layer  $m$  of meta-programming. Following [Cave and Pientka \[2012\]](#); [Pientka et al. \[2019\]](#), we further introduce the layer  $v$ , which only describes static code objects for MLTT variables. This layer only appears in the pattern variable when we hit the case for variables ( $t_x$ ) during a recursion on code of terms. Based on the matryoshka principle, these layers form a strict order:  $v < c < d < m$ . As explained in [Sec. 1](#), the computational power strictly increases as the layer ascends. In particular, code objects at layer  $c$  do not compute, so that the recursion principles are applied to the syntactic shapes of code. On the other hand, code promotion may bring a code object to its type at layer  $d$ , where computation may occur. The topmost layer  $m$  further extends the layer  $d$  with computational constructs that composes, executes and does recursion on code.

Most syntactic judgments are defined parametrically in layer  $i$ . Here the layer  $i$  refers to the layer which the *principal object* lives at. For example,  $L \mid \Psi; \Gamma \vdash_i t : T @ l$  defines that  $t$  is well-typed at layer  $i$ . In this case, the regular context  $\Gamma$  and the type  $T$  live at a higher layer than  $i$ . For example, if  $i = c$ , then  $t$  is a code object, and therefore its type  $T$  lives at layer  $d$ . We define the function  $\uparrow(i)$  to compute the layer of the surrounding typing environment when the principal object lives at layer  $i$ . It is defined as  $\uparrow(m) := m$ , and  $\uparrow(i) := d$  if  $i \neq m$ . Presupposition illustrates the purpose of  $\uparrow(i)$  (c.f. [Lemma 4.3](#)). We discuss below a few selected rules for each judgment in DELAM. We highlight the principal object in the informal explanation for each judgment in shades. For conciseness, we assume all parameterized universes  $l$  to be well-formed.

##### 4.1 Well-formed Regular and Meta-Contexts

We begin with the discussion on the well-formedness of meta- and regular contexts w.r.t. a universe context  $L$ . A meta-context  $\Psi$  is well-formed, if every contextual kind  $E$  in  $\Psi$  is well-formed at layer  $i$ . Here  $i$  determines [the well-formedness of contextual kind  \$E\$](#) . A regular context  $\Gamma$  is well-formed, if every type declaration  $x : T @ l$  in  $\Gamma$  is well formed.

$$\begin{array}{c}
 \boxed{L \vdash \Psi} \text{ Meta-context } \boxed{\Psi} \text{ is wf} \quad \boxed{L \mid \Psi \vdash_i \Gamma} \text{ At layer } i \text{ the regular context } \boxed{\Gamma} \text{ is wf} \\
 \hline
 \frac{}{L \vdash \cdot} \quad \frac{L \mid \Psi \vdash_i E}{L \vdash \Psi, u : E} \quad \frac{L \vdash \Psi}{L \mid \Psi \vdash_i \cdot} \quad \frac{L \vdash \Psi \quad u : \text{Ctx} \in \Psi}{L \mid \Psi \vdash_i u} \quad \frac{L \mid \Psi \vdash_i \Gamma \quad L \mid \Psi; \Gamma \vdash_i T @ l}{L \mid \Psi \vdash_i \Gamma, x : T @ l}
 \end{array}$$

A regular substitution  $\delta$  is well-formed, if all terms within are well-formed.

$$\boxed{L \mid \Psi; \Gamma \vdash_i \delta : \Delta} \text{ At layer } i \text{ regular substitution } \boxed{\delta} \text{ substitutes variables in } \Delta \text{ with terms in } \Gamma$$

$$\frac{L \mid \Psi \vdash_{\uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \dots} \quad \frac{L \mid \Psi \vdash_{\uparrow(i)} (u, \Gamma)}{L \mid \Psi; u, \Gamma \vdash_i \text{wk} : u} \quad \frac{L \mid \Psi; \Delta \vdash_{\uparrow(i)} T @ l \quad L \mid \Psi; \Gamma \vdash_i t : T[\delta] @ l}{L \mid \Psi; \Gamma \vdash_i \delta, t/x : \Delta, x : T @ l}$$

## 4.2 Contextual Kinds and Contextual Objects

We have three different possible contextual objects and kinds: MLTT contexts, MLTT types and MLTT terms. We use the layer  $i$  to control which contextual objects and kinds are accessible at a given layer. For example,  $\text{Ctx}$  is only well-formed at layer  $\text{D}$ . The well-formedness of  $\square E$  is defined such that we do not have  $\square \text{Ctx}$ , i.e. we cannot return a code object of context. However, meta-functions  $(u : \text{Ctx}) \Rightarrow^l T$  where we abstract over contexts are valid. Code objects include both contextual MLTT terms  $t$  of kind  $(\Gamma \vdash_i T @ l)$  and contextual MLTT types  $T$  of kind  $(\Gamma \vdash_i @ l)$ . Both are available at layer  $\text{c}$ , so we have access to code of types and terms.

$$\boxed{L \mid \Psi \vdash_i E} \text{ At layer } i \text{ the contextual kind } \boxed{E} \text{ is well-formed}$$

$$\frac{L \vdash \Psi}{L \mid \Psi \vdash_{\text{D}} \text{Ctx}} \quad \frac{L \mid \Psi \vdash_{\uparrow(i)} \Gamma \quad i \in \{\text{c}, \text{D}\}}{L \mid \Psi \vdash_i (\Gamma \vdash_i @ l)} \quad \frac{L \mid \Psi; \Gamma \vdash_{\uparrow(i)} T @ l \quad i \in \{\text{v}, \text{c}\}}{L \mid \Psi \vdash_i (\Gamma \vdash_i T @ l)}$$

$$\boxed{L \mid \Psi \vdash_i e : E} \text{ At layer } i \text{ contextual object } \boxed{e} \text{ has contextual kind } E$$

$$\frac{L \mid \Psi \vdash_{\text{D}} \Gamma}{L \mid \Psi \vdash_{\text{D}} \Gamma : \text{Ctx}} \quad \frac{L \mid \Psi; \Gamma \vdash_i T @ l \quad i \in \{\text{c}, \text{D}\}}{L \mid \Psi \vdash_i T : (\Gamma \vdash_i @ l)} \quad \frac{L \mid \Psi; \Gamma \vdash_i t : T @ l \quad i \in \{\text{v}, \text{c}\}}{L \mid \Psi \vdash_i t : (\Gamma \vdash_i T @ l)}$$

## 4.3 Types and Terms

We first define the well-formedness of types  $T$  at layer  $i$ . By controlling layer  $i$ , we control what types are available. For example,  $\square E$  is only available at layer  $\text{M}$  and hence all lower layers ( $\text{v}$ ,  $\text{c}$ ,  $\text{D}$ ) only have access to MLTT terms and types.

In the rules for types, due to Tarski universes à la Palmgren [1998],  $E1^l t$  decodes the encoding  $t : \text{Ty}_l$  to an actual type. The parameter  $i$  means that  $E1$  is available at all  $\text{c}$ ,  $\text{D}$  and  $\text{M}$ . Rules for types like  $\square E$  are only available at layer  $\text{M}$ . Note that  $\square E$  lives on universe level 0 regardless of the universe of the type or the term in  $E$ . This is because  $\square E$  encodes an intrinsically typed syntax of MLTT at layer  $\text{M}$ , which corresponds to a logically stronger sub-language than MLTT, so universe level 0 is large enough to encode all well-formed types and terms of MLTT. This observation is modeled in the semantics (c.f. Sec. 5.5), where  $\square E$  need not to refer to other semantics at layer  $\text{M}$ , so its semantics can be placed on level 0.

$$\boxed{L \mid \Psi; \Gamma \vdash_i T @ l} \text{ At layer } i \text{ type } \boxed{T} \text{ is wf on universe level } l$$

$$\frac{L \mid \Psi \vdash_{\text{M}} \Gamma \quad L \mid \Psi \vdash_{\text{c}} E}{L \mid \Psi; \Gamma \vdash_{\text{M}} \square E @ 0} \quad \frac{L \mid \Psi \vdash_{\text{D}} E \quad L \mid \Psi, u : E; \Gamma \vdash_{\text{M}} T @ l}{L \mid \Psi; \Gamma \vdash_{\text{M}} (u : E) \Rightarrow^l T @ l} \quad \frac{L \mid \Psi; \Gamma \vdash_i t : \text{Ty}_l @ l+1}{L \mid \Psi; \Gamma \vdash_i E1^l t @ l} \quad \frac{L, \vec{\ell} \mid \Psi; \Gamma \vdash_{\text{M}} T @ l}{L \mid \Psi; \Gamma \vdash_{\text{M}} \vec{\ell} \Rightarrow^l T @ \omega}$$

Following the same principle, **well-typed terms** in vanilla MLTT are defined parametrically in layer  $i$ . These terms include encodings of types, natural numbers, functions, and variables. When referring to a **meta-variable**  $u$ , we need to supply a regular substitution  $\delta$  to fill in the open variables. The premise  $i' \leq i$  builds the lifting lemma into the typing rule by allowing to use meta-variables from a lower layer at a higher one. Terms related to meta-programming are only available at layer  $\text{M}$ , e.g. the box constructor and `letbox`. The **core syntax of `letbox`** requires a motive  $x.M$ , which

computes the result type. The letbox body  $t$  lives in an extended meta-context with  $u$  and has type  $M$  with  $x$  substituted by  $\text{box } u^{\text{id}}$ , where  $\text{id}$  is the identity regular substitution.

$L \mid \Psi; \Gamma \vdash_i t : T @ l$  At layer  $i$  term  $t$  has type  $T$  at universe level  $l$

$$\begin{array}{c}
\frac{L \mid \Psi \vdash_{\uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \top y_l : \top y_{l+1} @ 2+l} \quad \frac{u : (\Delta \vdash_{i'} T @ l) \in \Psi \quad i' \leq i \quad L \mid \Psi; \Gamma \vdash_i \delta : \Delta}{L \mid \Psi; \Gamma \vdash_i u^\delta : T[\delta] @ l} \\
\frac{L \mid \Psi \vdash_{\uparrow(i)} \Gamma \quad x : T @ l \in \Gamma}{L \mid \Psi; \Gamma \vdash_i x : T @ l} \quad \frac{L \mid \Psi; \Gamma \vdash_i S @ l \quad L \mid \Psi; \Gamma, x : S @ l \vdash_i t : T @ l'}{L \mid \Psi; \Gamma \vdash_i \lambda^{l'}(x : S).t : \Pi^{l'}(x : S).T @ l \sqcup l'} \\
\frac{L \mid \Psi; \Gamma \vdash_i S @ l \quad L \mid \Psi; \Gamma, x : S @ l \vdash_i T @ l'}{L \mid \Psi; \Gamma \vdash_i t : \Pi^{l'}(x : S).T @ l \sqcup l'} \quad \frac{L \mid \Psi; \Gamma \vdash_i s : S @ l \quad L \mid \Psi \vdash_M \Gamma \quad L \mid \Psi \vdash_c e : E}{L \mid \Psi; \Gamma \vdash_M \text{box } e : \square E @ 0} \\
\frac{L \mid \Psi; \Gamma, x : \square E @ 0 \vdash_M M @ l \quad L \mid \Psi; \Gamma \vdash_M s : \square E @ 0 \quad L \mid \Psi, u : E; \Gamma \vdash_M t : M[\text{box } u^{\text{id}}/x] @ l}{L \mid \Psi; \Gamma \vdash_M \text{letbox}_{x.M}^l u \leftarrow (s : \square E) \text{ in } t : M[s/x] @ l}
\end{array}$$

Finally, we discuss some selected equivalence rules. We focus here on term equivalences and omit the equivalences for types  $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$ . All equivalence judgments include symmetry, transitivity, and naturally derived congruence rules at all layers. We omit these rules in the interest of space. We show here two  $\beta$  equivalence rules. In DELAM, no computation rule is available at layers  $v$  and  $c$ . Computation rules for terms in MLTT like the  $\beta$  rule for functions are available at both layers  $d$  and  $m$  to handle both code promotion and code execution. Rules for meta-programs like the  $\beta$  rule for letbox and for recursors are only available at layer  $m$ .

$L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$  At layer  $i$  the term  $t$  is equivalent to the term  $t'$

$$\begin{array}{c}
\frac{i \in \{d, m\} \quad L \mid \Psi; \Gamma \vdash_i S @ l}{L \mid \Psi; \Gamma, x : S @ l \vdash_i T @ l' \quad L \mid \Psi; \Gamma, x : S @ l \vdash_i t : T @ l' \quad L \mid \Psi; \Gamma \vdash_i s : S @ l} \\
\frac{L \mid \Psi; \Gamma \vdash_i (\lambda^{l'}(x : S).t : \Pi^{l'}(x : S).T) s \approx t[s/x] : T[s/x] @ l'}{L \mid \Psi \vdash_M \Gamma \quad L \mid \Psi \vdash_c e : E \quad L \mid \Psi; \Gamma, x : \square E @ 0 \vdash_M M @ l \quad L \mid \Psi, u : E; \Gamma \vdash_M t : M[\text{box } u^{\text{id}}/x] @ l} \\
\frac{L \mid \Psi; \Gamma \vdash_M \text{letbox}_{x.M}^l u \leftarrow (\text{box } e : \square E) \text{ in } t \approx t[e/u] : M[\text{box } e/x] @ l}
\end{array}$$

#### 4.4 Static Code and Lifting Lemma

Two guiding lemmas of DELAM are the *lifting lemma* and the *static code lemma*. The first one says that a well-typed term at a lower layer is also well-typed at higher layers. The latter states that equivalence between code objects is syntactic equality. For space, lemmas in this paper are not fully stated. Please see our technical report for details.

