Layered Modal Type Theory

Where Meta-programming Meets Intensional Analysis

Jason Z. S. Hu(✉) and Brigitte Pientka

School of Computer Science, McGill University, Montréal, QC, Canada H3A 0E9
zhong.s.hu@mail.mcgill.ca bpientka@cs.mcgill.ca

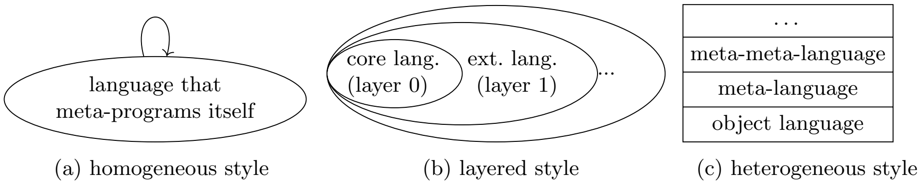
Abstract. We introduce layering to modal type theory to combine type theory with intensional analysis. In particular, we demonstrate this idea by developing a 2-layered modal type theory. At the core of this type theory (layer 0) is a simply typed λ-calculus with no modality. Layer 1 is obtained by extending the core language with one layer of contextual $\square$ types to support pattern matching on potentially open code from layer 0 while retaining normalization. Although both layers fundamentally share the same language and the same typing judgment, we only allow computation at layer 1. As a consequence, layer 0 accurately captures the syntactic representation of code in contrast to the computational behaviors at layer 1. The system is justified by normalization by evaluation (NbE) using a presheaf model. The normalization algorithm extracted from the model is sound and complete and is implemented in Agda.
Layered modal type theory provides a uniform foundation for meta-programming with intensional analysis. We see this work as an important step towards a foundational way to support meta-programming in proof assistants.

Keywords: modal type theory · contextual types · meta-programming · normalization by evaluation· presheaf model

1 Introduction

For the past decades, the problem of combining type theory and meta-programming has been in need for a solution (c.f. [57,15,18,36,47,50,7]). Given the solid and elegant foundations for describing proofs as programs provided by type theories, also supporting meta-programming allows us to think of proof generation as code generation. This opens up the possibility to support proof macros, domain-specific proof generators, proof transformations, and reasoning about meta-programs within the same language.

While support for meta-programming in existing proof assistants is common (e.g. [18,15,61,57]), this is typically achieved via some unverified mechanisms like reflection, requiring significant engineering effort. Moreover, the interplay between these mechanisms and the core type theory is not well-understood, often breaks critical type-theoretic properties like confluence, and lacks theoretical guarantees like normalization. As a consequence, it is often not clear how we



(a) homogeneous style (b) layered style (c) heterogeneous style

Fig. 1: Layered style as a middle ground

can reason about meta-programs themselves. Even guaranteeing that the generated code is well-typed and well-scoped is non-trivial. Hence this leads to a gap between implementations of meta-programming in proof assistants and their theoretical foundations.

Theoretical foundations that combine meta-programming with type-theory typically fall into two categories: the homogeneous style and the heterogeneous style. Homogeneous meta-programming uses a single language capable of meta-programming itself (depicted in Fig. 1a). To provide a logical, type-safe foundation in this style, Davies and Pfenning [17] give a modal λ-calculus with the $\square$ modality. They use the modal type $\square T$ to represent the code of type T. Having modal types allows us to differentiate on the type level meta-programs that manipulate code from regular programs in one unified language. Nanevski et al. [39] subsequently extend the modal λ-calculus [17] with contextual types, allowing meta-programming on open code. Nevertheless, the correspondence described by both systems only supports basic primitives like execution and composition of code, but does not suggest a way to support any form of intensional analysis. In fact, supporting intensional analysis in the homogeneous style while retaining properties like confluence and normalization has been fraught with difficulties (c.f. [48]). Most recently, Kavvos [33] notes that we can only soundly extend the modal λ-calculus with intensional analysis for *closed* code if we want to retain confluence. A significant step towards supporting pattern matching on open code in a homogeneous style is taken in Moebius [30]. Moebius is based on System F-style polymorphism. However, its pattern matching does not guarantee coverage. Therefore Moebius does not provide normalization.

In a heterogeneous system, we distinguish between the meta-language and the object language (illustrated by Fig. 1c). Recently, Kovács [36] adapts 2-level type theory (2LTT), originally conceived for homotopy type theory, to dependently typed meta-programming. Here, a dependently typed meta-language sits on top of a less expressive object language. However, this type theory does not support intensional analysis. In contrast, Cocon [47], another 2-level type theory following in the footsteps of previous work [17,39], supports modeling open code and intensional code analysis. Though these heterogeneous systems are modular, this comes at a price: a definition in one level is not directly accessible or reused in the other level. Unlike homogeneous systems, both heterogeneous systems do not support execution of code. Moreover, the separation into two languages leads to two separate investigations of meta-theoretic properties for two languages and ultimately two separate normalization arguments. How to elegantly scale these languages to multiple layers is not obvious, or at least very tedious.

In this paper, we propose a novel *layered* style as a schema to combine meta-programming and type theory (see Fig. 1b) and to combine the advantages of homogeneous and heterogeneous styles. Specifically, our layered modal type theory achieves three features: ① a *run* primitive, which extracts a term of type A given code of type A for all A; ② a *normalizing* type theory; ③ pattern matching on code, which is the most general form of intensional analysis. As a demonstration, we develop a layered modal simple type theory achieving these features. In this type theory, there are a fixed number of layers of languages. The type theory is uniform in the sense that all layers fundamentally *share* a common syntax for their languages and the same typing judgment as in the homogeneous style. Therefore, our layered system has a natural *run* primitive as all homogeneous systems. Furthermore, our layered system follows the matryoshka principle: the language at layer i is *contained* in its meta-language at layer $i + 1$. What is added to layer i at layer $i + 1$ is the ability to inspect and analyze code from the language at layer i. This matryoshka structure of layers of languages not only ensures uniformity in the syntax and the typing judgment of the type theory, but also provides extra flexibility in distinguishing computational behaviors at different layers. As a principle, we only allow β and η equivalence at the highest layer, so all lower layers are treated as static code which is only identified by its syntax. Layering allows us to encode different computational behaviors at different layers using the same set of equivalence rules. This is crucial to enable sound intensional analysis and establish normalization.

To introduce layering succinctly, we focus on a 2-layered modal simple type theory in this paper. In this 2-layered system, its core language at layer 0 is a simply typed λ-calculus (STLC). At layer 1, STLC is then extended with one layer of meta-programming with the $\square$ modality. The meta-language at layer 1 can only manipulate and analyze code from layer 0, but *not* from its own layer. Following our previous discussion, we only allow computation on layer 1, and terms at layer 0 are treated as pure syntax. This allows us to cleanly define covering pattern matching on code and eventually leads to an elegant normalization proof using a presheaf model.

Summary of Contributions:

1. We develop a 2-layered modal type theory (Sec. 3) which supports running code (feature ①). To prove normalization, we extend the classic presheaf model for STLC [5] to our type theory (Sec. 4). From this presheaf model, we extract its normalization algorithm that is complete and sound.
2. We extend the previous 2-layered modal type theory with pattern matching on code (Sec. 5). We adapt our previous presheaf model to support pattern matching on code and prove that the extracted algorithm is both complete and sound. Thus we achieve features ② and ③.
3. We outline three different dimensions to extend layered modal type theory in Sec. 6. In particular, we discuss extensions to richer systems like System F and Martin-Löf type theory. We also discuss how to extend the expressive power of the computational layer with additional operations, and how to scale our 2-layered system to n layers.

We believe that layering is versatile enough to be adapted to complex systems like System F and Martin-Löf type theory. As such, it provides a systematic way of supporting intensional analysis while retaining normalization. It is a significant step towards closing the gap between implementations that support meta-programming in practice and their theoretical foundations. Interested readers could find more details in our technical report [27] and our Agda code [28].

2 Example Programs in 2-layered Modal Type Theory

In this section, we show how to write and improve the well-known `power` function in layered modal type theory by gradually introducing more features. In general, many common meta-programs including the `power` function use only two layers.

2.1 A Layered Power Function

The `power` function defined by [17, Sec. 3.4] is a classic meta-program and we can define it in our 2-layered type theory with the help of contextual types:

```
power : Nat → □ (x : Nat ⊢ Nat)
power zero     = box (x. 1)
power (succ n) = letbox u ← power n in box (x. u[x/x] * x)
```

In the examples in this section, we use a front-end syntax similar to Haskell and Agda. For clarity, we abbreviate `succ ... (succ zero)` as numbers, e.g. `1` is notation for `succ zero`. The return type of this meta-function is a contextual type `□ (x : Nat ⊢ Nat)`. This type denotes code of type `Nat` with an open variable `x` of type `Nat`. In general, the number of open variables is arbitrary. In the body, we recurse on the input number. If it is `zero`, then the generated code is just `1`. The open variable `x` is not used. In the `succ` case, we first perform the recursive call `power n`. The eliminator `letbox` binds a new *global variable* `u` to an open type `(x : Nat ⊢ Nat)`. We say that `u` has type `Nat` with an open variable `x` of type `Nat`. A global variable is a placeholder for code. It remains visible under a `box` constructor. Regular variables like `n`, on the other hand, cannot directly participate in code construction, so they are hidden inside `box`. When we refer to `u` in `box`, we must instantiate the open variable `x` of `u`. In this case, an identity substitution `[x/x]` suffices. Now `u[x/x]` stands for the `n`'th power of `x` and we obtain our goal by multiplying it with an extra `x`. Our implementation of the `power` function is almost as expected except for the dangling `1`:

```
power 1 = box (x. 1 * x)        power 2 = box (x. (1 * x) * x)
```

We would like to remove the `1`'s because it is the unit element of multiplication. We will make this improvement in the next subsection. Nevertheless, we can already *run* the current code, which is critical for a meta-programming system:

```
letbox u ← power 2 in λ x. u[x/x] : Nat → Nat
```

generates a regular function computing squares. We can also directly run the code with a specific argument:

```
letbox u ← power 2 in u[5/x] = 25
```

would substitute `5` for `x` and give `25`, the square of `5`.

2.2 Pattern Matching for Intensional Analysis

An easy way to improve the previous implementation is to *pattern match* on the resulting code and remove all occurrences of `1`. However, supporting pattern matching on code in a type-theoretic setting has been notoriously difficult. Previous attempts in the homogeneous style fail to retain the normalization property. To illustrate, consider the intensional `isapp` function [33,19]. This function simply looks at the structure of a code and returns `true` if this code is a function application, or `false` otherwise. Note that `isapp`'s behavior purely depends on the syntactic structure of its argument. In our 2-layered system, this function can be implemented by a pattern matching on code:

```
isapp : □ ( ⊢ Nat) → Bool
isapp x = match x with | ?u ?u' ⇒ true | _ ⇒ false
```

We use pattern matching to inspect the input code `x`. In our first branch, we return `true` if `x` is some function application. Here, `?u` and `?u'` are both *pattern variables*. We use question marks to distinguish pattern variables and constants, e.g. `zero` and `succ` which are the constructors of `Nat`. This distinction is only necessary in the patterns, and we do not write a question mark when we refer to a pattern variable in the body of the branch. We also omit writing the local context in which the pattern is sensible because it is determined by the type of `x`. The pattern variables `u` and `u'` capture the code of the function and the argument respectively if `x` is a function application. As they are not used, we could also have written `_ _` instead. The other branches are captured by the wildcard and all return `false`. Let us see how this function behaves:

```
isapp (box ((λ x. x) 10)) = true
isapp (box 10)            = false
```

Kavvos [33] points out that Gabbay and Nanevski's [19] evaluation of `isapp` is not confluent. It is possible to evaluate the same program in different orders and obtain two different values. For some well-typed code `t` and `s`,

```
  letbox u ← box (t s) in isapp (box u)
= isapp (box (t s))                      = true
  letbox u ← box (t s) in isapp (box u)
= letbox u ← box (t s) in false          = false
```

In the second execution, `isapp (box u)` is evaluated first, and then the overall result is `false`. In our system, this confluence issue is avoided by preventing the execution of `isapp (box u)` until it is known what `u` stands for. This treatment ensures that `isapp` is stable under substitutions. Hence, the program only evaluates to `true`. This is a subtle but critical design decision which ultimately enables sound intensional analysis and normalization. We explain more in Sec. 5.2.

With sound pattern matching on code, a simple arithmetic simplifier is implemented to remove the redundant `1`'s in the previous subsection:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp y = match y with
         | 1 * ?u    ⇒ box (x. u[x/x])
         | ?u * ?u' ⇒ letbox u1 = simp (box (x. u[x/x]))
                      in box (x. u1[x/x] * u'[x/x])
         | _         ⇒ y
```

In the first case, we remove `1` from the multiplication. In the second case, we recursively simplify the first factor. We know this is sufficient because `1` only occurs in the leftmost factor. In the last case, we do not optimize. Since pattern matching is covering, we must either specify all cases or give a wildcard case. At last, we provide a wrapper function `power'`, where we invoke `simp` to simplify the code generated by `power`:

```
power' : Nat → □ (x : Nat ⊢ Nat)
power' n = simp (power n)
```

The `power'` function precisely does what we expect:

```
power' 1 = box (x. x)              power' 2 = box (x. x * x)
```

This example shows that we have full control over code via pattern matching on code, while running `power'` still gives the same behaviors as `power`.

3 A 2-Layered Modal Type Theory

In this section, we introduce a 2-layered modal type theory, which is simple yet powerful enough for many interesting programs like the unoptimized `power` function in the previous example. This system provides a starting point and a basis for a clear understanding of the impact of layering on syntax and semantics. We build a semantic framework for 2-layered modal type theory which is further extended with pattern matching on code in Sec. 5.

2-layered modal type theory is defined as follows:

$$
\begin{array}{llll}
S, T & := & \mathtt{Nat} \mid \Box(\Gamma \vdash T) \mid S \longrightarrow T & \text{(Types, } \mathsf{Typ}\text{)} \\
x, y & & & \text{(Local variables)} \\
u & & & \text{(Global variables)} \\
s, t & := & x \mid u^{\delta} \mid \mathsf{zero} \mid \mathsf{succ}\ t \mid \mathsf{rec}_T\ s\ (x\ y.s')\ t & \text{(Terms, } \mathsf{Exp}\text{)} \\
 & & \mid \mathtt{box}\ t \mid \mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t \mid \lambda x.t \mid s\ t & \\
\delta & := & \cdot \mid \delta, t/x & \text{(Local substitutions)} \\
\Gamma, \Delta & := & \cdot \mid \Gamma, x : T & \text{(Local contexts)} \\
\Phi, \Psi & := & \cdot \mid \Phi, u : (\Gamma \vdash T) & \text{(Global contexts)}
\end{array}
$$

We assume de Bruijn indices as our name representation for convenience but our development generalizes. We use natural numbers `Nat` as a base type. We can construct zero and succ of another `Nat`. $\mathsf{rec}_T\ s\ (x\ y.s')\ t$ is the recursor for `Nat`, where t is the scrutinee, s is the base case and s' is the step case, where x is the predecessor and y is the result from the recursive call. As the recursor for natural numbers is standard, we leave its discussion in the technical report [27].

A function is introduced by λ-abstraction and can be applied to an argument. $\square(\Gamma \vdash T)$ is a contextual type. It stands for code open in context Γ. The `box` constructor introduces terms of type $\square(\Gamma \vdash T)$ and `letbox` is the eliminator for it. We defer our discussion on pattern matching on code to Sec. 5.

For layered systems, we keep track of as many contexts as the layers. These contexts are contained in a fixed-sized *context array* in the judgments. With two layers, a context array only has two contexts $\Phi; \Gamma$. It hence defines a dual-context type theory. Following Pfenning and Davies [42,17], Γ is referred to as a *local context* and its variables are *local variables*, ranged over by x and y. Φ is a *global context* and contains *global variables*, ranged over by u. For a global binding $u : (\Gamma \vdash T)$, we say that u represents code of type T with an open context Γ.

When writing meta-programs, we conceptually distinguish between programs that are dynamic and compute, and code that is static and syntactic. In a homogeneous system, this distinction is captured by types, i.e. program t has type T while code has type $\square(\Gamma \vdash T)$. However, a term t itself does not provide information about whether it is inside of a `box` (hence treated as code), or outside of a `box` (hence a program). For example, only knowing that `succ zero` has type `Nat` does not reveal whether it is a piece of code or a program. The typing judgment for homogeneous systems like $\Psi; \Gamma \vdash t : T$ [42,17] only provides typing information, and does not a priori determine whether t should be considered as code or as a program. Even though one major advantage of a homogeneous system is to use the same language for code and programs, this lack of information is the critical reason for the challenges that we face when combining type theory and intensional analysis.

Layered modal type theory makes the distinction between code and programs explicit. In the typing judgment $\Psi; \Gamma \vdash_i t : T$, we use the subscript $i \in [0, 1]$ to identify the layer at which t is well-typed. This judgment states that the term t has type T *at layer* i. When $i = 0$, t is code and does not compute, and when $i = 1$, t is a program and therefore has rich reduction behaviors. There are three important implications of layering:

1. we can control what types are valid at each layer,
2. we can control what terms are well-typed at each layer, and
3. we can control what terms are equivalent at each layer.

In the first part, we control the validity of types using the validity predicate. In the rules below, we rule out the use of $\square$ at layer 0 and limit layer 1 to at most one layer of $\square$:

$$\frac{}{\texttt{Nat wf}^i} \qquad \frac{S\ \texttt{wf}^i \qquad T\ \texttt{wf}^i}{S \longrightarrow T\ \texttt{wf}^i} \qquad \frac{\Gamma\ \texttt{wf}^0 \qquad T\ \texttt{wf}^0}{\square(\Gamma \vdash T)\ \texttt{wf}^1}$$

This validity predicate only limits the depth of nested $\square$s. Therefore, $(\square(\cdot \vdash \texttt{Nat}) \to \square(\cdot \vdash \texttt{Nat}))\ \texttt{wf}^1$ holds although it has two $\square$s. $\square(\cdot \vdash \square(\cdot \vdash \texttt{Nat}))\ \texttt{wf}^1$ does not hold, because it has two nested layers of $\square$. Moreover, the validity judgment only provides an upper bound, so both $\texttt{Nat wf}^0$ and $\texttt{Nat wf}^1$ hold. This predicate generalizes to $\Psi\ \texttt{wf}^i$ and $\Gamma\ \texttt{wf}^i$ by requiring all types in Ψ and Γ

$\boxed{\Psi; \Gamma \vdash_i t : T}$ and $\boxed{\Psi; \Gamma \vdash_i \delta : \Delta}$ Term t and local substitution δ are well-typed, respectively, in context Ψ and Γ at layer i where $i \in [0, 1]$

$$\frac{\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i}{\Psi; \Gamma \vdash_i \cdot : \cdot} \qquad \frac{\Psi; \Gamma \vdash_i \delta : \Delta \quad \Psi; \Gamma \vdash_i t : T}{\Psi; \Gamma \vdash_i \delta, t/x : \Delta, x : T} \qquad \frac{\Psi; \Gamma \vdash_i \delta : \Delta \quad u : (\Delta \vdash T) \in \Psi}{\Psi; \Gamma \vdash_i u^\delta : T}$$

$$\frac{\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i \quad x : T \in \Gamma}{\Psi; \Gamma \vdash_i x : T} \qquad \frac{\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i}{\Psi; \Gamma \vdash_i \mathsf{zero} : \mathtt{Nat}} \qquad \frac{\Psi; \Gamma \vdash_i t : \mathtt{Nat}}{\Psi; \Gamma \vdash_i \mathsf{succ}\ t : \mathtt{Nat}}$$

$$\frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T} \qquad \frac{\Psi; \Gamma \vdash_i t : S \longrightarrow T \quad \Psi; \Gamma \vdash_i s : S}{\Psi; \Gamma \vdash_i t\ s : T}$$

$\boxed{\Psi; \Gamma \vdash_i t \approx t' : T}$ Term t and t' are equivalent in contexts Ψ and Γ at layer i

$$\frac{\Psi; \Gamma, x : S \vdash_1 t : T \quad \Psi; \Gamma \vdash_1 s : S}{\Psi; \Gamma \vdash_1 (\lambda x.t)\ s \approx t[s/x] : T} \qquad \frac{\Psi; \cdot \vdash_0 s : T \quad \Psi, u : T; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathtt{letbox}\ u \leftarrow \mathtt{box}\ s\ \mathtt{in}\ t \approx t[s/u] : T'}$$

$$\frac{\Psi; \Gamma \vdash_1 t : S \longrightarrow T}{\Psi; \Gamma \vdash_1 t \approx \lambda x.(t\ x) : S \longrightarrow T}$$

Fig. 2: Typing and equivalence judgments

to comply with the predicate. The validity predicates satisfy the lifting property:

Lemma 1 (Type lifting). *If T wf^0, then T wf^1; if Γ wf^0, then Γ wf^1.*

The lifting property characterizes the matryoshka principle for types and the diagram in Fig. 1b, and says that types and contexts at a lower layer are included at a higher layer.

The fact that the validity predicate only allows $\square$ at layer 1, suggests that its constructor `box` and eliminator `letbox` should also only appear at layer 1, while terms of types `Nat` and functions should appear at both layers. Having a layer in the typing judgment allows us to cleanly restrict valid terms at each layer:

$$\frac{\Gamma\ \mathtt{wf}^1 \quad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma \vdash_1 s : \square(\Delta \vdash T) \quad \Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t : T'}$$

`box` t is well-typed at layer 1, only if the code t is well-typed at layer 0. Now a clear line is drawn between code and programs: code lives at layer 0 while programs live at layer 1. The rule for `letbox` is only available at layer 1. The body is type-checked in an extended global context with a new global variable. This global variable is a placeholder for the code computed by s.

The rules are given in Fig. 2. The rules for terms coming from STLC, i.e. zero, succ, λ and function applications, are standard and valid at both layers. Given a term of type `Nat` or a function, we know whether it is code or a program by checking the layer it lives at. Extra validity predicates are added to the premises of the local variable rule and the zero rule to enforce the coherence between terms and types at layer i. Notice that terms from STLC can extend the local context via λ regardless of layers and they can only refer to but not introduce global variables. When referring to a global variable u, a local substitution δ is needed

to replace all variables in the local context Δ, as specified by the superscript. The coherence between terms and types requires terms at layer i to have types at the same layer. This criterion is formulated by the following lemma:

Lemma 2 (Syntactic validity). *If* $\Psi; \Gamma \vdash_i t : T$*, then* Ψ $\mathtt{wf}^0$*,* Γ $\mathtt{wf}^i$ *and* T $\mathtt{wf}^i$ *for* $i \in [0, 1]$.

Ψ is always valid at layer 0 because it is a context for code from layer 0.

The layer i in the typing judgment $\Psi; \Gamma \vdash_i t : T$ effectively leads to the encapsulation of two languages in the same system. When $i = 0$, only terms in STLC are well-typed, so we work in STLC, and hence Ψ $\mathtt{wf}^0$ and Γ $\mathtt{wf}^0$ hold. The typing rules ensure that we cannot write any meta-program at this layer and no $\square$ is involved. When $i = 1$, one layer of $\square$ is allowed in addition to STLC. At this layer, we can not only write regular STLC programs, but also write meta-programs that generate STLC programs through $\square$. Thus, we work with a meta-language and an extension of STLC. In this case, Ψ $\mathtt{wf}^0$ and Γ $\mathtt{wf}^1$ hold. Using layers, we fit both code (layer 0) and programs (layer 1) in a unified set of typing rules and arrive at a middle ground between homogeneous and heterogeneous styles. Code at layer 0 can be lifted to layer 1 and turned into a program. The resulting program is well-typed, due to the following lemma:

Lemma 3 (Term lifting). *If* $\Psi; \Gamma \vdash_0 t : T$*, then* $\Psi; \Gamma \vdash_1 t : T$.

The lifting property of well-typed terms has two indications. ① A language at layer 0 is *contained* at layer 1. This is the critical intuition of the matryoshka principle and the idea of layering. ② Though a term at layer 0 is code and static, its computational behaviors are recovered by lifting it to layer 1. The second point is what guarantees a universal *run* primitive for all code that is crucial for a meta-programming system and achieves the feature ① in Sec. 1. The term lifting behavior can be trigger by the β rule for $\square$. For some well-typed terms t and s at layer 0 and a local substitution δ that does not refer to u:

$$\mathtt{letbox}\ u \leftarrow \mathtt{box}\ (\lambda x.t)\ s\ \mathtt{in}\ u^\delta \approx ((\lambda x.t)\ s)[\delta] \approx t[s/x][\delta]$$