LEMMA 4.1 (LIFTING). *If  $i \leq i'$ , and*

- $L \mid \Psi; \Gamma \vdash_i T @ l$ , then  $L \mid \Psi; \Gamma \vdash_{i'} T @ l$ ;
- $L \mid \Psi; \Gamma \vdash_i t : T @ l$ , then  $L \mid \Psi; \Gamma \vdash_{i'} t : T @ l$ .

LEMMA 4.2 (STATIC CODE). *If  $i \in \{v, c\}$ , and*

- $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$ , then  $T = T'$ ;
- $L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$ , then  $t = t'$ ;

A consequence of the lifting lemma is that code objects at layer  $c$  can be lifted to layers  $d$  and  $m$  for free. It justifies *code execution* by using the lifting lemma to execute a code object of layer  $c$  at

layer  $m$ . It also enables *code promotion*, which allows us to promote the syntactic representation of a code object at layer  $c$  to layer  $d$  to allow us to capture all equivalences of MLTT at layer  $d$ .

One consequence of the static code lemma is no interesting computational behavior at layers  $v$  and  $c$ , so terms at both layers describe the static syntax of code objects. This is a necessary condition to intensionally analyze code from layer  $c$  and  $v$ .

Presupposition is another important lemma that holds about our syntactic typing rules. It says that if the principal object is being defined at layer  $i$ , regular contexts and types live at layer  $\uparrow(i)$ .

LEMMA 4.3 (PRESUPPOSITION).

- If  $L \mid \Psi; \Gamma \vdash_i T @ l$ , then  $L \mid \Psi \vdash_{\uparrow(i)} \Gamma$ .
- If  $L \mid \Psi; \Gamma \vdash_i t : T @ l$ , then  $L \mid \Psi \vdash_{\uparrow(i)} \Gamma$  and  $L \mid \Psi; \Gamma \vdash_{\uparrow(i)} T @ l$ .

#### 4.5 Weak-head Reductions

We follow [Abel et al. \[2018\]](#) to establish the proofs of weak normalization and the decidability of convertibility, so we need a description of weak-head normal forms (WHNFs) and a notion of weak-head reductions. Their definitions are entirely standard. WHNFs for types ( $W$ ) are either type constructors, or neutral types ( $V$ ), which are meta-variables ( $u^\delta$ ) or a decoding of neutral terms ( $E1^l v$ ). WHNFs for terms ( $w$ ) are either in introduction forms, or neutral terms ( $v$ ), which are either variables or elimination forms blocked by other neutrals. Weak normalization then proves that all well-formed types and terms must reach their WHNFs in finite steps of reductions.

The untyped one-step reductions for types ( $T \rightsquigarrow T'$ ) and terms ( $t \rightsquigarrow t'$ ) are also standard. In general, reductions are divided into two groups, one for reductions in head positions, and the other for actual computation. For types, head reductions only occur for  $E1$ :

$$E1^0 \text{Nat} \rightsquigarrow \text{Nat} \quad E1^{1+l} \text{Ty}_l \rightsquigarrow \text{Ty}_l \quad E1^{l+l'} \Pi^{l,l'}(x : s).t \rightsquigarrow \Pi^{l,l'}(x : E1^l s).E1^{l'} t \quad \frac{t \rightsquigarrow t'}{E1^l t \rightsquigarrow E1^l t'}$$

Reductions for terms also follow the same principle. One-step reductions enjoy typical properties like determinacy and preservation, and respect all substitutions. One-step reductions are generalized to multi-step reductions for types ( $T \rightsquigarrow^* T'$ ) and for terms ( $t \rightsquigarrow^* t'$ ). If multi-step reductions step to WHNFs, then determinacy ensures that WHNFs are unique.

Typed reductions  $L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow T' @ l$ ,  $L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow t' : T @ l$  and their multi-step variants are obtained by pairing untyped reductions with corresponding typing judgments.

#### 4.6 Recursion on Code

Finally, we describe the rules for the recursors for code ( $\text{elim}^{l_1, l_2} \vec{M} \vec{b} (t : \square E)$ ). We have listed the syntax towards the end of [Fig. 2](#). As mentioned before, this term encompasses in fact two mutually defined recursion principles: one for code of types and the other for code of terms. Therefore, we have two motives  $M_{\text{Typ}}$  and  $M_{\text{Trm}}$  in  $\vec{M}$ . They describe the return types of the recursion principles on code of types and of terms respectively. The branches also fall into two categories: the branches for recursively analyzing types ( $b_{\text{Typ}}$ ) and terms ( $b_{\text{Trm}}$ ).

As the recursor describes two recursion principles, they give rise to two typing rules. We focus on the rule for the recursor for code of terms. We abbreviate the well-formedness of branches as

$\vec{b}$  wf and concentrate on checking well-formedness of motives  $M_{\text{Typ}}$  and  $M_{\text{Trm}}$  and the scrutinee  $t$ .

$$\frac{L, \ell \mid \Psi, u_{\Gamma} : \text{Ctx}; \Gamma, x_T : \square(u_{\Gamma} \vdash_{\text{C}} @ \ell) @ 0 \vdash_{\text{M}} M_{\text{Typ}} @ l_1 \quad L, \ell \mid \Psi, u_{\Gamma} : \text{Ctx}, u_T : (u_{\Gamma} \vdash_{\text{D}} @ \ell); \Gamma, x_t : \square(u_{\Gamma} \vdash_{\text{C}} u_T @ \ell) @ 0 \vdash_{\text{M}} M_{\text{Trm}} @ l_2 \quad \vec{b} \text{ wf} \quad L \mid \Psi; \Gamma \vdash_{\text{M}} t : \square(\Delta \vdash_{\text{C}} T @ l) @ 0 \quad L \mid \Psi; \Delta \vdash_{\text{D}} T @ l}{L \mid \Psi; \Gamma \vdash_{\text{M}} \text{elim}^{l_1, l_2} \vec{M} \vec{b} (t : \square(\Delta \vdash_{\text{C}} T @ l)) : M_{\text{Trm}}[l/\ell, \Delta/u_{\Gamma}, T/u_T, t/x_t] @ l_2}$$

The motives  $M_{\text{Typ}}$  and  $M_{\text{Trm}}$  abstract over the context variable  $u_{\Gamma}$ , which might change during recursion. As the return type might also depend on the scrutinee, the motives also abstract over  $x_T$  (the scrutinee as code of type) or  $x_t$  (the scrutinee as code of term). In the latter case, we also keep track of the type of the scrutinee, denoted by  $u_T$ . The overall type of the recursion is  $M_{\text{Trm}}[l/\ell, \Delta/u_{\Gamma}, T/u_T, t/x_t]$  where we replace  $u_{\Gamma}$  with the concrete regular context  $\Delta$  of the scrutinee,  $u_T$  with the type  $T$  of the scrutinee, and we instantiate  $x_t$  with the scrutinee  $t$  itself in  $M_{\text{Trm}}$ .

Notice that  $T$  lives at layer  $\text{D}$ , which supports computation, while the term eventually computed by  $t$  will be a static contextual code object describing the syntax of an MLTT term. As a consequence, if  $T$  is for example  $(\lambda x.x) \text{Nat}$ , it is equivalent to  $\text{Nat}$ . In other words, it captures the fact that we are only interested in analyzing a code object of type  $\text{Nat}$  and the exact shape of the type  $T$  is unimportant. On the other hand, if the code object  $t$  contains  $(\lambda x.x) \text{Nat}$  as a sub-code, the redex would remain, because this code object represents a different static syntax tree from that of code  $\text{Nat}$  due to the static code lemma.

Now let us consider the well-formedness of branches. As described earlier in Sec. 3.4 we distinguish branches for types ( $b_{\text{Typ}}$ ) and terms ( $b_{\text{Trm}}$ ). We use the branch for function applications  $t_{\text{app}}$  as a running example and other branches can be derived naturally following the same principle (see all the branches in [Hu and Pientka 2024a, Sec. 4.6] and [Hu 2024, Appendix F]). To facilitate the discussion, we use colors to differentiate contexts, universe variables, types and terms in the pattern and the scrutinee. For more readability, we simply write  $u$  for  $u^{\text{id}}$  when a meta-variable is associated with the identity substitution. In this branch, the scrutinee is the code of  $(t : \Pi^{l, l'}(x : S).T) s$  and is matched against the pattern  $(u_t : \Pi^{l, l'}(x : u_S).u_T) u_s$ . Each sub-structure in the scrutinee is matched by a pattern variable. These pattern variables are meta-variables  $u$  extended to the meta-context  $\Psi$  and with matching subscripts of the sub-structures. The well-formedness conditions of the pattern variables record the ambient contexts, and the types if the pattern variables are for code of terms. When we line up the pattern variables and the sub-structures, we find a correspondence not only between them, but also between their well-formedness conditions. For example, the typing of the pattern variable  $u_s$  encodes the well-formedness of its matching sub-structure  $s$ .

	Regular Context	Code Object	Well-formedness
Sub-structures	$\Gamma$	$s$	$L \mid \Psi; \Gamma \vdash_{\text{C}} s : S @ l$
Pattern variables	$u_{\Gamma}$	$u_s$	$u_s : (u_{\Gamma} \vdash_{\text{C}} u_s @ \ell)$

In addition to the pattern variables, there are two new universe variables  $\ell$  and  $\ell'$  to capture the universes of  $S$  and  $T$ . Finally, each meta-variable gives rise to a recursive call. The recursive calls are regular variables  $x$  extended to the regular context with matching subscripts of the sub-structures. The recursive calls on code of MLTT types such as  $u_S$  and  $u_T$  have the corresponding type  $M_{\text{Typ}}$  appropriately refined. In particular, the scrutinee  $x_T$  in  $M_{\text{Typ}}$  is instantiated with box  $u_S$  and box  $u_T$ , respectively. Since  $u_T$  has a longer regular context than  $u_S$ , this fact is reflected in the variable  $u_{\Gamma}$  in  $M_{\text{Typ}}$ . Recursive calls on code of terms such as  $u_t$  and  $u_s$  have the corresponding type  $M_{\text{Trm}}$  appropriately instantiated. Here we replace the scrutinee  $x_t$  in  $M_{\text{Trm}}$  with box  $u_s$  and box  $u_t$ , respectively. In addition to the regular context for each of  $u_t$  and  $u_s$ , we also refine the type  $u_T$  in

$M_{\text{Trm}}$  with a  $\Pi$  type and  $u_S$ , respectively. Collecting all additional assumptions in the contexts gives rise to the following well-formedness condition for the branch  $t_{\text{app}}$  for function applications:

$$\begin{array}{l}
 L, \ell, \ell' \mid \Psi, u_\Gamma : \text{Ctx} \\
 , u_S : (u_\Gamma \vdash_{\text{c}} @ \ell), u_T : (u_\Gamma, x : u_S @ \ell \vdash_{\text{c}} @ \ell') \\
 , u_t : (u_\Gamma \vdash_{\text{c}} \Pi^{\ell, \ell'}(x : u_S).u_T @ \ell \sqcup \ell'), u_s : (u_\Gamma \vdash_{\text{c}} u_S @ \ell) \\
 ; \Gamma, x_S : M_{\text{Typ}}[\ell/\ell, u_\Gamma/u_\Gamma, \text{box } u_S/x_T] @ l_1 \\
 , x_T : M_{\text{Typ}}[\ell'/\ell, (u_\Gamma, x : u_S @ \ell)/u_\Gamma, \text{box } u_T/x_T] @ l_1 \\
 , x_t : M_{\text{Trm}}[\ell \sqcup \ell'/\ell, u_\Gamma/u_\Gamma, \Pi^{\ell, \ell'}(x : u_S).u_T/u_T, \text{box } u_t/x_t] @ l_2 \\
 , x_s : M_{\text{Trm}}[\ell/\ell, u_\Gamma/u_\Gamma, u_S/u_T, \text{box } u_s/x_t] @ l_2 \\
 \vdash_{\text{M}} t_{\text{app}} : M_{\text{Trm}}[\ell'/\ell, u_\Gamma/u_\Gamma, u_T^{\text{id}, u_s/x}/u_T, \text{box } ((u_t : \Pi^{\ell, \ell'}(x : u_S).u_T) u_s)/x_t] @ l_2
 \end{array}
 \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} \text{Pattern variables} \\ \\ \\ \text{Recursive Calls} \end{array}$$

Since recursions only occur for syntactic sub-terms, it informally justifies the recursors and their termination. This informal observation is made precise when we give semantics to code in Sec. 5.3. The reduction rule is straightforward. We replace each component with the appropriate instantiation. Here we focus on the rule for one-step reductions to save space; the equivalence rule is naturally derived from the reduction rule. In this rule, we require the type of the code to be  $W$ , i.e. in WHNF, for determinacy.

$$\text{elim}^{l_1, l_2} \vec{M} \vec{b} (\text{box } ((t : \Pi^{l, l'}(x : S).T) s) : \square(\Gamma \vdash_{\text{c}} W @ l)) \rightsquigarrow t_{\text{app}}[l/\ell, l'/\ell', \sigma, \delta]$$

where  $\sigma = \Gamma/u_\Gamma, S/u_S, T/u_T, t/u_t, s/u_s$  is the meta-substitution which instantiates all pattern variables, and  $\delta$  builds all recursive calls. It is defined as:

$$\begin{array}{l}
 \delta = \text{elim}^{l_1, l_2} \vec{M} \vec{b} (\text{box } S : \square(\Gamma \vdash_{\text{c}} @ l)) \quad /x_S \\
 , \text{elim}^{l_1, l_2} \vec{M} \vec{b} (\text{box } T : \square(\Gamma, x : S @ l \vdash_{\text{c}} @ l')) \quad /x_T \\
 , \text{elim}^{l_1, l_2} \vec{M} \vec{b} (\text{box } t : \square(\Gamma \vdash_{\text{c}} \Pi^{l, l'}(x : S).T @ l \sqcup l'))/x_t \\
 , \text{elim}^{l_1, l_2} \vec{M} \vec{b} (\text{box } s : \square(\Gamma \vdash_{\text{c}} S @ l)) \quad /x_s
 \end{array}$$

To end the discussion on syntax, we recapitulate the difference between `letbox` and `elim`. Based on [Hu and Pientka \[2024b\]](#), the former is responsible for code composition and running, while the latter is for intensional analysis. They differ specifically in their computational behavior. In particular,  $\text{elim}^{l_1, l_2} \vec{M} \vec{b} (\text{box } u^\delta : \square E)$  is neutral, because  $u^\delta$  simply does not find a branch in  $\vec{b}$ . Meanwhile,  $\text{letbox}_{x, M}^l u' \leftarrow (\text{box } u^\delta : \square E)$  in  $t$  reduces to  $t[u^\delta/u']$ . This distinction is further revealed in the logical relations, where semantics for terms at layer  $\text{c}$  must carry two kinds of information: syntactic information about their shapes and semantic information about how they run (c.f. Sec. 5.3). In the next section, we will describe how we model features in DELAM semantically.

## 5 Kripke Logical Relations

The Kripke logical relations of DELAM follow the same outline as in [Abel et al. \[2018\]](#): the logical relations are parameterized by generic equivalences. To derive the fundamental theorems, we instantiate these generic equivalences. However, DELAM is quite complex due to the presence of dependent types, Tarski-style universes, and the distinction between MLTT ( $\text{c}$  and  $\text{d}$  layer) and meta-programming ( $\text{m}$  layer) which enables quoting, evaluating, and recursively analyzing code. To shorten the proofs as much as possible, we simplify the logical relations by adopting PER-style definitions. This allows us to define the logical relations with only two predicates – in contrast, [\[Abel et al. 2018\]](#) requires four predicates for types and terms. This improvement further allows us to more compactly state the necessary lemmas and proofs by reducing their number to approximately half. This significantly eases our meta-theoretic development. Nevertheless, the

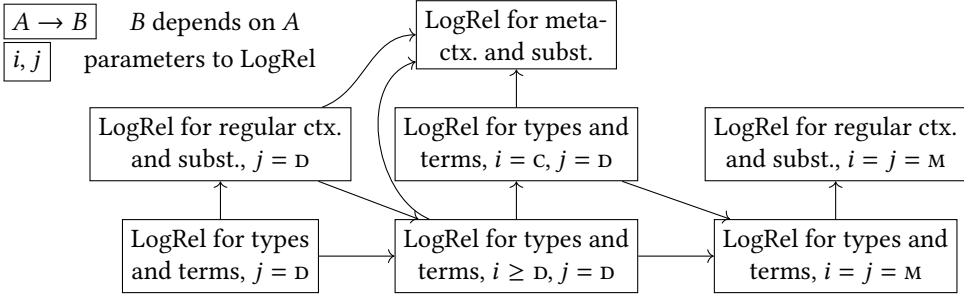


Fig. 3. Structure of logical relations

complete proofs remain very verbose and therefore, in this paper, we only focus on the main idea and refer the interested reader to the technical report [Hu and Pientka 2024a]. The structure of the logical relations is depicted in Fig. 3, where the nodes are clickable in a PDF viewer. The logical relations are parameterized by  $i$ , the layer for terms, and  $j$ , the layer for types. We give more explanations on  $i$  and  $j$  in Sec. 5.2.

### 5.1 Generic Equivalences

First, we define the generic equivalences for types and terms. The generic equivalences are parameters of judgments with laws to the logical relations, which we eventually instantiate with the conversion checking algorithm to obtain its completeness proof. There are four generic equivalence judgments:  $L \mid \Psi; \Gamma \vdash_i V \sim V' @ l$  and  $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$  are equivalences for neutral types and any types, and  $L \mid \Psi; \Gamma \vdash_i v \sim v' : T @ l$  and  $L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$  are those for neutral terms and any terms. The layer  $i$  only takes  $D$  or  $M$  because they are the only layers with computation. Their laws are formulated following closely the recipe by Abel et al. [2018]. These judgments are subsumed by syntactic equivalence at their layers and form PERs. They respect equivalence of contexts and types, and weakenings of contexts. The rest of the laws are to capture how equivalences are propagated under WHNFs. For example, for  $t$  and  $t'$  of type  $\Pi^{l,l'}(x : S).T$  to be equivalent, a law requires  $t x$  and  $t' x$  to be equivalent on type  $T$  for  $x : S$ . Two neutrals are generically equivalent if all sub-components are generically equivalent pointwise.

### 5.2 Logical Relations for Types in MLTT

The logical relations are defined on top of generic equivalence. They are Kripke in their stability under weakenings. We write  $L \mid \Psi; \Gamma \Longrightarrow_i L' \mid \Phi; \Delta$  for a weakening of three contexts, where  $\Gamma$  and  $\Delta$  are well-formed at layer  $i$ . If  $T$  is well-formed in  $L' \mid \Phi; \Delta$ , we directly say that  $T$  is weakened and is also well-typed in  $L \mid \Psi; \Gamma$ . As the first step, we define the logical relations for types and terms:  $L \mid \Psi; \Gamma \Vdash_i^j T \approx T' @ l$  and  $L \mid \Psi; \Gamma \Vdash_i^j t \approx t' : T @ l$ , where  $(i, j) \in \{(D, D), (M, D), (M, M)\}$  or equivalently  $i \geq j \geq D$ . The judgments say that  $T$  and  $T'$  are related (resp.  $t$  and  $t'$ ) as types at layer  $j$  and as terms at layer  $i$ . When  $j = D$ , it means that  $T$  and  $T'$  are related types in MLTT after reductions, regardless of which layer they and their terms actually live at. This separation of consideration in  $j$  is particularly important in the layering restriction lemma (Lemma 5.1), which gives a semantic explanation for code execution. As a mnemonic, the number of vertical bars in the turnstile matches the number of contexts. As we progress, this number gradually decreases, so it also serves as a progress bar for our development.

The logical relations are defined by

- (1) first, recursion on  $j$ , which means that we first define the semantics for types of MLTT before we consider all types at layer  $M$ ,

$$\begin{array}{c}
\frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* \text{Nat}_{@0}}{L \mid \Psi; \Gamma \Vdash_i^j T \approx T'_{@0}} \quad \frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* \text{Ty}_l_{@1+l}}{L \mid \Psi; \Gamma \Vdash_i^j T \approx T'_{@1+l}} \quad \frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* V_{@l}}{L \mid \Psi; \Gamma \Vdash_i^j T \approx T'_{@l}} \\
\frac{L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* \text{Nat}_{@0}}{L \mid \Psi; \Gamma \Vdash_i^j T' \approx T'_{@0}} \quad \frac{L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* \text{Ty}_l_{@1+l}}{L \mid \Psi; \Gamma \Vdash_i^j T' \approx T'_{@1+l}} \quad \frac{L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* V'_{@l}}{L \mid \Psi; \Gamma \Vdash_i^j T' \approx T'_{@l}} \\
L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* \Pi^{l,l'}(x : S_1).T_1_{@l \sqcup l'} \quad L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* \Pi^{l,l'}(x : S_2).T_2_{@l \sqcup l'} \\
\forall L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma . L' \mid \Phi; \Delta \Vdash_i^j S_1 \approx S_2_{@l} \\
\forall L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma \text{ and } L' \mid \Phi; \Delta \Vdash_i^j s \approx s' : S_1_{@l} . L' \mid \Phi; \Delta \Vdash_i^j T_1[s/x] \approx T_2[s'/x]_{@l'} \\
\hline
L \mid \Psi; \Gamma \Vdash_i^j T \approx T'_{@l \sqcup l'}
\end{array}$$

Fig. 4. Logical relations for types in MLTT

- (2) then a transfinite well-founded recursion on the universe levels,
- (3) at last, an induction-recursion [Dybjer and Setzer 2003] to relate types and terms.

The recursion on  $j$  before universe levels allows us to restart the universe level from 0 when referring to the logical relations for  $j = \mathbb{D}$  in those for  $j = \mathbb{M}$ . Therefore, all contextual types can safely live on level 0. We will discuss more when we define the logical relations for  $j = \mathbb{M}$  in Sec. 5.5.

The logical relations for types are defined inductively in Fig. 4. The four cases are natural numbers, universes, neutral types and  $\Pi$  types. The last case is the most complex one. First,  $T$  and  $T'$  reduce to their respective  $\Pi$  types. Then the third premise relates input types for all weakenings, and the fourth premise relates output types for all weakenings given related inputs  $s$  and  $s'$  of the type  $S_1$ . The logical relations for terms  $L \mid \Psi; \Gamma \Vdash_i^j t \approx t' : T_{@l}$  are defined by recursion on those of types. In the case of  $\Pi$  types where  $L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* \Pi^{l,l'}(x : S_1).T_1_{@l \sqcup l'}$ , we have

- $L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow^* w : \Pi^{l,l'}(x : S_1).T_1_{@l \sqcup l'}$  and  $L \mid \Psi; \Gamma \vdash_i t' \rightsquigarrow^* w' : \Pi^{l,l'}(x : S_1).T_1_{@l \sqcup l'}$ , i.e.  $t$  and  $t'$  reduce to WHNFs  $w$  and  $w'$ , respectively;
- $L \mid \Psi; \Gamma \vdash_i w \approx w' : \Pi^{l,l'}(x : S_1).T_1_{@l \sqcup l'}$ , saying that  $w$  and  $w'$  are generically equivalent,
- at last, for all  $L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma$  and related inputs  $L' \mid \Phi; \Delta \Vdash_i^j s \approx s' : S_1_{@l}$ , the results of applying  $w$  and  $w'$  are related:  $L' \mid \Phi; \Delta \Vdash_i^j w s \approx w' s' : T_1[s/x]_{@l}$ .

When  $j = \mathbb{D}$ , we have given the complete definition of the logical relations for MLTT. Compared to the version by Abel et al. [2018], we halve the number of definitions by defining the logical relations in a PER style. We still have to prove the same set of lemmas, but their statements and proofs are also halved in size. This makes the large semantic development of DELAM more manageable.

When  $j = \mathbb{M}$ , we still need more cases to handle types for meta-programming and universe-polymorphic functions (see Sec. 5.5). Nevertheless, the definitions that we have given here still apply. In other words, for types shared between layers  $\mathbb{D}$  and  $\mathbb{M}$ , their logical relations only differ in layers. This observation is captured by the layering restriction lemma:

LEMMA 5.1 (LAYERING RESTRICTION). *If  $L \mid \Psi; \Gamma \Vdash_{\mathbb{M}}^{\mathbb{D}} T \approx T'_{@l}$ ,*

- *then  $L \mid \Psi; \Gamma \Vdash_{\mathbb{M}}^{\mathbb{M}} T \approx T'_{@l}$ ;*
- *then  $L \mid \Psi; \Gamma \Vdash_{\mathbb{M}}^{\mathbb{M}} t \approx t' : T_{@l}$  and  $L \mid \Psi; \Gamma \Vdash_{\mathbb{M}}^{\mathbb{D}} t \approx t' : T_{@l}$  are equivalent.*

The most interesting direction in this lemma is  $\mathbb{M}$  to  $\mathbb{D}$ , i.e. that  $L \mid \Psi; \Gamma \Vdash_{\mathbb{M}}^{\mathbb{M}} t \approx t' : T_{@l}$  implies  $L \mid \Psi; \Gamma \Vdash_{\mathbb{M}}^{\mathbb{D}} t \approx t' : T_{@l}$ , given  $T$  as a type from layer  $\mathbb{D}$ , i.e. in MLTT. Effectively, if we know that  $T$  reduces to some type from MLTT, then we can *restrict* the relation between  $t$  and  $t'$  from  $j = \mathbb{M}$  to  $j = \mathbb{D}$ . This lemma is crucial to model code execution semantically. Consider an identity function  $\lambda(x : \text{Nat}).x$  for natural numbers from layer  $\mathbb{C}$ . The semantics of this function says that given a normalizing input  $t$  in MLTT,  $(\lambda x.x) t$  gives a normalizing output also in MLTT. However, after being lifted to layer  $\mathbb{M}$ , the identity function can be applied to meta-programs like



$$\begin{array}{c}
\frac{L \vdash \Psi}{L \mid \Psi \Vdash_i^j \cdot \approx \cdot} \quad \frac{L \vdash \Psi \quad u : \text{Ctx} \in \Psi}{L \mid \Psi \Vdash_i^j u \approx u} \quad \frac{\begin{array}{l} \forall L' \mid \Phi \implies L \mid \Psi . L' \mid \Phi \Vdash_i^j \Delta \approx \Delta' \\ \forall L' \mid \Phi \implies L \mid \Psi \text{ and } L' \mid \Phi; \Gamma \Vdash_i^j \delta \approx \delta' : \Delta . \\ L' \mid \Phi; \Gamma \Vdash_i^j T[\delta] \approx T'[\delta'] @ l \end{array}}{L \mid \Psi \Vdash_i^j \Delta, x : T @ l \approx \Delta', x : T' @ l}
\end{array}$$

Fig. 5. Logical relations for contexts

letbox  $u \leftarrow$  box zero in  $u$ , which is clearly not in MLTT. Layering restriction comes to the rescue by saying that even though meta-programs are not from MLTT, since we know  $\text{Nat}$  is a type in MLTT, every normalizing meta-program of type  $\text{Nat}$  can be lowered in the semantics to  $j = \text{D}$  to be passed as arguments to the identity function. Finally, the results are lifted again to  $j = \text{M}$  in the reverse direction. In this way, the machinery in code execution is semantically explained. Interestingly, though lifting (Lemma 4.1) monotonically brings terms from a lower layer to a higher one syntactically, layering restriction does need to be expressed as an equivalence in the semantics.