Due to the β rule, u is replaced by $(\lambda x.t)\ s$. The layer-0 term $(\lambda x.t)\ s$ is then lifted to layer 1 on the right hand side and computes. Thus its computational behavior is revived and it is further reduced to $t[s/x]$.

At last, due to layering in the typing rules, the equivalence rules are also layered. There are three groups of equivalence rules: the PER rules which include symmetry and transitivity, congruence rules which are naturally derived from the typing rules, and the computation rules which describe β and η equivalence. The PER and congruence rules apply to all layers, but the computation rules only apply to layer 1. We show all the β and η rules at the bottom of Fig. 2. The PER and congruence rules are standard. $[s/x]$ and $[s/u]$ are local and global substitutions, respectively. They substitute s for x and for u everywhere as expected. The lack of computation at layer 0 ensures that terms at layer 0 are identified *only* by their syntactic structures and indeed behave as code:

Lemma 4 (Static code). *If* $\Psi; \Gamma \vdash_0 t \approx s : T$*, then* $t = s$.

$$\frac{}{\varepsilon : \cdot \Longrightarrow_g \cdot} \qquad \frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \Gamma\ \mathtt{wf}^0 \qquad T\ \mathtt{wf}^0}{q(\gamma) : \Psi, u : (\Gamma \vdash T) \Longrightarrow_g \Phi, u : (\Gamma \vdash T)}$$

$$\frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \Gamma\ \mathtt{wf}^0 \qquad T\ \mathtt{wf}^0}{p(\gamma) : \Psi, u : (\Gamma \vdash T) \Longrightarrow_g \Phi} \qquad \frac{}{\varepsilon : \cdot \Longrightarrow_l \cdot} \qquad \frac{\tau : \Gamma \Longrightarrow_l \Delta \qquad T\ \mathtt{wf}^1}{q(\tau) : \Gamma, x : T \Longrightarrow_l \Delta, x : T}$$

$$\frac{\tau : \Gamma \Longrightarrow_l \Delta \qquad T\ \mathtt{wf}^1}{p(\tau) : \Gamma, x : T \Longrightarrow_l \Delta} \qquad \frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \tau : \Gamma \Longrightarrow_l \Delta}{\gamma; \tau : \Psi; \Gamma \Longrightarrow \Phi; \Delta}$$

Fig. 3: Global and local weakenings

This lemma justifies our treatment of terms at layer 0 as code and prepares for the addition of pattern matching on code in Sec. 5.

Finally, we specify global substitutions between global contexts. Global substitutions are defined in the usual way as lists of terms:

$$\sigma := \cdot \mid \sigma, t/u \qquad \text{(Global substitutions)}$$

Due to layering, all terms in a global substitution must live at layer 0:

$$\frac{\Psi\ \mathtt{wf}^0}{\Psi \vdash \cdot : \cdot} \qquad \frac{\Psi \vdash \sigma : \Phi \qquad \Psi; \Gamma \vdash_0 t : T}{\Psi \vdash \sigma, t/u : \Phi, u : (\Gamma \vdash T)}$$

Given a global substitution σ, we can apply it to a term:

$$\begin{array}{rcll}
x[\sigma] & := & x & \\
u^\delta[\sigma] & := & \sigma(u)[\delta[\sigma]] & \text{(lookup } u \text{ in } \sigma) \\
\mathsf{zero}[\sigma] & := & \mathsf{zero} & \\
\mathsf{succ}\ t[\sigma] & := & \mathsf{succ}\ (t[\sigma]) & \\
\lambda x.t[\sigma] & := & \lambda x.(t[\sigma]) & \\
s\ t[\sigma] & := & (s[\sigma])\ (t[\sigma]) & \\
\mathtt{box}\ t[\sigma] & := & \mathtt{box}\ (t[\sigma]) & \\
\mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t[\sigma] & := & \mathtt{letbox}\ u \leftarrow s[\sigma]\ \mathtt{in}\ (t[\sigma, u/u]) &
\end{array}$$

where $\delta[\sigma]$ applies σ to all terms in δ. Global substitutions do not handle local variables, so in the case of local variables we just return x, while in the case of global variables we look up σ and apply the globally substituted local substitution $\delta[\sigma]$ to the result of the lookup. σ propagate in most cases recursively. In the case of `letbox`, we extend the substitution and apply $\sigma, u/u$ to the body t. Global substitutions compose and have identity. We write $\Psi \vdash \mathsf{id}_\Psi : \Psi$, and often omit the subscript whenever it can be inferred.

4 Presheaf Model and Normalization by Evaluation

We now establish normalization by evaluation (NbE) [37,11,1] of the 2-layered modal type theory. NbE is a technique to establish the normalization property. An NbE proof usually proceeds in two steps: first, we evaluate terms of a type

$$\begin{aligned} [\![_]\!] &: \mathsf{Typ} \to \mathcal{W}^{op} \Longrightarrow \mathcal{S}et \\ [\![\mathtt{Nat}]\!] &:= \mathsf{Nf}^{\mathtt{Nat}} \\ [\![\square(\Gamma \vdash T)]\!] &:= \mathsf{Nf}^{\square(\Gamma\vdash T)} \\ [\![S \longrightarrow T]\!] &:= [\![S]\!] \hat{\longrightarrow} [\![T]\!] \end{aligned} \qquad \begin{aligned} [\![\Phi]\!]^0_\Psi &:= \{\gamma \mid \gamma : \Psi \Longrightarrow_g \Phi\} \\ [\![\Phi]\!]^1_\Psi &:= \{\sigma \mid \Psi \vdash \sigma : \Phi\} \\ [\![\cdot]\!]_{\Psi;\Gamma} &:= \{*\} \\ [\![\Delta, x : T]\!]_{\Psi;\Gamma} &:= [\![\Delta]\!]_{\Psi;\Gamma} \times [\![T]\!]_{\Psi;\Gamma} \end{aligned}$$

Fig. 4: Interpretations of types, and global and local contexts

theory into some chosen domain; second, normal forms are extracted from values in this domain. Our chosen domain is a presheaf category. A presheaf category is a functor category from some base category to the category of sets. A carefully chosen base category leads to an intuitive normalization proof. In this section, we use the category of weakenings as the base category. The presheaf model shown here is a moderate extension of the classic presheaf model of STLC [5].

4.1 Category of Weakenings

In the category of weakenings, the objects are the dual contexts and morphisms are weakenings between dual contexts. Weakenings between dual contexts are just tuples of global and local weakenings. They individually are defined in the same way as weakenings in STLC as below:

$$\gamma := \varepsilon \mid q(\gamma) \mid p(\gamma) \quad \text{(Global weakenings)} \quad \tau := \varepsilon \mid q(\tau) \mid p(\tau) \quad \text{(Local weakenings)}$$

Their typing rules are virtually identical with the only difference in the validity predicates (Fig. 3). The q constructor extends a weakening with the same type, while p actually weakens the context. $\Psi \Longrightarrow_g \Phi$ denotes global weakenings and $\Gamma \Longrightarrow_l \Delta$ denotes local weakenings. Then weakenings of dual contexts $\gamma;\tau : \Psi;\Gamma \Longrightarrow \Phi;\Delta$ are tuples of global and local weakenings. Both global and local weakenings have composition and identity. We write id_Ψ and id_Γ for the identity global and local weakenings, respectively. We often omit the subscript when it can be inferred from the context. Identity and composition of weakenings $\Psi;\Gamma \Longrightarrow \Phi;\Delta$ are defined pairwise. We verify that dual contexts and weakenings form a category, which is referred to as $\mathcal{W}$. This is the base category that we will be working with. We sometimes also need to work with $\mathcal{GW}$, the category of global contexts and global weakenings.

4.2 Presheaf Model and Interpretations

In this section, we define the normalization algorithm with $\mathcal{W}$ as our base category. The algorithm normalizes terms to their $\beta\eta$-normal forms, which are defined as follows:

$$\begin{aligned} w &:= v \mid \mathsf{zero} \mid \mathsf{succ}\ w \mid \mathtt{box}\ t \mid \lambda x.w && \text{(Normal form (}\mathsf{Nf}\text{))} \\ v &:= x \mid u^\theta \mid v\ w \mid \mathsf{rec}_T\ w\ (x\ y.w')\ v \mid \mathtt{letbox}\ u \leftarrow v\ \mathtt{in}\ w && \text{(Neutral form (}\mathsf{Ne}\text{))} \\ \theta &:= \cdot \mid \theta, w/x && \text{(Normal local substitutions)} \end{aligned}$$

Notice that `box` t is already normal for any t. This is expected because `box` t regards t as static code so t cannot be reduced. These definitions induce the sets of well-typed normal and neutral forms:

$$\mathsf{Nf}^{T}_{\Psi;\Gamma} := \{w \mid \Psi;\Gamma \vdash_1 w : T\} \qquad \mathsf{Ne}^{T}_{\Psi;\Gamma} := \{v \mid \Psi;\Gamma \vdash_1 v : T\}$$

The sets only capture terms at layer 1 due to the lack of reductions at layer 0. Nf^T and Ne^T then are induced presheaves mapping dual contexts to the sets of normal and neutral forms, respectively.

Next we give the interpretation of types. The interpretation of function types is presheaf exponentials derived from the Yoneda lemma with naturality:

$$\begin{gathered} F \,\hat{\longrightarrow}\, G : \mathcal{W}^{op} \Longrightarrow \mathcal{S}et \\ (F \,\hat{\longrightarrow}\, G)_{\Psi;\Gamma} := \forall\ \gamma;\tau : \Phi;\Delta \Longrightarrow \Psi;\Gamma\ .\ F_{\Phi;\Delta} \to G_{\Phi;\Delta} \end{gathered}$$

We define the interpretations of types, and global and local contexts in Fig. 4. Both `Nat` and $\square(\Gamma \vdash T)$ are interpreted as their presheaves of normal forms. In particular, $[\![\square(\Gamma \vdash T)]\!]$ is not even recursive. This case effectively interprets $\square(\Gamma \vdash T)$ as the code of T open in Γ. Based on the definition, two possible kinds of terms in $\mathsf{Nf}^{\square(\Gamma \vdash T)}$ are either neutral or of the form `box` t. In the latter case, we have gained access to the syntax of t, permitting more complex operations like pattern matching on code.

The interpretation of global contexts is layered. At layer 1, it is the presheaf of global substitutions, containing code at layer 0 awaiting to be evaluated. At layer 0, it is the Hom set of $\mathcal{GW}$, i.e. the presheaf of global weakenings. This definition is motivated technically to ensure the naturality of evaluation of terms to be defined shortly. We let σ to range over $[\![\Phi]\!]^i_\Psi$ when i is unknown. Local contexts are interpreted as iterated products of values as usual, where $*$ is the unique element of a chosen singleton set. A dual context is interpreted pairwise:

$$[\![\Phi;\Delta]\!]^i_{\Psi;\Gamma} := [\![\Phi]\!]^i_\Psi \times [\![\Delta]\!]_{\Psi;\Gamma}$$

All interpretations above are functors:

Lemma 5. *$[\![T]\!]$, $[\![\Phi]\!]^i$, $[\![\Delta]\!]$ and $[\![\Phi;\Delta]\!]^i$ are presheaves. $[\![\Phi]\!]^i$ is from $\mathcal{GW}$.*

If $a \in [\![T]\!]_{\Phi;\Delta}$ and $\gamma;\tau : \Psi;\Gamma \Longrightarrow \Phi;\Delta$, we write $a[\gamma;\tau]$ for the functorial action of $\gamma;\tau$ on a. We generalize this notation to other functors.

Finally, we define the evaluation functions, interpreting terms as natural transformations between presheaves. This interpretation relies on two other natural transformations, reification and reflection, which map Ne^T to $[\![T]\!]$ and $[\![T]\!]$ to Nf^T, respectively. All four natural transformations are defined in Fig. 5. Since `Nat` and $\square(\Gamma \vdash T)$ are interpreted as presheaves of normal forms, their cases in reification and reflection are just identities. The case for functions is defined in the same way as in STLC.

Our evaluation is a moderate extension of the evaluation of STLC [5]. The evaluation function is layered because the type theory itself is layered. The cases overlapping with STLC are identical, so we only discuss the modal cases. The

$$
\begin{aligned}
&\downarrow^{T}_{\Psi;\Gamma} : [\![T]\!]_{\Psi;\Gamma} \to \mathsf{Nf}^{T}_{\Psi;\Gamma} && \textbf{(Reification)}\\
&\downarrow^{\mathtt{Nat}}_{\Psi;\Gamma}(a) := a\\
&\downarrow^{\square T}_{\Psi;\Gamma}(a) := a\\
&\downarrow^{S\longrightarrow T}_{\Psi;\Gamma}(a) := \lambda x.\downarrow^{T}_{\Psi;\Gamma,x:S}(a\ (\mathsf{id};p(\mathsf{id})\ ,\ \uparrow^{S}_{\Psi;\Gamma,x:S}(x)))\\
&&& \text{(where } \mathsf{id};p(\mathsf{id}) : \Psi;\Gamma,x{:}S \Longrightarrow \Psi;\Gamma)
\end{aligned}
$$

$$
\begin{aligned}
&\uparrow^{T}_{\Psi;\Gamma} : \mathsf{Ne}^{T}_{\Psi;\Gamma} \Longrightarrow [\![T]\!]_{\Psi;\Gamma} && \textbf{(Reflection)}\\
&\uparrow^{B}_{\Psi;\Gamma}(v) := v\\
&\uparrow^{\square T}_{\Psi;\Gamma}(v) := v\\
&\uparrow^{S\longrightarrow T}_{\Psi;\Gamma}(v) :=\\
&\quad(\gamma;\tau : \Phi;\Delta \Longrightarrow \Psi;\Gamma)(a \in [\![S]\!]_{\Phi;\Delta}) \mapsto \uparrow^{T}_{\Phi;\Delta}(v[\gamma;\tau]\ \downarrow^{S}_{\Phi;\Delta}(a))
\end{aligned}
$$

$$
\begin{aligned}
&[\![_]\!]^{i}_{\Psi;\Gamma} : \Phi;\Delta \vdash_i t : T \to [\![\Phi;\Delta]\!]^{i}_{\Psi;\Gamma} \to [\![T]\!]_{\Psi;\Gamma} && \textbf{(Evaluation)}\\
&[\![\mathsf{zero}]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) := \mathsf{zero}\\
&[\![\mathsf{succ}\ t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) := \mathsf{succ}\ ([\![t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho))\\
&[\![u^{\delta}]\!]^{0}_{\Psi;\Gamma}(\gamma;\rho) := \uparrow^{T}_{\Psi;\Gamma}(u[\gamma]^{\theta})\\
&\quad\text{(where } u : (\Delta' \vdash T) \in \Phi,\ \Phi;\Delta \vdash_0 \delta : \Delta',\text{ and } \theta := \downarrow^{\Delta'}_{\Psi;\Gamma}([\![\delta]\!]^{0}_{\Psi;\Gamma}(\gamma;\rho)))\\
&[\![u^{\delta}]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) := [\![\sigma(u)]\!]^{0}_{\Psi;\Gamma}(\mathsf{id};[\![\delta]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho))\\
&[\![x]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) := \rho(x) && \text{(lookup } x \text{ in } \rho)\\
&[\![\lambda x : S.t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) :=\\
&\quad(\gamma;\tau : \Phi';\Delta' \Longrightarrow \Psi;\Gamma)(a \in [\![S]\!]_{\Phi';\Delta'}) \mapsto [\![t]\!]^{i}_{\Phi';\Delta'}(\sigma';(\rho',a))\\
&&& \text{(where } (\sigma';\rho') := \sigma;\rho[\gamma;\tau] \in [\![\Phi;\Delta]\!]^{i}_{\Phi';\Delta'})\\
&[\![t\ s]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) := [\![t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho)(\mathsf{id}_{\Psi;\Gamma}, [\![s]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho))\\
&[\![\mathtt{box}\ t]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) := \mathtt{box}\ (t[\sigma])\\
&[\![\mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) := [\![t]\!]^{1}_{\Psi;\Gamma}(\sigma, s'/u;\rho) && \text{(if } [\![s]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) = \mathtt{box}\ s')\\
&[\![\mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) :=\\
&\quad\uparrow^{T}_{\Psi;\Gamma}(\mathtt{letbox}\ u \leftarrow v\ \mathtt{in}\ \downarrow^{T}_{\Psi,u:S;\Gamma}([\![t]\!]^{1}_{\Psi,u:S;\Gamma}(\sigma', u^{\mathsf{id}}/u;\rho')))\\
&\quad\text{(if } [\![s]\!]^{1}_{\Psi;\Gamma}(\sigma;\rho) = v,\text{ also } (\sigma';\rho') := (\sigma;\rho)[p(\mathsf{id});\mathsf{id}] \in [\![\Phi;\Delta]\!]^{1}_{\Psi,u:S;\Gamma})
\end{aligned}
$$

$$
\begin{aligned}
&[\![_]\!]^{i}_{\Psi;\Gamma} : \Phi;\Delta \vdash_i \delta : \Delta' \to [\![\Phi;\Delta]\!]^{i}_{\Psi;\Gamma} \to [\![\Delta']\!]_{\Psi;\Gamma}\\
&&& \textbf{(Substitution Evaluation)}\\
&[\![\cdot]\!]^{i}_{\Psi;\Gamma}(_) := *\\
&[\![\delta, t/x]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho) := ([\![\delta]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho), [\![t]\!]^{i}_{\Psi;\Gamma}(\sigma;\rho))
\end{aligned}
$$

Fig. 5: Definitions of reification, reflection and evaluation

`box` t case is only available at layer 1. In this case, we directly propagate σ under `box`. In the `letbox` case, we first evaluate s. Given $[\![s]\!]^{1}_{\Psi;\Gamma} \in [\![\square(\Gamma \vdash S)]\!]^{1}_{\Psi;\Gamma} = \mathsf{Nf}^{\square(\Gamma\vdash S)}_{\Psi;\Gamma}$, this evaluation has two possible results: it returns either a `box` s', or a neutral v. In the first case, we just recurse with σ extended with s' for u. In the second case of `letbox`, some neutral v blocks the evaluation, so we can only recurse on the body t with u as is and with σ and ρ properly weakened. To obtain a $[\![T]\!]_{\Psi;\Gamma}$, we reify the evaluation of t and obtain a normal form, using which we obtain a neutral of `letbox`. A reflection of this neutral gives us a $[\![T]\!]_{\Psi;\Gamma}$.

The interpretation of global variables is the most interesting. When u^{δ} is referred to at layer 1, we are evaluating some code and turning it into a program, i.e. *running* it. We retrieve the code by looking up u in σ, and continue the evaluation at layer 0 with an environment obtained by evaluating δ. Notice that

the layer decreases so the interpretation is well-founded regardless of the size of $\sigma(u)$. The evaluation of a local substitution recursively evaluates all terms in the local substitution. If we refer to u^δ at layer 0, then u should stay neutral. Moreover, the evaluation function is required to return a natural transformation. Both requirements lead to the interpretation of global contexts as Hom set of $\mathcal{GW}$ at layer 0, since a weakened global variable is still a global variable and neutral. We first normalize δ by evaluating and then reifying it $(\downarrow^{\Delta'}_{\Psi;\Gamma})$ and obtain a $[\![T]\!]_{\Psi;\Gamma}$ by reflection. Last, reification, reflection and evaluation are all natural transformations:

Lemma 6 (Naturality). *If $\gamma;\tau : \Psi';\Gamma' \Longrightarrow \Psi;\Gamma$,*

- *if $a \in [\![T]\!]_{\Psi;\Gamma}$, then $\downarrow^T_{\Psi;\Gamma}(a)[\gamma;\tau] = \downarrow^T_{\Psi';\Gamma'}(a[\gamma;\tau])$;*
- *if $v \in \mathsf{Ne}^T_{\Psi;\Gamma}$, then $\uparrow^T_{\Psi;\Gamma}(v)[\gamma;\tau] = \uparrow^T_{\Psi';\Gamma'}(v[\gamma;\tau])$;*
- *if $\Phi;\Delta \vdash_i t : T$ and $\sigma;\rho \in [\![\Phi;\Delta]\!]^i_{\Psi;\Gamma}$, then $[\![t]\!]^i_{\Psi;\Gamma}(\sigma;\rho)[\gamma;\tau] = [\![t]\!]^i_{\Psi';\Gamma'}((\sigma;\rho)[\gamma;\tau])$.*

The NbE algorithm is given by composing the interpretations:

Definition 1. *A normalization by evaluation algorithm given $\Psi;\Gamma \vdash_1 t : T$ is*

$$\begin{aligned} &\mathsf{nbe}^T_{\Psi;\Gamma}(t) : \mathsf{Nf}^T_{\Psi;\Gamma} \\ &\mathsf{nbe}^T_{\Psi;\Gamma}(t) := \downarrow^T_{\Psi;\Gamma}([\![t]\!]^1_{\Psi;\Gamma}(\uparrow^{\Psi;\Gamma})) \end{aligned}$$

where $\uparrow^{\Psi;\Gamma} \in [\![\Psi;\Gamma]\!]^1_{\Psi;\Gamma}$ is a tuple of the identity global substitution and the identity environment.

This algorithm is correct due to the following two theorems:

Theorem 1 (Completeness). *If $\Psi;\Gamma \vdash_1 t \approx t' : T$, then $\mathsf{nbe}^T_{\Psi;\Gamma}(t) = \mathsf{nbe}^T_{\Psi;\Gamma}(t')$.*

Theorem 2 (Soundness). *If $\Psi;\Gamma \vdash_1 t : T$, then $\Psi;\Gamma \vdash_1 t \approx \mathsf{nbe}^T_{\Psi;\Gamma}(t) : T$.*

The completeness theorem states that equivalent terms have equal normal forms, so we can compare the syntactic equality between normal forms to decide whether two terms are equivalent. The soundness theorem states that a well-typed term has and is equivalent to its normal form. Notice that the theorems are about layer 1 because only terms at this layer compute. In the remainder of this section, we outline only the soundness proof. For complete details, please refer to our technical report [27].

4.3 Soundness

The soundness theorem is established via gluing models, which relate syntactic terms with semantic values. In our 2-layered system, we need two layers of gluing models, which reflect the fact that we are actually operating in two languages. For a gluing relation R, we write $a \sim b \in R$ to denote $(a, b) \in R$.

Layer-0 Gluing Model We begin with the gluing model for natural numbers. It recursively relates a term t and a normal form of type `Nat`. This gluing relation applies for both layers:

$$\frac{\Psi; \Gamma \vdash_1 t \approx \mathsf{zero} : \mathtt{Nat}}{t \sim \mathsf{zero} \in \mathtt{Nat}_{\ \Psi;\Gamma}} \qquad \frac{\Psi; \Gamma \vdash_1 t \approx \mathsf{succ}\ t' : \mathtt{Nat} \quad t' \sim w \in \mathtt{Nat}_{\ \Psi;\Gamma}}{t \sim \mathsf{succ}\ w \in \mathtt{Nat}_{\ \Psi;\Gamma}} \qquad \frac{\Psi; \Gamma \vdash_1 t \approx v : \mathtt{Nat}}{t \sim v \in \mathtt{Nat}_{\ \Psi;\Gamma}}$$

At layer 0, for all T $\mathtt{wf}^0$, its gluing model is:

$$\begin{aligned} (\!|T|\!)^0_{\Psi;\Gamma} &\subseteq \mathsf{Exp} \times [\![T]\!]_{\ \Psi;\Gamma} \\ (\!|\mathtt{Nat}|\!)^0_{\Psi;\Gamma} &:= \mathtt{Nat}_{\ \Psi;\Gamma} \\ (\!|S \longrightarrow T|\!)^0_{\Psi;\Gamma} &:= \{(t, a) \mid \forall\ \gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma, s \sim b \in (\!|S|\!)^0_{\Phi;\Delta}\ . \\ &\qquad\qquad t[\gamma; \tau]\ s \sim a(\gamma; \tau, b) \in (\!|T|\!)^0_{\Phi;\Delta}\} \end{aligned}$$

$(\!|T|\!)^0$ does not have a case for $\square$ due to T $\mathtt{wf}^0$. The function case requires that the results of function applications remain related for all weakenings and all related arguments. The gluing between local substitutions and evaluation environments $\delta \sim \rho \in (\!|\Delta|\!)^0_{\Psi;\Gamma}$ is defined by using $(\!|T|\!)^0$ to relate terms and values pairwise.

Definition 2. *We define the semantic judgment at layer* 0*:*

$$\Psi; \Gamma \Vdash_0 t : T := \forall\ \gamma : \Phi \Longrightarrow_g \Psi \ \text{and}\ \delta \sim \rho \in (\!|\Gamma|\!)^0_{\Phi;\Delta}\ . t[\gamma][\delta] \sim [\![t]\!]^0_{\Phi;\Delta}(\gamma; \rho) \in (\!|T|\!)^0_{\Phi;\Delta}$$

The semantic judgment at layer 0 only universally quantifies over global weakenings.

Layer-1 Gluing Model The reason why we must define the layer-0 gluing model first is that we refer to $\Psi; \Gamma \Vdash_0 t : T$ in our layer-1 model. The semantics of $\square(\Delta \vdash T)$ is given in terms of the semantic judgment at layer 0:

$$\frac{\Psi; \Gamma \vdash_1 t \approx \mathtt{box}\ s : \square(\Delta \vdash T) \qquad \Psi; \Delta \Vdash_0 s : T}{t \sim \mathtt{box}\ s \in \square(\Delta \vdash T)_{\ \Psi;\Gamma}} \qquad \frac{\Psi; \Gamma \vdash_1 t \approx v : \square(\Delta \vdash T)}{t \sim v \in \square(\Delta \vdash T)_{\ \Psi;\Gamma}}$$

In the first rule, t is related to `box` s and s is a semantically well-typed term at layer 0. The premise $\Psi; \Delta \Vdash_0 s : T$ is necessary when we prove the semantic typing rule for `letbox`. Without it, we will not able to maintain the semantic well-formedness of global substitutions during evaluation and in the semantic judgment at layer 1. The details are in our technical report. The gluing model at layer 1 for T $\mathtt{wf}^1$ is now defined as:

$$\begin{aligned} (\!|T|\!)^1_{\Psi;\Gamma} &\subseteq \mathsf{Exp} \times [\![T]\!]_{\ \Psi;\Gamma} \\ (\!|\mathtt{Nat}|\!)^1_{\Psi;\Gamma} &:= \mathtt{Nat}_{\ \Psi;\Gamma} \\ (\!|\square(\Delta \vdash T)|\!)^1_{\Psi;\Gamma} &:= \square(\Delta \vdash T)_{\ \Psi;\Gamma} \\ (\!|S \longrightarrow T|\!)^1_{\Psi;\Gamma} &:= \{(t, a) \mid \forall\ \gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma, s \sim b \in (\!|S|\!)^1_{\Phi;\Delta}\ . \\ &\qquad\qquad t[\gamma; \tau]\ s \sim a(\gamma; \tau, b) \in (\!|T|\!)^1_{\Phi;\Delta}\} \end{aligned}$$

Compared to the layer-0 model, the layer-1 model only has an extra case $(\!|\square(\Delta \vdash T)|\!)^1$. The other two cases are just the same:

Lemma 7. *If T $\texttt{wf}^0$, then $(\!|T|\!)^0_{\Psi;\Gamma} = (\!|T|\!)^1_{\Psi;\Gamma}$.*

This lemma semantically describes the matryoshka principle, witnessing the subsumption of layer 0 into layer 1. The semantic judgment at layer 1 is universally quantified over a semantic global substitution defined below:

$$\frac{\Psi\ \texttt{wf}^0}{\Psi \Vdash \cdot : \cdot} \qquad \frac{\Psi \Vdash \sigma : \Phi \qquad \Psi; \Gamma \Vdash_0 t : T}{\Psi \Vdash \sigma, t/u : \Phi, u : (\Gamma \vdash T)}$$