The logical relations for regular contexts ( $L \mid \Psi \Vdash_i^j \Gamma \approx \Delta$ ) and regular substitutions ( $L \mid \Psi; \Gamma \Vdash_i^j \delta \approx \delta' : \Delta$ ) are also defined inductive-recursively. The former is defined inductively in Fig. 5 and requires all relations between types to be stable under regular substitutions pointwise. The latter is defined recursively on the former, and is a generalization of the logical relations of terms. They are also defined in the PER style so that subsequent proofs are simplified.

### 5.3 Semantics for MLTT and Code

At the end of Sec. 4.6, we mentioned that terms at layer  $c$  need to maintain both semantic information and syntactic information. In their work, [Hu and Pientka \[2024b\]](#) achieve this by embedding the semantic information in an inductively defined judgment which stores syntactic information. This is exactly how we proceed here as well. First, we define semantics for types, terms and regular substitutions that are stable under regular substitutions. Effectively, these definitions give the semantics for pure MLTT. We always set  $j = \text{D}$  because we are only concerned about types from MLTT. We define  $L \mid \Psi; \Gamma \Vdash_{\geq \text{D}}^{\text{D}} T \approx T' @ l$  as for all  $L' \mid \Phi \implies L \mid \Psi$ ,  $k \geq \text{D}$ , and  $L' \mid \Phi; \Delta \Vdash_k^{\text{D}} \delta \approx \delta' : \Gamma$ , we have  $L' \mid \Phi; \Delta \Vdash_k^{\text{D}} T[\delta] \approx T'[\delta'] @ l$ . The condition  $k \geq \text{D}$  means that  $T$  and  $T'$  are related at both layers  $\text{D}$  and  $\text{M}$ . The judgment  $L \mid \Psi; \Gamma \Vdash_{\geq \text{D}}^{\text{D}} t \approx t' : T @ l$  is defined similarly, but instead we require the conclusion to be  $L' \mid \Phi; \Delta \Vdash_k^{\text{D}} t[\delta] \approx t'[\delta'] : T[\delta] @ l$ . The generalizations  $L \mid \Psi \Vdash_{\geq \text{D}}^{\text{D}} \Delta \approx \Delta'$  and  $L \mid \Psi; \Gamma \Vdash_{\geq \text{D}}^{\text{D}} \delta \approx \delta' : \Delta$  are defined in a similar manner. These semantic judgments capture the running information of these objects. For convenience, we also defined asymmetric variants by requiring both sides to be equal, e.g.  $L \mid \Psi; \Gamma \Vdash_{\geq \text{D}}^{\text{D}} T @ l$  is defined as  $L \mid \Psi; \Gamma \Vdash_{\geq \text{D}}^{\text{D}} T \approx T @ l$ . Given that these judgments characterize pure MLTT, they are closed under semantically related regular substitutions. For example,

**LEMMA 5.2 (REGULAR SUBSTITUTIONS).** *If  $L \mid \Psi; \Gamma \Vdash_{\geq \text{D}}^{\text{D}} t \approx t' : T @ l$  and  $L \mid \Psi; \Delta \Vdash_{\geq \text{D}}^{\text{D}} \delta \approx \delta' : \Gamma$ , then  $L \mid \Psi; \Delta \Vdash_{\geq \text{D}}^{\text{D}} t[\delta] \approx t'[\delta'] : T[\delta] @ l$ .*

Given the semantic judgments for running types, terms, etc., we then encapsulate them with syntactic information about shapes in the semantic judgments for code. In particular, we inductively define  $L \mid \Psi; \Gamma \Vdash_c^{\text{D}} T @ l$  for the semantics of code  $T$ ,  $L \mid \Psi; \Gamma \Vdash_c^{\text{D}} t : T @ l$  for the semantics of code  $t$  of type  $T$ , and  $L \mid \Psi; \Gamma \Vdash_c^{\text{D}} \delta : \Delta$  for the semantics of code for regular substitution  $\delta$ . We give two example rules for the judgments in Fig. 6 (see all the rules in [[Hu and Pientka 2024a](#), Sec. 7.4] and [[Hu 2024](#), Appendix G]). The framed premises in the rules come from the typing rules.

$\frac{L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} S @ l \quad L \mid \Psi; \Gamma, x : S @ l \Vdash_c^{\mathfrak{D}} T @ l'}{L \mid \Psi; \Gamma \Vdash_{\geq \mathfrak{D}}^{\mathfrak{D}} \Pi^{l,l'}(x : S).T @ l \sqcup l'}$	$\frac{L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} S @ l \quad L \mid \Psi; \Gamma, x : S @ l \Vdash_c^{\mathfrak{D}} T @ l' \quad L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} t : \Pi^{l,l'}(x : S).T @ l \sqcup l' \quad L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} s : S @ l}{L \mid \Psi; \Gamma \Vdash_{\geq \mathfrak{D}}^{\mathfrak{D}} T' \approx T[s/x] @ l'}$
$\frac{L \mid \Psi; \Gamma \Vdash_{\geq \mathfrak{D}}^{\mathfrak{D}} \Pi^{l,l'}(x : S).T @ l \sqcup l'}{L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} \Pi^{l,l'}(x : S).T @ l \sqcup l'}$	$\frac{L \mid \Psi; \Gamma \Vdash_{\geq \mathfrak{D}}^{\mathfrak{D}} (t : \Pi^{l,l'}(x : S).T) s : T' @ l'}{L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} (t : \Pi^{l,l'}(x : S).T) s : T' @ l'}$

Fig. 6. Selected rules for semantic judgment for code

They recursively record the syntactic information of sub-structures. The other premises are for semantic information. For a type, e.g. a  $\Pi$  type,  $L \mid \Psi; \Gamma \Vdash_{\geq \mathfrak{D}}^{\mathfrak{D}} \Pi^{l,l'}(x : S).T @ l$  denotes that the  $\Pi$  type can be run at both layers  $\mathfrak{D}$  and  $\mathfrak{M}$ . For a term, e.g. a function application, we need two pieces of information: the semantic information of  $t s$  and the semantic equivalence between  $T'$  and  $T[s/x]$ . Note that  $T[s/x]$  is originated from layer  $\mathfrak{C}$  and is brought to layer  $\mathfrak{D}$  via lifting to establish a semantic equivalence with  $T'$ . This semantic equivalence characterizes code promotion, where code objects  $T$  and  $s$  are lifted to layer  $\mathfrak{D}$  for computation. All other rules in the semantic judgments follow this exact pattern to encode both kinds of information. The following semantic lifting lemma extracts the semantic information from the judgments.

LEMMA 5.3 (SEMANTIC LIFTING). *If  $L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} t : T @ l$ , then  $L \mid \Psi; \Gamma \Vdash_{\geq \mathfrak{D}}^{\mathfrak{D}} t : T @ l$ .*

Since the semantic judgments for MLTT are closed under regular substitutions and the semantic judgments for code are basically typing judgments with extra semantic information, we can show that the semantic judgments for code are also closed under regular substitutions, e.g.

LEMMA 5.4 (REGULAR SUBSTITUTIONS). *If  $L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} t : T @ l$  and  $L \mid \Psi; \Delta \Vdash_c^{\mathfrak{D}} \delta : \Gamma$ , then  $L \mid \Psi; \Delta \Vdash_c^{\mathfrak{D}} t[\delta] : T[\delta] @ l$ .*

The closure under regular substitutions is crucial to model code composition semantically. Given a regular substitution  $\delta$ , though it does not propagate under box, i.e.  $(\text{box } e)[\delta] = \text{box } e$ , it might still be applied if it is given as part of a meta-variable. In other words, assuming a meta-substitution  $\sigma$ ,  $u^\delta[\sigma] = \sigma(u)[\delta[\sigma]]$  where  $\sigma(u)$  first looks up  $u$  in  $\sigma$ , and then we apply  $\delta[\sigma]$ , the result of applying  $\sigma$  to all terms in  $\delta$ , to the result of the lookup. The regular substitution lemma ensures that the overall result of code composition still maintains both semantic and syntactic information properly.

Next, we define the symmetrized variants  $L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} T \approx T' @ l$  as  $L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} T @ l$  and  $T = T'$ , and  $L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} t \approx t' : T @ l$  as  $L \mid \Psi; \Gamma \Vdash_c^{\mathfrak{D}} t : T @ l$  and  $t = t'$ . Effectively, we extend the logical relations for types and terms in Sec. 5.2 with  $i = \mathfrak{C}$ . This extension allows us to conveniently express the final semantic judgments for the fundamental theorems in Sec. 5.6.

#### 5.4 Logical Relations for Meta-contexts and Meta-substitutions

The semantic judgments for code in Sec. 5.3 are used in two places in the semantics: one is the logical relations for meta-contexts and substitutions in this section, and the other is the logical relations for types and terms for layer  $\mathfrak{M}$  in the next section. Continuing the recipe, the logical relations for meta-contexts and substitutions are defined inductive-recursively in the PER style. We define those for meta-contexts  $[L \Vdash \Psi \approx \Phi]$  inductively in Fig. 7. The Kripke structure of the logical relations is in the weakening of universe contexts. All premises  $\forall L' \implies L . L' \Vdash \Phi \approx \Phi'$  build the Kripke structure into the logical relations. The premise (1) requires the relation between  $\Gamma$  and  $\Gamma'$  to be stable under related meta-substitutions for  $k \geq \mathfrak{D}$ . Similarly, the premise (2) requires the relation between  $T$  and  $T'$  to be stable under both related meta- and regular substitutions.

$$\begin{array}{c}
\frac{\forall L' \Longrightarrow L . L' \Vdash \Phi \approx \Phi'}{L \Vdash \cdot \approx \cdot} \quad \frac{\forall L' \Longrightarrow L . L' \Vdash \Phi \approx \Phi' \quad \forall L' \Longrightarrow L \text{ and } L' \mid \Psi \Vdash \sigma \approx \sigma' : \Phi \text{ and } k \geq \mathsf{D} . \quad L' \mid \Psi \Vdash_k^{\mathsf{D}} \Gamma[\sigma] \approx \Gamma[\sigma']}{L \Vdash \Phi, u : \text{Ctx} \approx \Phi', u : \text{Ctx}} \quad \frac{\forall L' \Longrightarrow L . L' \Vdash \Phi \approx \Phi' \quad i \in \{\mathsf{C}, \mathsf{D}\} \quad \forall L' \Longrightarrow L \text{ and } L' \mid \Psi \Vdash \sigma \approx \sigma' : \Phi \text{ and } k \geq \mathsf{D} . \quad L' \mid \Psi \Vdash_k^{\mathsf{D}} \Gamma[\sigma] \approx \Gamma[\sigma']}{L \Vdash \Phi, u : (\Gamma \vdash_i @ l) \approx \Phi', u : (\Gamma' \vdash_i @ l)} \quad (1) \\
\frac{\forall L' \Longrightarrow L . L' \Vdash \Phi \approx \Phi' \quad i \in \{\mathsf{V}, \mathsf{C}\} \quad \forall L' \Longrightarrow L \text{ and } L' \mid \Psi \Vdash \sigma \approx \sigma' : \Phi \text{ and } k \geq \mathsf{D} . L' \mid \Psi \Vdash_k^{\mathsf{D}} \Gamma[\sigma] \approx \Gamma[\sigma'] \quad \forall L' \Longrightarrow L \text{ and } L' \mid \Psi \Vdash \sigma \approx \sigma' : \Phi \text{ and } k \geq \mathsf{D} \text{ and } L' \mid \Psi; \Delta \Vdash_k^{\mathsf{D}} \delta \approx \delta' : \Gamma[\sigma] . \quad L' \mid \Psi; \Delta \Vdash_k^{\mathsf{D}} T[\sigma][\delta] \approx T'[\sigma'][\delta'] @ l}{L \Vdash \Phi, u : (\Gamma \vdash_i T @ l) \approx \Phi', u : (\Gamma' \vdash_i T' @ l)} \quad (2)
\end{array}$$

Fig. 7. Logical relations for meta-contexts

The logical relations for meta-substitutions  $\boxed{L \mid \Psi \Vdash \sigma \approx \sigma' : \Phi}$  are defined by recursion on those for meta-contexts. We consider the case for  $L \mid \Psi \Vdash \sigma \approx \sigma' : \Phi, u : (\Gamma \vdash_i @ l)$ , where  $i \in \{\mathsf{C}, \mathsf{D}\}$ , which needs to satisfy the following conditions:

- $\sigma = \sigma_1, T/u$  and  $\sigma' = \sigma'_1, T'/u$ , with two related types  $T$  and  $T'$  to substitute  $u$ ;
- for all  $L' \Longrightarrow L$ , we have  $L' \mid \Psi \Vdash \sigma_1 \approx \sigma'_1 : \Phi$ , i.e.  $\sigma_1$  and  $\sigma'_1$  are recursively related;
- finally, depending on the value of  $i$ ,
  - if  $i = \mathsf{D}$ , then  $L \mid \Psi; \Gamma \Vdash_{\geq \mathsf{D}}^{\mathsf{D}} T \approx T' @ l$ , so  $T$  and  $T'$  are related at both layers  $\mathsf{D}$  and  $\mathsf{M}$ . In particular, they do not need to be syntactically identical;
  - if  $i = \mathsf{C}$ , then  $L \mid \Psi; \Gamma \Vdash_{\mathsf{C}}^{\mathsf{D}} T \approx T' @ l$ , which stores the syntactic information of  $T$  and implies  $T = T'$ . Due to the escape lemma, it also implies  $L \mid \Psi; \Gamma \Vdash_{\geq \mathsf{D}}^{\mathsf{D}} T \approx T' @ l$ .

In short, the logical relations for meta-substitutions relate two meta-substitutions pointwise, and store the syntactic information of types and terms for contextual kinds at layer  $\mathsf{C}$ . In Sec. 5.6, the fundamental theorems use these logical relations to require types, terms, etc. to be stable under meta-substitutions.

## 5.5 Logical Relations for Layer $\mathsf{M}$

Up until this section, we only use the logical relations for  $j = \mathsf{M}$ . In this section, we roll all the way back and revisit the logical relations for types and terms, but for  $i = j = \mathsf{M}$ . This section gives semantics to types for meta-programming and is the last step before giving the semantic judgments. The logical relations for types at layer  $\mathsf{M}$  are defined by extending Fig. 4 after setting  $i = j = \mathsf{M}$  with Fig. 8. We omit the rules for meta-functions due to their similarity to  $\Pi$  types. As the first step, we first reduce  $T$  and  $T'$  to some normal types, e.g. contextual types or universe-polymorphic functions. For contextual types for types to be related, we require  $\Delta$  and  $\Delta'$  to be related at both layers  $\mathsf{D}$  and  $\mathsf{M}$  (due to  $\geq \mathsf{D}$ ). For contextual types for terms, we in addition require the types  $T_1$  and  $T'_1$  to be stably related under regular substitutions. Since the logical relations for contextual types is not recursive for  $i = j = \mathsf{M}$ , we can safely restart the universe level at 0. This justifies the syntactic rules as well, where we put contextual types on universe level 0. For the logical relations of terms, we give the case for  $L \mid \Psi; \Gamma \Vdash_i^j t \approx t' : \boxed{(\Delta \vdash_{\mathsf{C}} T_1 @ l) @ 0}$ :

- first,  $L \mid \Psi; \Gamma \vdash_{\mathsf{M}} t \rightsquigarrow^* w : \boxed{(\Delta \vdash_{\mathsf{C}} T_1 @ l) @ 0}$  and  $L \mid \Psi; \Gamma \vdash_{\mathsf{M}} t' \rightsquigarrow^* w' : \boxed{(\Delta' \vdash_{\mathsf{C}} T_1 @ l) @ 0}$ , reducing  $t$  and  $t'$  to WHNFs, and
- $L \mid \Psi; \Gamma \vdash_{\mathsf{M}} w \approx w' : \boxed{(\Delta \vdash_{\mathsf{C}} T_1 @ l) @ 0}$ , i.e. the WHNFs are generically equivalent, and
- finally,  $L \mid \Psi; \Gamma \Vdash w \approx w' : \boxed{(\Delta \vdash_{\mathsf{C}} T_1 @ l)}$ , which is an inductive relation of two cases:

$$\frac{L \mid \Psi; \Delta \Vdash_{\mathsf{C}}^{\mathsf{D}} t_1 : T_1 @ l}{L \mid \Psi; \Gamma \Vdash \text{box } t_1 \approx \text{box } t_1 : \boxed{(\Delta \vdash_{\mathsf{C}} T_1 @ l)}} \quad \frac{L \mid \Psi; \Gamma \vdash_{\mathsf{M}} v \sim v' : \boxed{(\Delta \vdash_{\mathsf{C}} T_1 @ l) @ 0}}{L \mid \Psi; \Gamma \Vdash v \approx v' : \boxed{(\Delta \vdash_{\mathsf{C}} T_1 @ l)}}$$

$$\begin{array}{c}
L \mid \Psi; \Gamma \vdash_M T \rightsquigarrow^* \square(\Delta \vdash_c @i) @0 \\
L \mid \Psi; \Gamma \vdash_M T' \rightsquigarrow^* \square(\Delta' \vdash_c @i) @0 \\
L \mid \Psi \Vdash_{\geq D}^D \Delta \approx \Delta' \\
\hline
L \mid \Psi; \Gamma \Vdash_M^M T \approx T' @0
\end{array}
\qquad
\begin{array}{c}
L \mid \Psi; \Gamma \vdash_M T \rightsquigarrow^* \square(\Delta \vdash_c T_1 @i) @0 \\
L \mid \Psi; \Gamma \vdash_M T' \rightsquigarrow^* \square(\Delta' \vdash_c T'_1 @i) @0 \\
L \mid \Psi \Vdash_{\geq D}^D \Delta \approx \Delta' \quad L \mid \Psi; \Delta \Vdash_{\geq D}^D T_1 \approx T'_1 @i \\
\hline
L \mid \Psi; \Gamma \Vdash_M^M T \approx T' @0
\end{array}$$

$$\begin{array}{c}
L \mid \Psi; \Gamma \vdash_M T \rightsquigarrow^* \vec{\ell} \Rightarrow^l T_1 @\omega \quad L \mid \Psi; \Gamma \vdash_M T' \rightsquigarrow^* \vec{\ell} \Rightarrow^l T'_1 @\omega \\
\forall L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma \text{ and } \vec{l} \text{ that are well-formed in } L' \text{ and } |\vec{l}| = |\vec{\ell}|. \\
L' \mid \Phi; \Delta \Vdash_M^M T_1[\vec{l}/\vec{\ell}] \approx T'_1[\vec{l}/\vec{\ell}] @i[\vec{l}/\vec{\ell}] \\
\hline
L \mid \Psi; \Gamma \Vdash_M^M T \approx T' @\omega
\end{array} \tag{3}$$

Fig. 8. Logical relations for types at layer  $m$ 

The last condition relates  $w$  and  $w'$  on the contextual types  $\square(\Delta \vdash_c T_1 @i)$  in two cases. Either  $w = w' = \text{box } t_1$  for some  $t_1$ , which accompanies its semantic judgment for code, i.e.  $L \mid \Psi; \Delta \Vdash_c^D t_1 : T_1 @i$ . When we recurse on  $w$ , in the semantics, the mutual recursion principle for code is in fact interpreted as the mutual induction principle for  $L \mid \Psi; \Delta \Vdash_c^D t_1 : T_1 @i$ . On the other hand, if we compose  $t_1$  with some other code, the regular substitution lemma (Lemma 5.4) ensures that the result code still maintains proper semantic and syntactic information. Finally, if we run  $t_1$ , then we use the semantic lifting lemma (Lemma 5.3) to obtain the semantic information for execution, in conjunction with the layering restriction lemma (Lemma 5.1). In conclusion, the semantics justifies all usages of code. If  $w$  and  $w'$  are neutral, then we cannot say more than that they are generically equivalent.

The last case in the logical relations is the universe-polymorphic functions. In the premise (3), we substitute some arbitrary well-formed  $\vec{l}$  for  $\vec{\ell}$  in  $l$ . We cannot know in particular which exact universe level the result of the substitution is. The only thing that we are sure about is that  $l[\vec{l}/\vec{\ell}]$  is some finite, well-formed universe level. Therefore, in order to refer to any finite universe level, universe-polymorphic functions must be modeled on level  $\omega$ , hence requiring a transfinite recursion on universe levels in the definition of the logical relations.

The logical relations for regular contexts and substitutions for  $i = j = m$  have been defined in Fig. 5.

## 5.6 Semantic Judgments and Fundamental Theorems

Finally, we define the semantic judgments for DELAM. The semantic judgments state the stability of principal objects in the judgments under all substitutions at all higher layers. First, we define the semantic judgment for meta-contexts  $L \Vdash \Psi$  as for all  $L' \vdash \phi : L$ , we have  $L' \models \Psi[\phi] \approx \Psi[\phi]$ , where  $\phi$  is a universe substitution. Then the semantic judgment for equivalent regular contexts  $L \mid \Psi \Vdash_i \Gamma \approx \Delta$  where  $i \in \{D, M\}$  is defined as a conjunction of  $L \Vdash \Psi$  and for all  $L' \vdash \phi : L$ ,  $L' \mid \Phi \Vdash \sigma \approx \sigma' : \Psi[\phi]$  and  $k \geq i$ , we have  $L' \mid \Phi \Vdash_k^{(i)} \Gamma[\phi][\sigma] \approx \Delta[\phi][\sigma']$ . Effectively, this judgment says that the relation between  $\Gamma$  and  $\Delta$  is stable under all universe and meta-substitutions at all layers  $k \geq i$ . Notice that in the conclusion, we set  $j = \uparrow(i)$ , which is where regular contexts live when terms live at layer  $i$ . Its asymmetric version  $L \mid \Psi \Vdash_i \Gamma$  requires both sides to be equal:  $L \mid \Psi \Vdash_i \Gamma \approx \Gamma$ .

Next, we define the semantic judgment for types  $L \mid \Psi; \Gamma \Vdash_i T \approx T' @i$ . It follows the same principle. We first require  $L \mid \Psi \Vdash_{\uparrow(i)} \Gamma$ , and then for all  $L' \vdash \phi : L$ ,  $L' \mid \Phi \Vdash \sigma \approx \sigma' : \Psi[\phi]$ ,  $k \geq i$

and  $L' \mid \Phi; \Delta \Vdash_k^{\uparrow(i)} \delta \approx \delta' : \Gamma[\phi][\sigma]$ , we have

$$L' \mid \Phi; \Delta \Vdash_k^{\uparrow(i)} T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] @ l[\phi]$$

In other words, the relation between  $T$  and  $T'$  is stable under all substitutions at all layers above  $i$ . Then the semantic judgment for terms  $L \mid \Psi; \Gamma \Vdash_i t \approx t' : T @ l$  is defined in the same way, except that we require  $L \mid \Psi; \Gamma \Vdash_{\uparrow(i)} T @ l$ , and at the end, the conclusion is changed to

$$L' \mid \Phi; \Delta \Vdash_k^{\uparrow(i)} t[\phi][\sigma][\delta] \approx t'[\phi][\sigma'][\delta'] : T[\phi][\sigma][\delta] @ l[\phi]$$

The last semantic judgment is for regular substitutions  $L \mid \Psi; \Gamma \Vdash_i \delta \approx \delta' : \Delta$ , which is again defined similarly. These semantic judgments set up the right inductive invariants for the fundamental theorems, so that they can be proved by induction on the syntactic judgments. The most important cases in the theorems are:

**THEOREM 5.5 (FUNDAMENTAL).**

- If  $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$ , then  $L \mid \Psi; \Gamma \Vdash_i T \approx T' @ l$ .
- If  $L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$ , then  $L \mid \Psi; \Gamma \Vdash_i t \approx t' : T @ l$ .

The proof of the fundamental theorems follows a very interesting pattern, where we must work backwards following the order of layers. More specifically, to prove the first statement of Theorem 5.5 in an induction, if a typing rule is only available for  $i = M$ , then we proceed normally as in other type theories. However, if a rule is available at multiple layers, e.g. the congruence rule for  $\Pi$  types, then we have multiple statements to prove.

First, we take  $i = M$ , then the semantic judgment eventually requires a proof of  $L' \mid \Phi; \Delta \Vdash_M^M T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] @ l[\phi]$ , since  $k \geq M$  means  $k = M$ . Next we take  $i = D$ . In this case,  $k \in \{D, M\}$ , so the proof requires  $L' \mid \Phi; \Delta \Vdash_k^D T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] @ l[\phi]$ . This case is very similar to the case for  $i = M$ , with very minor differences in layers. When we take  $i = C$ , then  $k \geq C$  now can take three different values. As the proof obligation, we need to prove  $L' \mid \Phi; \Delta \Vdash_k^D T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] @ l[\phi]$ . Notice that here cases for  $k \geq D$  actually have been proved when  $i = D$ , so the only addition is to handle  $k = C$ . But then even this case is quite trivial, because when  $k = C$ , our goal becomes the semantic judgment for code of types, which is just a repackaging of the cases for  $k \geq D$ .