We define the semantic judgment at layer 1:

$$\Psi; \Gamma \Vdash_1 t : T := \forall\ \Phi \Vdash \sigma : \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^1_{\Phi;\Delta}\ .\ t[\sigma][\delta] \sim [\![t]\!]^1_{\Phi;\Delta}(\sigma;\rho) \in (\!|T|\!)^1_{\Phi;\Delta}$$

The fundamental theorem is established by proving all semantic typing rules:

Theorem 3 (Fundamental). *If $\Psi; \Gamma \vdash_i t : T$, then $\Psi; \Gamma \Vdash_i t : T$.*
If $\Psi; \Gamma \vdash_i \delta : \Delta$, then $\Psi; \Gamma \Vdash_i \delta : \Delta$.

5 Supporting Pattern Matching on Code

In the previous section, we have achieved feature ① and partly feature ②. In this section, we extend the previous system with pattern matching on code, so all features are concluded. We adapt the presheaf model and show that the normalization algorithm remains complete and sound. We introduce a creative semantics in the soundness proof in order to justify pattern matching on code.

5.1 Extension of Pattern Matching

In this section, we extend our previous 2-layered modal type theory with pattern matching on code as follows.

$$\begin{array}{lll} s, t := & \cdots \mid \mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} & \text{(Terms, } \mathsf{Exp}\text{)} \\ b \quad := & \mathsf{var}_x \Rightarrow t \mid \mathsf{zero} \Rightarrow t \mid \mathsf{succ}\ ?u \Rightarrow t \mid \lambda x.?u \Rightarrow t \mid ?u\ ?u' \Rightarrow t & \\ & \mid \mathsf{rec}_T\ ?u\ (x\ y.?u')\ ?u'' \Rightarrow t & \text{(Branches)} \end{array}$$

We extend the system with another elimination form of $\square(\Gamma \vdash T)$, pattern matching ($\mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b}$), where $\overrightarrow{b}$ is a list of *all possible branches* of t. The branches only need to match terms in STLC because pattern matching is only available at layer 1 and the scrutinee is code from layer 0. We do not directly support nested patterns like $(\lambda y.?u)\ ?u'$ to keep the system simple, but they can be encoded as nested pattern matchings. Supporting any useful general recursor (e.g. [47,29]) would require context variables, which abstract over local contexts, and type polymorphism. We see these extensions orthogonal to layering and leave it to future work.

Further modifications to the typing and equivalence rules are in Fig. 6. We omit the case for rec for conciseness. The typing rule for match uses the judgment $\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'$. This judgment enumerates all possible branches based

$\boxed{\Psi; \Gamma \vdash_i t : T}$ Term t has type T in contexts Ψ and Γ at layer i where $i \in [0, 1]$

$$\frac{\Psi; \Gamma \vdash_1 s : \Box(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \mathsf{match}\ s\ \mathsf{with}\ \overrightarrow{b} : T'}$$

$\boxed{\Psi; \Gamma \vdash_1 b : \Delta \vdash T \Rightarrow T'}$ b is a branch of type T' w.r.t. a code of type T open in Δ.

$$\frac{\Delta\ \mathtt{wf}^0 \qquad x : T \in \Delta \qquad \Psi; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{var}_x \Rightarrow t : \Delta \vdash T \Rightarrow T'} \qquad \frac{\Delta\ \mathtt{wf}^0 \qquad \Psi; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{zero} \Rightarrow t : \Delta \vdash \mathtt{Nat} \Rightarrow T'}$$

$$\frac{\Psi, u : (\Delta \vdash \mathtt{Nat}); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{succ}\ ?u \Rightarrow t : \Delta \vdash \mathtt{Nat} \Rightarrow T'} \qquad \frac{\Psi, u : (\Delta, x : S \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \lambda x.?u \Rightarrow t : \Delta \vdash S \longrightarrow T \Rightarrow T'}$$

$$\frac{\forall\ S\ \mathtt{wf}^0 . \Psi, u : (\Delta \vdash S \longrightarrow T), u' : (\Delta \vdash S); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 ?u\ ?u' \Rightarrow t : \Delta \vdash T \Rightarrow T'}$$

$\boxed{\Psi; \Gamma \vdash_i t \approx t' : T}$ Terms t and t' are equivalent (β rules for match)

$$\frac{x : T \in \Delta \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T' \qquad \overrightarrow{b}(x) = \mathsf{var}_x \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ x\ \mathsf{with}\ \overrightarrow{b} \approx t : T'}$$

$$\frac{\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash \mathtt{Nat} \Rightarrow T' \qquad \overrightarrow{b}(\mathsf{zero}) = \mathsf{zero} \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ \mathsf{zero}\ \mathsf{with}\ \overrightarrow{b} \approx t : T'}$$

$$\frac{\Psi; \Delta \vdash_0 s : \mathtt{Nat} \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash \mathtt{Nat} \Rightarrow T' \qquad \overrightarrow{b}(\mathsf{succ}\ s) = \mathsf{succ}\ ?u \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ (\mathsf{succ}\ s)\ \mathsf{with}\ \overrightarrow{b} \approx t[s/u] : T'}$$

$$\frac{\Psi; \Delta, x : S \vdash_0 s : T \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash S \longrightarrow T \Rightarrow T' \qquad \overrightarrow{b}(\lambda x.s) = \lambda x.?u \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ (\lambda x.s)\ \mathsf{with}\ \overrightarrow{b} \approx t[s/u] : T'}$$

$$\frac{\Psi; \Delta \vdash_0 t : S \longrightarrow T \quad \Psi; \Delta \vdash_0 s : S \quad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T' \quad \overrightarrow{b}(t\ s) = ?u\ ?u' \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathsf{box}\ (t\ s)\ \mathsf{with}\ \overrightarrow{b} \approx t[t/u, s/u'] : T'}$$

Fig. 6: Adjusted rules with contextual types

on the type of the scrutinee. This guarantees coverage of pattern matching, i.e. that $\overrightarrow{b}$ is indeed a list of *all possible branches* for a given scrutinee of type $\Box(\Delta \vdash T)$.

All typing rules for individual branches are similar. For example, if the pattern is $\lambda x.?u$, then u captures the body of some λ. The branch body t is checked with u bound to $(\Delta, x : S \vdash T)$, which has a larger local context than Δ which we begin with. If the branch matches a function application, our premise requires t is well-typed for all $S\ \mathtt{wf}^0$. This universal quantification should be read as a higher-order derivation that applies for all $S\ \mathtt{wf}^0$ (see also [60]) and where we keep S abstract as a parameter.

The bottom of Fig. 6 are the β rules for pattern matching. Based on the structure of the scrutinee, we dispatch to the right branch and propagate instantiations for pattern variables via global substitutions to the bodies. Notations like $\overrightarrow{b}(\mathsf{succ}\ s)$ denote the lookup of $\overrightarrow{b}$ based on a given shape. For example, $\overrightarrow{b}(\mathsf{succ}\ s) = \mathsf{succ}\ ?u \Rightarrow t$ means that we look up $\mathsf{succ}\ s$ in $\overrightarrow{b}$, and find the

branch $\mathsf{succ}\ ?u \Rightarrow t$. Then s is meant to substitute u in t. This lookup is guaranteed to succeed because $\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'$ is covering.

5.2 Neutral Forms

Careful readers might have noticed that in our grammar of branches, we do not have a case for global variables, nor do we have a β rule for pattern matching on $\mathtt{box}\ u^\delta$. So what should $\mathsf{match}\ \mathtt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{b}$ be reduced to? The answer might be surprising: this term in fact is *blocked*. Previously, we mentioned a concern about `isapp` raised by Kavvos [33]. His subsequent analysis concludes that sound intensional operations can only act on globally and locally closed code. This restriction is clearly too strong. After looking into the analysis, we see that this conclusion is based on the assumption that intensional operations reduce on $\mathtt{box}\ u^\delta$, which leads to the strong restriction. $\mathsf{match}\ \mathtt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{b}$ should not reduce, just for the same reason $\mathsf{match}\ x\ \mathsf{with}\ \overrightarrow{b}$ should not. They are both waiting for substitutions to supply an actual code to unblock the evaluation. Their only difference is that they act on different substitutions. This observation leads to a renewed definition of neutral forms:

$$
\begin{aligned}
v &:= \cdots \mid \mathsf{match}\ v\ \mathsf{with}\ \overrightarrow{r} \mid \mathsf{match}\ \mathtt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{r} && \text{(Neutral form } (\mathsf{Ne})) \\
r &:= \mathsf{var}_x \Rightarrow w \mid \mathsf{zero} \Rightarrow w \mid \mathsf{succ}\ ?u \Rightarrow w \mid \lambda x.?u \Rightarrow w \mid ?u\ ?u' \Rightarrow w \\
&\quad \mid \mathsf{rec}_T\ ?u\ (x\ y.?u')\ ?u'' \Rightarrow w && \text{(Normal branches)}
\end{aligned}
$$

The definition of normal forms, described by w, remains the same. To obtain $\beta\eta$ normal forms, all branches should be normalized. If u is a scrutinee of a match, its local substitution stays as is because it is considered as code. This adjustment is subtle but critical to give a sound presheaf model.

Moreover, notice that $\mathtt{letbox}\ u \leftarrow \mathtt{box}\ u'^\delta\ \mathtt{in}\ t$ does not get blocked. This difference in computational behaviors is due to different purposes of two elimination forms. `letbox` is primary for code composition and the execution of code, while pattern matching focuses on intensional analysis of code. For this reason, we include both `letbox` and pattern matching as elimination forms. They coexist perfectly at layer 1 because our core theory at layer 0 is unaffected. Without layering, both `letbox` and pattern matching are available everywhere, including inside of `box`, which causes all sorts of complex interactions and makes the computational behavior of the whole type theory difficult to control. This is why former systems based on [17] are so difficult to extend with intensional analysis in a controlled manner. Now we have introduced pattern matching on code and achieved feature ③ outlined in Sec. 1. In the rest of this section, we fix the normalization proof and justify that this system is a proper type theory.

5.3 Adjusting the Presheaf Model

Since we only add an elimination form, we simply extend the model in Sec. 4. The adjustments are shown in Fig. 7. Two additional functions are defined: first, the match function dispatches evaluation to the proper branch based on the

input code and evaluates the body with a global substitution and an evaluation environment; second, nfbranch normalizes the body of a branch and obtains a normal branch. Applying nfbranch to $\overrightarrow{b}$ normalizes all branches in $\overrightarrow{b}$. nfbranch is invoked when we normalize a pattern matching on some neutral code.

Let us consider the match case. We first evaluate the scrutinee. If the result is a neutral term, then we simply invoke nfbranch to normalize all branch bodies, and then use reflection to obtain a $[\![T]\!]_{\Psi;\Gamma}$. Each case in nfbranch proceeds similarly. The evaluation of the body continues with the global substitution extended with pattern variables. The evaluated body is then reified to a normal form and thus the resulting branch is also normal. If the result is `box` s, then we match the code s accordingly with a branch and evaluate the body. This is done by calling the match function. Based on the shape of s, the match function looks up $\overrightarrow{b}$ and extends σ accordingly before evaluating the body. For example, if s is a λ, then the branch $\lambda x.?u \Rightarrow t$ is picked, and t is evaluated. The lookup of $\overrightarrow{b}$ must succeed because our pattern matching is covering. However, if s is just a global variable, based on the previous discussion on neutral forms, we must block the evaluation and only normalize the branches. The case forms a neutral form and is essentially the same as if the scrutinee is evaluated to a neutral.

The presheaf interpretation gives a semantic explanation of how layering enables sound pattern matching on code and why it is difficult in purely homogeneous systems. A term of type $\square(\Gamma \vdash T)$ has two different uses: it either stands for code that will be run (due to `letbox`) or it stands for code that will be analyzed (due to pattern matching). In the former case, it is evaluated to a natural transformation in the semantics while in the latter, only its syntactic information is needed. Moreover, these two uses are not mutually exclusive. A program might use both semantic and syntactic information of the same code. To support pattern matching on code, we must maintain both semantic and syntactic information and therefore the evaluation of code must be postponed. In our setting, this evaluation only happens when we evaluate a global variable at layer 1. The evaluation function evaluates the code represented by the global variable and maintains its well-foundedness by decreasing the layer from 1 to 0. Meanwhile, in a homogeneous system without layers, it is no longer clear how to give a well-founded evaluation of a global variable due to the lack of proper measure if intensional analysis is supported.

As we will see very shortly, the intuition above based on two different uses of code must be formalized in the gluing model in order to establish a soundness proof, giving a formal account for the importance of layering.

5.4 Soundness

Recall that in Sec. 4.3, we need a 2-layered model, where the layer-1 model refers to the semantic judgment at layer 0 to support `letbox`. The semantic judgment at layer 0 is defined as a universal quantification over global weakenings and the gluing between local substitutions and evaluation environments:

$$\Psi; \Gamma \Vdash_0 t : T := \forall\ \gamma : \Phi \Longrightarrow_g \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^0_{\Phi;\Delta}\ .\ t[\gamma][\delta] \sim [\![t]\!]^0_{\Phi;\Delta}(\gamma;\rho) \in (\!|T|\!)^0_{\Phi;\Delta}$$

$$\llbracket _ \rrbracket^i_{\Psi;\Gamma} : \Phi; \Delta \vdash_i t : T \to \llbracket \Phi; \Delta \rrbracket^i_{\Psi;\Gamma} \to \llbracket T \rrbracket_{\Psi;\Gamma} \quad \textbf{(Evaluation)}$$
$$\llbracket \mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) := \mathsf{match}(s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) \qquad (\text{if } \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) = \mathtt{box}\ s)$$
$$\llbracket \mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) := \uparrow^T_{\Psi;\Gamma} (\mathsf{match}\ v\ \mathsf{with}\ \overrightarrow{r})$$
$$(\text{if } \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) = v : \Box(\Delta' \vdash S) \text{ for some } \Delta' \text{ and } \overrightarrow{r} := \mathsf{nfbranch}(\overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho))$$

$$\mathsf{match} : \Psi; \Gamma \vdash_0 t : T \to \Phi; \Delta \vdash_1 \overrightarrow{b} : \Delta' \vdash T \Rightarrow T' \to \llbracket \Phi; \Delta \rrbracket^1_{\Psi;\Gamma} \to \llbracket T \rrbracket_{\Psi;\Gamma}$$
$$\textbf{(Branch Evaluation Based on Code)}$$
$$\mathsf{match}(x, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) \qquad (\text{where } x \Rightarrow t := \overrightarrow{b}(x))$$
$$\mathsf{match}(\mathsf{zero}, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho) \qquad (\text{where } \mathsf{zero} \Rightarrow t := \overrightarrow{b}(\mathsf{zero}))$$
$$\mathsf{match}(\mathsf{succ}\ s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma, s/u;\rho) \qquad (\text{where } \mathsf{succ}\ ?u \Rightarrow t := \overrightarrow{b}(\mathsf{succ}\ s))$$
$$\mathsf{match}(\lambda x.s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma, s/u;\rho) \qquad (\text{where } \lambda x.?u \Rightarrow t := \overrightarrow{b}(\lambda x.s))$$
$$\mathsf{match}(t'\ s, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma, t'/u, s/u;\rho) \qquad (\text{where } ?u\ ?u' \Rightarrow t := \overrightarrow{b}(t'\ s))$$
$$\mathsf{match}(u^\delta, \overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho) := \uparrow^{T'}_{\Psi;\Gamma} (\mathsf{match}\ \mathtt{box}\ u^\delta\ \mathsf{with}\ \overrightarrow{r})$$
$$(\text{where } \overrightarrow{r} := \mathsf{nfbranch}(\overrightarrow{b})_{\Psi;\Gamma}(\sigma;\rho))$$

$$\mathsf{nfbranch} : \Phi; \Delta \vdash_1 b : \Delta' \vdash T \Rightarrow T' \to \llbracket \Phi; \Delta \rrbracket^1_{\Psi;\Gamma} \to \Psi; \Gamma \vdash_1 r : \Delta' \vdash T \Rightarrow T'$$
$$\textbf{(Normalization of A Branch)}$$
$$\mathsf{nfbranch}(x \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) := x \Rightarrow \downarrow^{T'}_{\Psi;\Gamma} (\llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho))$$
$$\mathsf{nfbranch}(\mathsf{zero} \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) := \mathsf{zero} \Rightarrow \downarrow^{T'}_{\Psi;\Gamma} (\llbracket t \rrbracket^1_{\Psi;\Gamma}(\sigma;\rho))$$
$$\mathsf{nfbranch}(\mathsf{succ}\ ?u \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) :=$$
$$\mathsf{succ}\ ?u \Rightarrow \downarrow^{T'}_{\Psi, u:(\Delta' \vdash \mathtt{Nat});\Gamma} (\llbracket t \rrbracket^1_{\Psi, u:(\Delta' \vdash \mathtt{Nat});\Gamma}(\sigma', u^{\mathsf{id}}/u;\rho'))$$
$$(\text{where } (\sigma';\rho') := (\sigma;\rho)[p(\mathsf{id});\mathsf{id}] \in \llbracket \Phi; \Delta \rrbracket^1_{\Psi, u:(\Delta' \vdash \mathtt{Nat});\Gamma})$$
$$\mathsf{nfbranch}(\lambda x.?u \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) :=$$
$$\lambda x.?u \Rightarrow \downarrow^{T'}_{\Psi, u:(\Delta', x:S \vdash T);\Gamma} (\llbracket t \rrbracket^1_{\Psi, u:(\Delta', x:S \vdash T);\Gamma}(\sigma', u^{\mathsf{id}}/u;\rho'))$$
$$(\text{where } x : S \text{ and } (\sigma';\rho') := (\sigma;\rho)[p(\mathsf{id});\mathsf{id}] \in \llbracket \Phi; \Delta \rrbracket^1_{\Psi, u:(\Delta', x:S \vdash T);\Gamma})$$
$$\mathsf{nfbranch}(?u\ ?u' \Rightarrow t)_{\Psi;\Gamma}(\sigma;\rho) := ?u\ ?u' \Rightarrow \downarrow^{T'}_{\Psi';\Gamma} (\llbracket t \rrbracket^1_{\Psi';\Gamma}(\sigma', u^{\mathsf{id}}/u, u'^{\mathsf{id}}/u';\rho'))$$
$$(\text{for any } S\ \mathtt{wf}^0, \text{ where } \Psi' := \Psi, u : (\Delta' \vdash S \longrightarrow T'), u' : (\Delta' \vdash S),)$$
$$(\text{and } (\sigma';\rho') := (\sigma;\rho)[p(p(\mathsf{id}));\mathsf{id}] \in \llbracket \Phi; \Delta \rrbracket^1_{\Psi';\Gamma})$$

Fig. 7: Adjustments to the presheaf model

This definition taken from Sec. 4.3, unfortunately, cannot support the semantic rule for pattern matching. Consider some $\Psi; \Gamma \Vdash_0 t\ s : T$. To prove that pattern matching on code is semantically sound, we perform an analysis on the structure of $t\ s$. But we are stuck, because we cannot derive $\Psi; \Gamma \Vdash_0 t : S \longrightarrow T$ or $\Psi; \Gamma \Vdash_0 s : S$ for some S. In general, the semantic information of subterms is lost. To support pattern matching on code, our semantic judgment must maintain both the syntactic structure of the code and the semantic information of all subterms. Therefore, our semantic judgments at layer 0 become inductively defined (Fig. 8). These rules are essentially just the typing rules with some extra $\Psi; \Gamma \Vdash_0 t : T$ premises. The inductive definition makes sure that the semantic information for all subterms are maintained.

Finally, we refer to $\Psi; \Gamma \Vdash_0 t : T$ when we define the gluing relation for $\Box(\Gamma \vdash T)$ at layer 1. This allows us to inspect the syntactic structure of t during an evaluation of pattern matching and access its semantic information at the same time. We refer readers to our technical report [27] for the proofs of the semantic rules for pattern matching. At layer 1, we use the new semantic

$$\frac{\Psi\ \mathtt{wf}^0 \qquad \Gamma\ \mathtt{wf}^0}{\Psi; \Gamma \Vdash_0 \cdot : \cdot} \qquad \frac{\Psi; \Gamma \Vdash_0 \delta : \Delta \qquad \Psi; \Gamma \Vdash_0 t : T}{\Psi; \Gamma \Vdash_0 \delta, t/x : \Delta, x : T} \qquad \frac{\Psi\ \mathtt{wf}^0 \qquad \Gamma\ \mathtt{wf}^0}{\Psi; \Gamma \Vdash_0 \mathsf{zero} : \mathtt{Nat}}$$

$$\frac{\begin{array}{c}\Psi; \Gamma \Vdash_0 t : \mathtt{Nat} \\ \Psi; \Gamma \VDash_0 \mathsf{succ}\ t : \mathtt{Nat}\end{array}}{\Psi; \Gamma \Vdash_0 \mathsf{succ}\ t : \mathtt{Nat}} \qquad \frac{\begin{array}{c}\Psi\ \mathtt{wf}^0 \quad \Gamma\ \mathtt{wf}^i \\ x : T \in \Gamma\end{array}}{\Psi; \Gamma \Vdash_0 x : T} \qquad \frac{\begin{array}{c}\Psi; \Gamma \Vdash_0 \delta : \Delta \\ u : (\Delta \vdash T) \in \Psi \quad \Psi; \Gamma \VDash_0 u^\delta : \Box(\Delta \vdash T)\end{array}}{\Psi; \Gamma \Vdash_0 u^\delta : \Box(\Delta \vdash T)}$$

$$\frac{\begin{array}{c}\Psi; \Gamma, x : S \Vdash_0 t : T \\ \Psi; \Gamma \VDash_0 \lambda x.t : S \longrightarrow T\end{array}}{\Psi; \Gamma \Vdash_0 \lambda x.t : S \longrightarrow T} \qquad \frac{\begin{array}{c}\Psi; \Gamma \Vdash_0 t : S \longrightarrow T \\ \Psi; \Gamma \Vdash_0 s : S \qquad \Psi; \Gamma \VDash_0 t\ s : T\end{array}}{\Psi; \Gamma \Vdash_0 t\ s : T}$$

Fig. 8: Layer-0 semantic judgment

judgment $\Psi; \Gamma \Vdash_0 t : T$ at layer 0 to define the semantic judgment for global substitutions $\Phi \Vdash \sigma : \Psi$, and then define the semantic judgment for terms and local substitutions:

$$\begin{aligned}
\Psi; \Gamma \Vdash_1 t : T &:= \forall\ \Phi; \Gamma \Vdash_0 \sigma : \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^1_{\Phi;\Delta} . \\
&\qquad t[\sigma; \delta] \sim [\![t]\!]^1_{\Phi;\Delta}(\sigma; \rho) \in (\!|T|\!)^1_{\Phi;\Delta} \\
\Psi; \Gamma \Vdash_1 \delta' : \Delta' &:= \forall\ \Phi; \Gamma \Vdash_0 \sigma : \Psi \text{ and } \delta \sim \rho \in (\!|\Gamma|\!)^1_{\Phi;\Delta} . \\
&\qquad \delta'[\sigma] \circ \delta \sim [\![\delta']\!]^1_{\Phi;\Delta}(\sigma; \rho) \in (\!|\Delta'|\!)^1_{\Phi;\Delta}
\end{aligned}$$

By proving and then instantiating the fundamental theorems, we obtain the soundness proof.

6 Future Extensions to Layered Systems

We have shown that 2-layered modal type theory supports intensional analysis and retains normalization. In this section, we build on our previous development and describe three possible extensions of layered systems as future work.

6.1 Extending to Complex Type Systems

In this paper, so far we only focused on simple types. Layering, however, is a powerful idea that, we believe, scales naturally. A natural extension is to consider System F, the foundation for many practical programming languages like Haskell and ML family, as the core language. Haskell and ML communities have expressed strong interest in meta-programming [49,59,58,34,35,53, etc.]. Layering provides a simple solution to this problem. In 2-layered System F, we replace validity of types with well-kindedness of types: $\Psi; \Gamma \vdash_i T : *$. Following the matryoshka principle, at layer 0, we operate in System F, while at layer 1, we work in a meta-language extending System F with one layer of $\Box$. We hope that 2-layered System F not only guarantees the well-scopedness and well-typedness of code, but is also normalizing, following our development here.

Besides System F, we are also interested in using Martin-Löf type theory (MLTT) as the base language. 2-layered MLTT would provide a foundation for tactic languages and meta-programming in proof assistants like Coq, Agda and