If we consider what information layers contain, this proof pattern make even more sense. For a term at layer  $M$ , the only information that it has is how it runs at layer  $M$ . Meanwhile if a term is from layer  $D$ , then we know that it can be run at both layers  $D$  and  $M$  due to lifting. For code from layer  $C$ , we must in addition remember its syntactic shape, which adds strictly more information on top of its running information. The proof of the fundamental theorems is similar to opening an onion: we peel off and assign semantics to DELAM from the outside layer by layer as we add more and more information, until the very end when DELAM is entirely modeled.

If we set all substitutions to identities and  $k = i$ , then the fundamental theorems simply imply the logical relations.

**COROLLARY 5.6 (COMPLETENESS OF LOGICAL RELATIONS).** *If  $i \in \{D, M\}$ ,*

- If  $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$ , then  $L \mid \Psi; \Gamma \Vdash_i^i T \approx T' @ l$  and  $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$ .
- If  $L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$ , then  $L \mid \Psi; \Gamma \Vdash_i^i t \approx t' : T @ l$  and  $L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$ .

Note that  $j = i = \uparrow(i)$  because we know  $i \in \{D, M\}$ . In other words, syntactically equivalent types or terms are also logically related and thus generically equivalent. In Sec. 6.3, where we set the generic equivalences to the conversion checking algorithm, this corollary immediately proves the completeness of the conversion checking algorithm.

## 6 Consequences of Fundamental Theorems

With the fundamental theorems, we are able to instantiate the generic equivalences to obtain important conclusions like weak normalization, injectivity of type constructors, consistency and the decidability of convertibility. In this section, we focus on deriving these conclusions.

### 6.1 First Instantiation: Syntactic Equivalence

The first instantiation assigns syntactic equivalence for types and terms to the generic equivalences. In this case, the laws are quite trivial to prove. The first important theorem to extract from the fundamental theorems is weak normalization.

**THEOREM 6.1 (WEAK NORMALIZATION).** *If  $i \in \{D, M\}$ , then*

- *if  $L \mid \Psi; \Gamma \vdash_i T @ l$ , then for some WHNF  $W$ ,  $L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W @ l$ ;*
- *if  $L \mid \Psi; \Gamma \vdash_i t : T @ l$ , then for some WHNF  $w$ ,  $L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow^* w : T @ l$ .*

It is only a weak normalization theorem, because we fix one specific reduction strategy. The weak normalization theorem says that well-formed types and terms always reduce to some WHNFs. This theorem is proved from Corollary 5.6, where weak normalization is built into the logical relations.

The next important theorem is injectivity of type constructors.

**THEOREM 6.2 (INJECTIVITY OF TYPE CONSTRUCTORS).**

- *If  $L \mid \Psi; \Gamma \vdash_i \Pi^{l,l'}(x : S).T \approx \Pi^{l,l'}(x : S').T' @ l \sqcup l'$  and  $i \in \{D, M\}$ , then  $L \mid \Psi; \Gamma \vdash_i S \approx S' @ l$  and  $L \mid \Psi; \Gamma, x : S @ l \vdash_i T \approx T' @ l'$ .*
- *If  $L \mid \Psi; \Gamma \vdash_M \Box(\Delta \vdash_C @ l) \approx \Box(\Delta' \vdash_C @ l) @ 0$ , then  $L \mid \Psi \vdash_D \Delta \approx \Delta'$ .*
- *If  $L \mid \Psi; \Gamma \vdash_M \Box(\Delta \vdash_C T @ l) \approx \Box(\Delta' \vdash_C T @ l) @ 0$ , then  $L \mid \Psi \vdash_D \Delta \approx \Delta'$  and  $L \mid \Psi; \Delta \vdash_D T \approx T' @ l$ .*

We only list three cases here for brevity. The theorem says that type constructors are injective w.r.t. syntactic equivalence. This theorem is proved from Corollary 5.6. The logical relations for types require matching sub-structures to be related, from which we extract their equivalences.

The last theorem that we obtain from this instantiation is the consistency theorem.

**THEOREM 6.3 (CONSISTENCY).** *There is no term  $t$  that satisfies this typing judgment:*

$$\cdot \mid \cdot \vdash_M t : \ell \implies^{1+\ell} \Pi^{1+\ell, \ell}(x : \text{Ty}_\ell). \text{El}^\ell x @ \omega$$

The consistency theorem says that, it is not possible to generically construct a term of any type on any universe level. The proof proceeds as follows. First, this theorem is the same as proving that there is no  $t'$  such that  $\ell \mid \cdot; x : \text{Ty}_\ell @ 1 + \ell \vdash_M t' : \text{El}^\ell x @ \ell$ . Let us assume such  $t'$ . Then it has a neutral type  $\text{El}^\ell x$ , so  $t'$  must reduce to some neutral  $v$  by the logical relations of  $\text{El}^\ell x$ . We do induction on  $v$  and see that  $v$  must be eventually blocked by  $x$ . But  $x$  as a neutral type cannot be eliminated, nor can it have type  $\text{El}^\ell x$ , so a contradiction is established.

### 6.2 Conversion Checking

Following Abel et al. [2018], we define the conversion checking algorithm. Due to layering and meta-programming constructs, there are more operations in our conversion checking algorithm than that by Abel et al., as we also need to compare regular contexts and substitutions. The conversion checking algorithm is split into two modes. In the checking mode, the algorithm returns true or false, while in the inference mode, the algorithm infers a universe level and potentially a type on that level for neutral terms. Selected rules for the algorithm are defined Fig. 9. The algorithm is layered at  $i \in \{D, M\}$ , the only two layers where interesting computation occurs. The main entry points are  $\boxed{L \mid \Psi; \Gamma \vdash_i T \overset{\wedge}{\iff} T' @ l}$ , which checks the convertibility of  $T$  and  $T'$  on

$$\begin{array}{c}
\boxed{L \mid \Psi; \Gamma \vdash_i T \hat{\longleftrightarrow} T' @ l} \text{ and } \boxed{L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' @ l} \text{ and } \boxed{L \mid \Psi; \Gamma \vdash_i W \longleftrightarrow W' @ l} \text{ for types} \\
\frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W @ l \quad L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* W' @ l \quad L \mid \Psi; \Gamma \vdash_i W \longleftrightarrow W' @ l}{L \mid \Psi; \Gamma \vdash_i T \hat{\longleftrightarrow} T' @ l} \quad \frac{L \mid \Psi \vdash_i \Gamma \quad u : (\Delta \vdash_{i'} @ l) \in \Psi \quad i' \leq i \quad L \mid \Psi; \Gamma \vdash_i \delta \hat{\longleftrightarrow} \delta' : \Delta}{L \mid \Psi; \Gamma \vdash_i u^\delta \longleftrightarrow u^{\delta'} @ l} \\
\frac{L \mid \Psi; \Gamma \vdash_i v \longleftrightarrow v' : \text{Ty}_l @_{1+l} \quad L \mid \Psi \vdash_i \Gamma}{L \mid \Psi; \Gamma \vdash_i \text{El}^l v \longleftrightarrow \text{El}^l v' @ l} \quad \frac{L \mid \Psi \vdash_i \Gamma}{L \mid \Psi; \Gamma \vdash_i \text{Nat} \longleftrightarrow \text{Nat} @_0} \quad \frac{L \mid \Psi \vdash_i \Gamma}{L \mid \Psi; \Gamma \vdash_i \text{Ty}_l \longleftrightarrow \text{Ty}_l @_{1+l}} \\
\frac{L \mid \Psi; \Gamma \vdash_i S \hat{\longleftrightarrow} S' @ l \quad L \mid \Psi; \Gamma, x : S @ l \vdash_i T \hat{\longleftrightarrow} T' @ l'}{L \mid \Psi; \Gamma \vdash_i \Pi^{l,l'}(x : S).T \longleftrightarrow \Pi^{l,l'}(x : S').T' @ l \sqcup l'} \quad \frac{L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' @ l}{L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' @ l} \\
\frac{L \mid \Psi \vdash_M \Gamma \quad L, \vec{\ell} \mid \Psi; \Gamma \vdash_M T \hat{\longleftrightarrow} T' @ l}{L \mid \Psi; \Gamma \vdash_M (\vec{\ell} \Rightarrow^l T) \longleftrightarrow (\vec{\ell} \Rightarrow^l T') @_\omega} \quad \frac{L \mid \Psi \vdash_M \Gamma \quad L \mid \Psi \vdash_D \Delta \hat{\longleftrightarrow} \Delta'}{L \mid \Psi; \Gamma \vdash_M \Box(\Delta \vdash_C @ l) \longleftrightarrow \Box(\Delta' \vdash_C @ l) @_0} \\
\frac{L \mid \Psi \vdash_M \Gamma \quad L \mid \Psi \vdash_D \Delta \hat{\longleftrightarrow} \Delta' \quad L \mid \Psi; \Delta \vdash_D T \hat{\longleftrightarrow} T' @ l}{L \mid \Psi; \Gamma \vdash_M \Box(\Delta \vdash_C T @ l) \longleftrightarrow \Box(\Delta' \vdash_C T' @ l) @_0} \\
\boxed{L \mid \Psi; \Gamma \vdash_i t \hat{\longleftrightarrow} t' : T @ l} \text{ and } \boxed{L \mid \Psi; \Gamma \vdash_i w \longleftrightarrow w' : W @ l} \text{ for any and normal terms} \\
\frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W @ l \quad L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow^* w : T @ l \quad L \mid \Psi; \Gamma \vdash_i t' \rightsquigarrow^* w' : T @ l \quad L \mid \Psi; \Gamma \vdash_i w \longleftrightarrow w' : W @ l}{L \mid \Psi; \Gamma \vdash_i t \hat{\longleftrightarrow} t' : T @ l} \quad \frac{L \mid \Psi; \Gamma \vdash_i v \longleftrightarrow v' : W @ l}{L \mid \Psi; \Gamma \vdash_i v \longleftrightarrow v' : V @ l} \\
\frac{L \mid \Psi; \Gamma \vdash_i S @ l \quad L \mid \Psi; \Gamma \vdash_i w : \Pi^{l,l'}(x : S).T @ l \sqcup l' \quad L \mid \Psi; \Gamma \vdash_i w' : \Pi^{l,l'}(x : S).T @ l \sqcup l' \quad L \mid \Psi; \Gamma, x : S @ l \vdash_i (w : \Pi^{l,l'}(x : S).T) x \hat{\longleftrightarrow} (w' : \Pi^{l,l'}(x : S).T) x : T @ l'}{L \mid \Psi; \Gamma \vdash_i w \longleftrightarrow w' : \Pi^{l,l'}(x : S).T @ l \sqcup l'} \\
\frac{L \mid \Psi \vdash_M \Gamma \quad L \mid \Psi; \Delta \vdash_C t : T @ l \quad t = t'}{L \mid \Psi; \Gamma \vdash_M \text{box } t \longleftrightarrow \text{box } t' : \Box(\Delta \vdash_C T @ l) @_0} \quad \frac{L \mid \Psi; \Gamma \vdash_M v \longleftrightarrow v' : \Box(\Delta \vdash_C T @ l) @_0}{L \mid \Psi; \Gamma \vdash_M v \longleftrightarrow v' : \Box(\Delta \vdash_C T @ l) @_0} \\
\boxed{L \mid \Psi; \Gamma \vdash_i v \longleftrightarrow v' : W @ l} \text{ and } \boxed{L \mid \Psi; \Gamma \vdash_i v \hat{\longleftrightarrow} v' : T @ l} \text{ for neutral terms} \\
\frac{L \mid \Psi; \Gamma \vdash_i v \hat{\longleftrightarrow} v' : T @ l \quad L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W @ l}{L \mid \Psi; \Gamma \vdash_i v \longleftrightarrow v' : W @ l} \quad \frac{L \mid \Psi \vdash_i \Gamma \quad x : T @ l \in \Gamma}{L \mid \Psi; \Gamma \vdash_i x \hat{\longleftrightarrow} x : T @ l} \\
\frac{L \mid \Psi \vdash_i \Gamma \quad u : (\Delta \vdash_{i'} T @ l) \in \Psi \quad i' \leq i \quad L \mid \Psi; \Gamma \vdash_i \delta \hat{\longleftrightarrow} \delta' : \Delta}{L \mid \Psi; \Gamma \vdash_i u^\delta \hat{\longleftrightarrow} u^{\delta'} : T[\delta] @ l} \\
\frac{L \mid \Psi; \Gamma \vdash_i S \hat{\longleftrightarrow} S' @ l \quad L \mid \Psi; \Gamma, x : S @ l \vdash_i T \hat{\longleftrightarrow} T' @ l' \quad L \mid \Psi; \Gamma \vdash_i v \longleftrightarrow v' : \Pi^{l,l'}(x : S'').T'' @ l \sqcup l' \quad L \mid \Psi; \Gamma \vdash_i s \hat{\longleftrightarrow} s' : S'' @ l}{L \mid \Psi; \Gamma \vdash_i (v : \Pi^{l,l'}(x : S).T) s \hat{\longleftrightarrow} (v' : \Pi^{l,l'}(x : S').T') s' : T[s/x] @ l'} \\
\frac{L \mid \Psi \vdash_M \Gamma \quad L \mid \Psi \vdash_D \Delta \hat{\longleftrightarrow} \Delta' \quad L \mid \Psi; \Gamma, x : \Box(\Delta \vdash_C @ l') @_0 \vdash_M M \hat{\longleftrightarrow} M' @ l}{L \mid \Psi, u : (\Delta \vdash_C @ l'); \Gamma \vdash_M t \hat{\longleftrightarrow} t' : M[\text{box } u^{\text{id}}/x] @ l \quad L \mid \Psi; \Gamma \vdash_M v \longleftrightarrow v' : \Box(\Delta \vdash_C @ l') @_0} \\
\text{left} = \text{letbox}_{x,M}^l u \leftarrow (v : \Box(\Delta \vdash_C @ l')) \text{ in } t \\
\text{right} = \text{letbox}_{x,M'}^l u \leftarrow (v' : \Box(\Delta' \vdash_C @ l')) \text{ in } t' \\
\hline
L \mid \Psi; \Gamma \vdash_M \text{left} \hat{\longleftrightarrow} \text{right} : M[t/x] @ l
\end{array}$$

Fig. 9. Selected rules for conversion checking algorithm

level  $l$ , and  $\boxed{L \mid \Psi; \Gamma \vdash_i t \hat{\longleftrightarrow} t' : T @ l}$ , which checks the convertibility of  $t$  and  $t'$  of type  $T$  on level  $l$ . Both entry points first reduce the inputs to WHNFs. Then the WHNFs are compared by  $\boxed{L \mid \Psi; \Gamma \vdash_i W \longleftrightarrow W' @ l}$  and  $\boxed{L \mid \Psi; \Gamma \vdash_i w \longleftrightarrow w' : W @ l}$ , which actually do the case analyses based on the shapes. At some point, the checking for WHNFs enters the inference mode, in order to compare neutrals. The neutral form checking algorithm for types  $\boxed{L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' @ l}$  returns the universe level  $l$  of  $V$  and  $V'$  if they are convertible. The neutral form checking algorithm for terms is a bit more complex. The actual worker is  $\boxed{L \mid \Psi; \Gamma \vdash_i v \hat{\longleftrightarrow} v' : T @ l}$ , which returns a type  $T$  and its universe level  $l$ , if  $v$  and  $v'$  are convertible. However, before returning the type  $T$  to the checking mode,  $\boxed{L \mid \Psi; \Gamma \vdash_i v \longleftrightarrow v' : W @ l}$  first reduces it to a WHNF, so the output type of this algorithm is a normal type  $W$ . Finally, the checking mode for types and terms is generalized to regular contexts and substitutions pointwise, giving  $\boxed{L \mid \Psi \vdash_D \Gamma \hat{\longleftrightarrow} \Delta}$  and  $\boxed{L \mid \Psi; \Gamma \vdash_i \delta \hat{\longleftrightarrow} \delta' : \Delta}$ . The checking mode for regular contexts is needed when we compare contextual types. We also need to check the convertibility between regular substitutions when we encounter neutral meta-variables  $u^\delta$  and  $u^{\delta'}$ , in which case we need to compare  $\delta$  and  $\delta'$ .

The conversion checking algorithm is very close to that by Abel et al. in its spirit. The checking mode for terms is type-directed. We employ a suitable check according to the shape of the input type. For functions, we check in suitably extended contexts, and for other types, we case-analyze terms accordingly. The inference mode is syntax-directed. It fails immediately if two neutral forms fail to have the same syntactic structure.

### 6.3 Second Instantiation: Conversion Checking

In the second instantiation, we assign the conversion checking algorithm to the generic equivalences:  $L \mid \Psi; \Gamma \vdash_i V \sim V' @ l$  as  $L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' @ l$ ,  $L \mid \Psi; \Gamma \vdash_i T \simeq T' @ l$  as  $L \mid \Psi; \Gamma \vdash_i T \hat{\longleftrightarrow} T' @ l$ , and  $L \mid \Psi; \Gamma \vdash_i t \simeq t' : T @ l$  as  $L \mid \Psi; \Gamma \vdash_i t \hat{\longleftrightarrow} t' : T @ l$ . The most complex case is  $L \mid \Psi; \Gamma \vdash_i v \sim v' : T @ l$ , which is assigned a conjunction of  $L \mid \Psi; \Gamma \vdash_i v \hat{\longleftrightarrow} v' : T' @ l$  and  $L \mid \Psi; \Gamma \vdash_i T \simeq T' @ l$ . Here we always take  $i \in \{\mathbb{D}, \mathbb{M}\}$ . First, the soundness lemma for the conversion checking algorithm is proved by simple mutual induction:

LEMMA 6.4 (SOUNDNESS).

- If  $L \mid \Psi; \Gamma \vdash_i T \hat{\longleftrightarrow} T' @ l$ , then  $L \mid \Psi; \Gamma \vdash_i T \simeq T' @ l$ .
- If  $L \mid \Psi; \Gamma \vdash_i t \hat{\longleftrightarrow} t' : T @ l$ , then  $L \mid \Psi; \Gamma \vdash_i t \simeq t' : T @ l$ .

The completeness lemma is established by Corollary 5.6.

LEMMA 6.5 (COMPLETENESS).

- If  $L \mid \Psi; \Gamma \vdash_i T \simeq T' @ l$ , then  $L \mid \Psi; \Gamma \vdash_i T \hat{\longleftrightarrow} T' @ l$ .
- If  $L \mid \Psi; \Gamma \vdash_i t \simeq t' : T @ l$ , then  $L \mid \Psi; \Gamma \vdash_i t \hat{\longleftrightarrow} t' : T @ l$ .

Therefore, conversion checking and syntactic equivalence are logically equivalent. To establish the decidability of convertibility, we need the following lemma, which establishes the decidability of conversion checking between reflexively convertible types or terms.

LEMMA 6.6 (DECIDABILITY OF CONVERSION CHECKING).

- if  $L \mid \Phi; \Delta \vdash_i T \hat{\longleftrightarrow} T @ l$ ,  $L \mid \Psi; \Gamma \vdash_i T' \hat{\longleftrightarrow} T' @ l$ ,  $L \vdash \Phi \simeq \Psi$  and  $L \mid \Phi \vdash_i \Delta \simeq \Gamma$ , then whether  $L \mid \Phi; \Delta \vdash_i T \hat{\longleftrightarrow} T' @ l$  is decidable.
- if  $L \mid \Phi; \Delta \vdash_i t \hat{\longleftrightarrow} t : T @ l$ ,  $L \mid \Psi; \Gamma \vdash_i t' \hat{\longleftrightarrow} t' : T @ l$ ,  $L \vdash \Phi \simeq \Psi$  and  $L \mid \Phi \vdash_i \Delta \simeq \Gamma$ , then whether  $L \mid \Phi; \Delta \vdash_i t \hat{\longleftrightarrow} t' : T @ l$  is decidable.

By using the completeness lemma, we obtain the desired decidability proof.



THEOREM 6.7 (DECIDABILITY OF CONVERTIBILITY).

- If  $L \mid \Psi; \Gamma \vdash_i T @ l$  and  $L \mid \Psi; \Gamma \vdash_i T' @ l$ , then whether  $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$  is decidable.
- If  $L \mid \Psi; \Gamma \vdash_i t : T @ l$  and  $L \mid \Psi; \Gamma \vdash_i t' : T @ l$ , then whether  $L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$  is decidable.

## 7 Related Work and Future Work

### 7.1 Normalization for Type Theories

In this paper, we follow [Abel et al. \[2018\]](#) closely and obtain the weak normalization and the decidability of convertibility theorems. The same method is used in many recent works, including [Abel et al. 2023](#); [Adjedj et al. 2024](#); [Pientka et al. 2019](#); [Pujet and Tabareau 2022, 2023](#). As opposed to the more standard Tait's approach [\[Girard 1989; Tait 1967\]](#), one big advantage of this method à la Abel et al. is that it avoids proving the Church-Rosser property, which requires much more technical setup to reason about reduction strategies. However, one main disadvantage is the lack of a strong normalization conclusion. Nevertheless, we still choose to use this method, because we can directly reason about DELAM syntactically and refer to many mechanizations [\[Abel et al. 2023; Adjedj et al. 2024; Pujet and Tabareau 2022, 2023\]](#) during the technical investigation.

In contrast to weak normalization and a verbose conversion checking algorithm, [Hu and Pientka \[2024b\]](#)'s work on simply typed layered modal type theory presents a strong normalization algorithm which returns  $\beta\eta$  normal forms, so that the conversion checking algorithm reduces to a trivial syntactic equality check of normal forms. In their work, Hu and Pientka employ a normalization-by-evaluation (NbE) proof, where they extend the classic NbE proof for STLC based on a presheaf model [\[Altenkirch et al. 1995\]](#). NbE is a technique originally by [Berger and Schwichtenberg \[1991\]](#); [Martin-Löf \[1975\]](#). An NbE proof proceeds in two steps: we first embed well-formed types and terms into a mathematical domain, and then extract normal forms from this chosen domain. The main idea of an NbE proof is to piggyback many syntactic properties like Church-Rosser of the target system onto this chosen domain, so there is no need to establish these lemmas explicitly at all. An NbE proof has many advantages, including more concise models, a more compact normalization proof, and a trivial conversion checking because of  $\beta\eta$  normal forms. [Altenkirch and Kaposi \[2016a,b, 2017\]](#) mechanize an NbE algorithm for dependent types based on CwFs [\[Dybjer 1995\]](#), which is a presheaf formulation for dependent types.

Instead of presheaf categories, another frequent style of NbE proofs is given by [Abel \[2013\]](#), where Abel chooses a defunctionalized untyped domain. In addition to the advantages above, this style is easy to mechanize and implement [\[Gratzer et al. 2019; Hu et al. 2023; Stassen et al. 2022; Wieczorek and Biernacki 2018\]](#). One possible future work for DELAM is to develop an NbE algorithm using an untyped domain model. The main difficulties are likely the modelling of code in this method. Conventionally, the untyped domain models variables in de Bruijn *levels*, while the syntax represents variables in de Bruijn *indices*. Due to recursion on code, the untyped domain must model the syntax of code faithfully. This seems to suggest that we need to model meta- and regular substitutions in the untyped domain for de Bruijn levels, leading to a cumbersome duplication. Whether this duplication is something that we must live with is left for a future investigation.

### 7.2 Meta-programming and Modalities

In type theory and programming languages, modalities are a common device for meta-programming. In Sec. 1, we have described that (contextual)  $\lambda^\square$  [\[Davies and Pfenning 2001; Nanevski et al. 2008; Pfenning and Davies 2001\]](#) is a pioneer in this direction, where meta-programming and the modal logic  $S4$  are connected by Curry-Howard correspondence. One formulation of  $\lambda^\square$  given by Davies, Pfenning and others is the dual-context style. In this style, variables are distinguished into two kinds,

regular and meta, hence introducing two different contexts to store different variables. This style is what [Hu and Pientka \[2024b\]](#) and this paper is based on. [Boespflug and Pientka \[2011\]](#) extend the dual-context style to the multi-context style, where “more meta-” variables are introduced. Combining this idea with contextual types, Moebius [\[Jang et al. 2022\]](#) supports meta-programming with intensional analysis for system F. The soundness of Moebius is justified by progress and preservation. However, since Moebius does not guarantee coverage of pattern matching on code, normalization does not hold in general. One interesting future work is to adapt layering to system F and see if we are able to obtain a normalizing theory for meta-programming in system F.

Cocon [\[Pientka et al. 2019\]](#) combines ideas from  $\lambda^\square$  and contextual types in a 2-level manner. On the lower level is logical framework (LF), in which we define the syntax of object languages. This syntax can be accessed through contextual types in MLTT on the higher level and can be intensionally analyzed. A run function can be defined in MLTT to evaluate terms from a less expressive object language than MLTT. However, this run function is not possible, if the expressive power is not strictly weaker. Semantically, Cocon is similar to DELAM in many ways. Both systems need to have multiple layers in the semantics to model sub-languages in the system. These layers all need their normalization arguments to eventually justify normalization for the whole type theory. Both systems need to model syntax to support recursion on code. Nevertheless, there are also substantial differences between them. In Cocon, object languages are user defined, so their semantics do not have any particular relation with MLTT. As a consequence, the semantics of Cocon cannot always provide a run function to embed the code from object languages into MLTT. In DELAM, on the other hand, we know the sub-language at layers  $c$  and  $d$  is exactly MLTT, so code execution becomes possible. Syntactically, code execution comes virtually for free due to the lifting lemma (Lemma 4.1). Semantically, code execution is modeled by the layering restriction lemma (Lemma 5.1). The models for different layers in DELAM also follow the matryoshka principle, saying that lower layers include strictly more information than higher layers.

Other systems using modalities for meta-programming include MetaML [\[Taha and Sheard 2000\]](#), which is a ML dialect for meta-programming without intensional analysis, 2LTT [\[Kovács 2022\]](#), which uses MLTT to compose programs in a programming language, e.g. ML, and MINT [\[Hu et al. 2023\]](#), which extends MLTT with the  $\square$  modality and serves as a program logic for MetaML.

### 7.3 Future Work

**7.3.1 DELAM in Russell-style.** In this paper, we introduced DELAM with a Tarski-style universe hierarchy. A Tarski-style universe hierarchy separates types and terms syntactically. This is also how the semantics are constructed in this paper. This initially helped in our technical development. However, Tarski style is not often how proof assistants, e.g. Agda, Coq and Lean, are implemented. Moreover, Tarski style also leads to a rather complex formulation of the mutual recursion principle for code.

In retrospect, we believe that DELAM can equally be developed using Russell style universes. A Russell-style universe hierarchy only includes one syntax for types and terms. Furthermore, it is how universes are organized in existing proof assistants. The use of Russell style universes also induces other simplifications. For one, there is only one kind of contextual types:  $\square(\Gamma \vdash_c T @ l)$ , and a code object for types simply has type  $\square(\Gamma \vdash_c \text{Ty}_l @ 1+l)$ . In addition, there is only one elimination principle for code that does not need mutual recursion:

$$\text{elim}^l (\ell, g, u_T, x_t.M) \vec{b} \quad (t : \square(\Gamma \vdash_c T @ l'))$$

Since the recursion principle is no longer mutual, it only requires one motive  $M$  on universe level  $l$  for the return type of the recursion. In this case, the whole recursor has type  $M[l'/\ell, \Gamma/g, T/u_T, t/x_t]$ .

While the recursor still requires a list of branches  $\vec{b}$ , it only contains the cases in  $\vec{b}_{\text{Trm}}$  in Fig. 2, as there is no distinction between types and terms anymore. We note that the recursion still occurs on sub-structures, so termination remains quite natural.

A detailed development of the semantics based on Russell-style universes is beyond the scope of this paper, so we leave this work for the future. We expect that the adaptation of the semantics that is described in this paper to Russell style universes is moderate and that future extensions of DELAM can benefit from the simplifications brought by Russell style.

**7.3.2 Supporting Other Features in DELAM.** In this paper, we only study DELAM as a core system. In reality, we often need more features from the type theory. For features that are already present in MLTT, e.g. inductive type families (propositional equality types, finite number types, etc.), they should be natural extensions of DELAM. Their semantics can be modeled parametrically in  $i$  and  $j$  in the Kripke logical relations as in Sec. 5.2, so that they apply to all layers.

The meta-programming layer  $M$  might also need other extensions for more practicality. It might be helpful to introduce code of regular contexts, i.e.  $\square \text{Ctx}$ , code of regular substitutions, i.e.  $\square(\Gamma \vdash_c \Delta)$ , and their recursion principles following [Cave and Pientka \[2013, 2018\]](#). This extension should be relatively simple as the logical relations already give their semantics.

**7.3.3 Other Future Work.** Another important direction is to develop an NbE algorithm for DELAM which provides a simpler normalization proof for mechanization and would be easier to mechanize. The mechanization of DELAM could be quite challenging, as we will need to handle transfinite recursion of universe levels, as well as employ good engineering practice to tackle duplication in the logical relations at different layers. Last, an implementation of DELAM, where we can actually experiment and try to understand what additional features are needed in practice would be desirable.

## 8 Conclusions

In this paper, we have introduced DELAM, a dependent layered modal type theory for meta-programming with intensional analysis. In DELAM, we delaminate MLTT into multiple layers. At layer  $c$ , code does not compute due to static code, so we can soundly recurse on code. At layer  $M$ , code is programmatically generated and is executed using lifting, which ensures that all code from layer  $c$  is also well-formed at layer  $M$ . Combining both features, we obtain a type-theoretic foundation for proof assistants, in which we can write type-safe tactics to generate proofs that are guaranteed well-formed. In addition, we have proved weak normalization for DELAM and its decidability of convertibility. We believe DELAM provides a theoretical perspective and complements our understanding of existing tactic systems. In particular, it is a significant step towards type-safe meta- and tactic programming in proof assistants.

## Acknowledgments

This work was funded by the Natural Sciences and Engineering Research Council of Canada (grant number 206263) and Fonds de recherche du Québec - Nature et Technologies (grant number 253521). The first author is funded partly by Postgraduate Scholarship - Doctoral from the Natural Sciences and Engineering Research Council of Canada and partly by Doctoral (B2X) Research Scholarship from Fonds de recherche du Québec - Nature et technologies.

## References

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

- Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP (2023), 920–954. <https://doi.org/10.1145/3607862>
- Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL (2018), 23:1–23:29. <https://doi.org/10.1145/3158111>
- Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédro, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 230–245. <https://doi.org/10.1145/3636501.3636951>
- Guillaume Allais. 2024. Scoped and Typed Staging by Evaluation. In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, January 16, 2024*, Gabriele Keller and Meng Wang (Eds.). ACM, 83–93. <https://doi.org/10.1145/3635800.3636964>
- Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical Reconstruction of a Reduction Free Normalization Proof. In *Proceedings of the 6th International Conference on Category Theory and Computer Science, CTCS 1995, Cambridge, UK, August 7-11, 1995 (Lecture Notes in Computer Science, Vol. 953)*, David H. Pitt, David E. Rydeheard, and Peter T. Johnstone (Eds.). Springer, 182–199. [https://doi.org/10.1007/3-540-60164-3\\_27](https://doi.org/10.1007/3-540-60164-3_27)
- Thorsten Altenkirch and Ambrus Kaposi. 2016a. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, Porto, Portugal, June 22-26, 2016 (LIPICs, Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:16. <https://doi.org/10.4230/LIPICs.FSCD.2016.6>
- Thorsten Altenkirch and Ambrus Kaposi. 2016b. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, Florida, USA, January 20-22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 18–29. <https://doi.org/10.1145/2837614.2837638>
- Thorsten Altenkirch and Ambrus Kaposi. 2017. Normalisation by Evaluation for Type Theory, in Type Theory. *Log. Methods Comput. Sci.* 13, 4 (2017). [https://doi.org/10.23638/LMCS-13\(4:1\)2017](https://doi.org/10.23638/LMCS-13(4:1)2017)
- Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *Proceedings of the 9th International Conference on Interactive Theorem Proving, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018 (Lecture Notes in Computer Science, Vol. 10895)*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer, 20–39. [https://doi.org/10.1007/978-3-319-94821-8\\_2](https://doi.org/10.1007/978-3-319-94821-8_2)
- Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed Lambda-calculus. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science, LICS 1991, Amsterdam, the Netherlands, July 15-18, 1991*. IEEE Computer Society, 203–211. <https://doi.org/10.1109/LICS.1991.151645>
- Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. 2022. Type Theory with Explicit Universe Polymorphism. In *28th International Conference on Types for Proofs and Programs, TYPES 2022, LS2N, University of Nantes, France, June 20-25, 2022 (LIPICs, Vol. 269)*, Delia Kesner and Pierre-Marie Pédro (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:16. <https://doi.org/10.4230/LIPICs.TYPES.2022.13>
- Mathieu Boespflug and Brigitte Pientka. 2011. Multi-level Contextual Type Theory. In *Proceedings of the 6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, FMTP 2011, Nijmegen, the Netherlands, August 26, 2011 (EPTCS, Vol. 71)*, Herman Geuvers and Gopalan Nadathur (Eds.). 29–43. <https://doi.org/10.4204/EPTCS.71.3>
- Andrew Cave and Brigitte Pientka. 2012. Programming with Binders and Indexed Data-types. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 413–424. <https://doi.org/10.1145/2103656.2103705>
- Andrew Cave and Brigitte Pientka. 2013. First-class Substitutions in Contextual Type Theory. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, FMTP 2013, Boston, Massachusetts, USA, September 23, 2013*, Alberto Momigliano, Brigitte Pientka, and Randy Pollack (Eds.). ACM, 15–24. <https://doi.org/10.1145/2503887.2503889>
- Andrew Cave and Brigitte Pientka. 2018. Mechanizing Proofs with Logical Relations - Kripke-style. *Math. Struct. Comput. Sci.* 28, 9 (2018), 1606–1638. <https://doi.org/10.1017/S0960129518000154>
- David R. Christiansen and Edwin C. Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 284–297. <https://doi.org/10.1145/2951913.2951932>
- Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (2001), 555–604. <https://doi.org/10.1145/382780.382785>
- Peter Dybjer. 1995. Internal Type Theory. In *International Workshop on Types for Proofs and Programs, TYPES 1995, Torino, Italy, June 5-8, 1995, Selected Papers (Lecture Notes in Computer Science, Vol. 1158)*, Stefano Berardi and Mario Coppo (Eds.). Springer, 120–134. [https://doi.org/10.1007/3-540-61780-9\\_66](https://doi.org/10.1007/3-540-61780-9_66)

- Peter Dybjer and Anton Setzer. 2003. Induction-recursion and Initial Algebras. *Ann. Pure Appl. Log.* 124, 1-3 (2003), 1–47. [https://doi.org/10.1016/S0168-0072\(02\)00096-9](https://doi.org/10.1016/S0168-0072(02)00096-9)
- Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 34:1–34:29. <https://doi.org/10.1145/3110278>
- Jean-Yves Girard. 1989. *Proofs and Types*. Number 7 in Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge [England] ; New York. <https://dl.acm.org/doi/10.5555/64805>
- Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing A Modal Dependent Type Theory. *Proc. ACM Program. Lang.* 3, ICFP (2019), 107:1–107:29. <https://doi.org/10.1145/3341711>
- Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184. <https://doi.org/10.1145/138027.138060>
- Jason Z. S. Hu. 2024. *Foundations and Applications of Modal Type Theories*. PhD Thesis. McGill University, Montréal, Canada.
- Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. 2023. Normalization by Evaluation for Modal Dependent Type Theory. *J. Funct. Program.* 33 (2023). <https://doi.org/10.1017/S0956796823000060>
- Jason Z. S. Hu and Brigitte Pientka. 2024a. DeLaM: A Dependent Layered Modal Type Theory for Meta-programming. *CoRR* abs/2404.17065 (2024). arXiv:2404.17065 <https://doi.org/10.48550/arXiv.2404.17065>
- Jason Z. S. Hu and Brigitte Pientka. 2024b. Layered Modal Type Theory: Where Meta-programming Meets Intensional Analysis. In *Proceedings of the 33rd European Symposium on Programming on Programming Languages and Systems, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Part I (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 52–82. [https://doi.org/10.1007/978-3-031-57262-3\\_3](https://doi.org/10.1007/978-3-031-57262-3_3)
- Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: Metaprogramming using Contextual Types: The Stage Where System F Can Pattern Match on Itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. <https://doi.org/10.1145/3498700>
- Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: Typed Tactics for Backward Reasoning in Coq. *Proc. ACM Program. Lang.* 2, ICFP (2018), 78:1–78:31. <https://doi.org/10.1145/3236773>
- András Kovács. 2022. Staged Compilation with Two-level Type Theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. <https://doi.org/10.1145/3547641>
- Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium 1973*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73–118. [https://doi.org/10.1016/S0049-237X\(08\)71945-1](https://doi.org/10.1016/S0049-237X(08)71945-1)
- Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. <https://doi.org/10.1145/1352582.1352591>
- Erik Palmgren. 1998. On Universes in Type Theory. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press. <https://doi.org/10.1093/oso/9780198501275.003.0012>
- Frank Pfenning and Rowan Davies. 2001. A Judgmental Reconstruction of Modal Logic. *Math. Struct. Comput. Sci.* 11, 4 (2001), 511–540. <https://doi.org/10.1017/S0960129501003322>
- Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 199–208. <https://doi.org/10.1145/53990.54010>
- Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, 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 2019, Vancouver, British Columbia, Canada, June 24-27, 2019*. IEEE, 1–13. <https://doi.org/10.1109/LICS.2019.8785683>
- Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now for Good. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. <https://doi.org/10.1145/3498693>
- Loïc Pujet and Nicolas Tabareau. 2023. Impredicative Observational Equality. *Proc. ACM Program. Lang.* 7, POPL (2023), 2171–2196. <https://doi.org/10.1145/3571739>
- Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *J. Autom. Reason.* 64, 5 (2020), 947–999. <https://doi.org/10.1007/s10817-019-09540-0>
- Philipp Stassen, Daniel Gratzer, and Lars Birkedal. 2022. mitten: A Flexible Multimodal Proof Assistant. In *28th International Conference on Types for Proofs and Programs, TYPES 2022, LS2N, University of Nantes, France, June 20-25, 2022 (LIPIcs, Vol. 269)*, Delia Kesner and Pierre-Marie Pédro (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:23. <https://doi.org/10.4230/LIPIcs.TYPES.2022.6>
- Walid Taha and Tim Sheard. 2000. MetaML and Multi-stage Programming with Explicit Annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. [https://doi.org/10.1016/S0304-3975\(00\)00053-0](https://doi.org/10.1016/S0304-3975(00)00053-0)
- William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. <https://doi.org/10.2307/2271658>

- Paul van der Walt and Wouter Swierstra. 2012. Engineering Proof by Reflection in Agda. In *24th International Symposium on Implementation and Application of Functional Languages, IFL 2012, Oxford, UK, August 30-September 1, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8241)*, Ralf Hinze (Ed.). Springer, 157–173. [https://doi.org/10.1007/978-3-642-41582-1\\_10](https://doi.org/10.1007/978-3-642-41582-1_10)
- Makarius Wenzel et al. 2024. The Isabelle/Isar Reference Manual. <https://isabelle.in.tum.de/doc/isar-ref.pdf>
- Pawel Wieczorek and Dariusz Biernacki. 2018. A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, California, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 266–279. <https://doi.org/10.1145/3167091>
- Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A Monad for Typed Tactic Programming in Coq. *J. Funct. Program.* 25 (2015). <https://doi.org/10.1017/S0956796815000118>

Received 2024-07-08; accepted 2024-11-07