Lean. Following our previous development, we expect that 2-layered MLTT enables ① the reuse of *all* definitions from layer 0 at layer 1, and ② the guarantee of well-scopedness and well-typedness of all code. Since in MLTT types are also terms, we simply reuse contextual types $\square(\Gamma \vdash \texttt{Type})$ for code of types.

One challenge we expect from extending MLTT with layering is the semantics of code. For example, when we pattern match on code $t : (\lambda x.x)\ \texttt{Nat}$, we expect that the type is reduced to $\texttt{Nat}$. That is, $(\lambda x.x)\ \texttt{Nat}$ and $\texttt{Nat}$ as types are considered the same even for code. We effectively take quotient of code over types. This behavior aligns well with quotient inductive-inductive types (QIIT) [31,6] and we expect QIIT to appear in the semantics in some form, but we leave the detailed investigation as future work.

6.2 Extending Power of Layer 1

Though pattern matching allows inspection of code, not all operations can be defined easily in this way. For example, the weak head normal form reduction (whnf) on a term is not defined by a simple structural recursion on the syntax of the term. In 2-layered modal type theory, we can extend a whnf operation at layer 1 and still maintain normalization. The following are the rewrite rules for whnf and we can extend our previous normalization algorithm in Sec. 5 with a rewrite process [9,10]:

$$\mathsf{whnf}\ (\texttt{box}\ \mathsf{zero}) \rightsquigarrow \texttt{box}\ \mathsf{zero} \quad \mathsf{whnf}\ (\texttt{box}\ (\mathsf{succ}\ t)) \rightsquigarrow \texttt{box}\ (\mathsf{succ}\ t)$$
$$\mathsf{whnf}\ (\texttt{box}\ (\lambda x.t)) \rightsquigarrow \texttt{box}\ (\lambda x.t) \quad \mathsf{whnf}\ (\texttt{box}\ ((\lambda x.t)\ s)) \rightsquigarrow \mathsf{whnf}\ (\texttt{box}\ (t[s/x]))$$
$$\mathsf{whnf}\ (\texttt{box}\ x) \rightsquigarrow \texttt{box}\ x \quad \frac{\mathsf{whnf}\ (\texttt{box}\ t) \rightsquigarrow \mathsf{whnf}\ (\texttt{box}\ t')}{\mathsf{whnf}\ (\texttt{box}\ (t\ s)) \rightsquigarrow \mathsf{whnf}\ (\texttt{box}\ (t'\ s))}$$

whnf does not go under succ or λ and is only available at layer 1. Both local and global substitutions simply propagate under whnf. The rewrite process repeats these rules until no rule matches. This process will terminate due to the strong normalization of STLC and therefore the whole system remains terminating. However, with global variables, we must apply extra care to maintain confluence and eventually normalization. In Sec. 5.2, we discuss the impact of global substitutions and the necessity of their stability. When we extend layer 1 with another operation, we must also make sure that this extended operation is stable under global substitutions. When whnf encounters a global variable in the head position, such as $\mathsf{whnf}(\texttt{box}\ u^\delta)$ or $\mathsf{whnf}(\texttt{box}\ (u^\delta\ s))$, there is no matching rule and the rewrite process stops for the same reason for pattern matching stopping for $\texttt{box}\ u^\delta$. The lack of a reduction rule when a global variable is in the head position is particularly important. With whnf, we can now simplify a term before matching, which is a very useful and typical tactic in proof assistants:

$$\begin{array}{ll}\mathsf{match}\ \texttt{box}\ ((\lambda x.x)\ \mathsf{zero})\ \mathsf{with} & \mid \mathsf{zero} \Rightarrow \mathsf{true} \mid _ \Rightarrow \mathsf{false} \approx \mathsf{false} \\ \mathsf{match}\ \mathsf{whnf}\ (\texttt{box}\ ((\lambda x.x)\ \mathsf{zero}))\ \mathsf{with} & \mid \mathsf{zero} \Rightarrow \mathsf{true} \mid _ \Rightarrow \mathsf{false} \approx \mathsf{true}\end{array}$$

Due to layering, whnf only needs to consider terms in STLC at layer 0. In a homogeneous system, whnf must apply to all possible code, and thus becomes

troublesome to define. This extensibility of layer 1 is an important and useful feature for a foundation for meta-programming in proof assistants.

6.3 Extending to More Layers

Another potential of layering is to generalize the 2-layered system to n layers for a fixed $n > 2$. Scaling to n layers is in fact technically detailed, but conceptually simple. We sketch the process briefly here and leave the details as future work.

In a layered system, terms are type-checked in a *context array*. For an n-layered system, this context array has length n:

$$\Gamma_{n-1}; \cdots ; \Gamma_1; \Gamma_0 \vdash_i t : T \qquad \text{or} \qquad \overrightarrow{\Gamma} \vdash_i t : T \qquad \text{where } i \in [0, n-1].$$

We now use x, y to range over variables in all contexts. Each context in the context array contains bindings of a fixed shape. Bindings in Γ_0 are $x : T$. Bindings in Γ_i for $i \in [1, n-1]$ are of the shape $x : (\Delta_{i-1}; \cdots ; \Delta_0 \vdash T)$. Bindings in each Δ_j also have the specified shape. Contextual types are generalized to context arrays: $\square(\Delta_{i-1}; \cdots ; \Delta_0 \vdash T)$. The design of a n-layered system is guided by two principles: the matryoshka principle, which says types and terms at lower layers are subsumed by higher layers, and the static code principle, which only terms at layer $n-1$ compute. Particularly, the latter principle means that terms from layer 0 to $n-2$ are static code. Following both principles, we will be able to fill in the details and design an n-layered system.

We expect the n-layered generalization to be compatible with the extension with operations described in Sec. 6.2. Instead of extending layer 1, we extend layer $n-1$ so that all lower layers are unaffected.

7 Related Work and Conclusion

7.1 Normalization of Modal Type Theories

The core of our paper is the normalization of 2-layered modal type theory. Recently, there have been a number of approaches that explore modal type theories. One of the earliest is from Nanevski et al. [39]. They prove the normalization for contextual modal type theory (CMTT) indirectly by a translation to the system by de Groote [23]. This translation does not give a direct normalization algorithm for CMTT. Our system in Sec. 3 is strictly weaker than CMTT by disallowing nested $\square$s. Even if we scale our system to n layers as outlined in Sec. 6.3, the resulting system will only have a hierarchy of contextual types, so we still cannot recover the same expressive power as CMTT to do arbitrary nesting of $\square$s. Nevertheless, in Sec. 5, we have shown that this temporary loss in expressive power enables an orthogonal avenue of intensional analysis that is difficult to obtain in CMTT. Kavvos [32] proposes formulations for a few different modalities in the dual-context style and proves the normalization using a translation to de Groote's system as well. The normalization of System GL, however, is proved directly by reducibility candidates. Lately, Gratzer [20] proves the normalization

of multimodal type theory, a generalization of the dual-context systems, using Sterling's [51] synthetic Tait's computability.

The Kripke-style systems are another kind of formulation of modalities and is different from ours in context management. The normalization problem for this style is more intensively investigated. Borghuis [13] proves the strong normalization of modal pure type systems in his PhD thesis. More recently, Clouston [16] proves the normalization of System K using reducbility candidates. Gratzer et al.[22] prove the normalization of a dependent type theory with idempotent $S4$ by parameterizing Abel's [1] untyped domain method with a poset. Valliappan et al. [55] use the same method and prove the normalization for Systems K, T, $K4$ and $S4$. Hu and Pientka [26] establish the same result but introduce a "truncoid" algebraic structure to the untyped domain model instead, so that one normalization proof can be instantiated to adapt to all four systems. This method using truncoids has been scaled to dependent types [25]. It is worth emphasizing that none of these modal type theories supports pattern matching on code as we do in our 2-layer modal type theory.

7.2 Homogeneous Meta-programming and Its Foundations

Early ideas of metaprogramming using quasi-quoting style can be traced back to Lisp/Scheme [3]. In Lisp's untyped setting, all programs are represented as lists, so intensional analysis is reduced to inspections of lists and is relatively simple. Supporting type-safe metaprogramming leads to all the complications. MetaML [54] is an early example for type-safe meta-programming. MetaML employs a quasi-quoting style similar to Lisp. However, MetaML does not support any form of intensional analysis. In fact, MetaML's meta-theory even allows reduction of code under quote [52], so intensional analysis is deliberately avoided. The correspondence between meta-programming and modal logic S4 is described by Davies and Pfenning [17]. The correspondence explains how the modal logic $S4$ models multi-staging and code composition, but it does not explain how intensional analysis should be supported. Two formulations of $S4$ are presented: the dual-context style and the Kripke style. While the Kripke-style formulation provides a type-theoretic formulation for quasi-quotation, it is more challenging to extend and support intensional analysis. On the other hand, the dual-context style forces programmers to write meta-programs in a comonadic style, but it has better setup for intensional analysis as we have demonstrated. This is also the approach taken in Beluga [43,45] and Moebius [30].

The semantics for dual-context style has also been studied previously. In the context of contextual types, Gabbay and Nanevski [19] attempt to give a set-theoretic semantics to contextual types. As pointed out by Kavvos [33], their exact formulation of contextual types seems to break the confluence property.

Boespflug and Pientka [12] extend the dual-context style to the multi-context style. Though the multi-context style and the Kripke style both use multiple contexts for typing, the number of contexts in the former is more or less fixed (hence context arrays), while in the latter, contexts are often pushed and popped during typing (hence context stacks). Davies and Pfenning [17] show that the

Kripke style system is equivalent to the dual-context style. Moebius [30] combines the multi-context style and contextual types, and supports pattern matching on code in System F. Moebius has subject reduction. However, to adapt Moebius to a type theory, normalization must be proved, but it is not obvious how to support coverage. Whether layering provides a solution requires a future investigation.

*Ω*mega [56] is another example for a sound meta-programming system with pattern matching on code. *Ω*mega implements the quasi-quoting style. The open context of a code is annotated in the type, similar to contextual types. However, the type of the code itself is not remembered, so their type system is not as complex due to reduced type information.

7.3 Intensionality in Type Theories

Interest in intensionality is often associated with modalities. Pfenning [41] describes a type theory in which terms might be treated intensionally, extensionally, or irrelevantly when corresponding modalities are employed. This is similar to our setting, where intensionality of code is marked by the $\square$ modality. In the same setting, Kavvos [33] describes a special kind of intensional recursions using Löb induction ($(\square A \to A) \to A$), which supports meta-programs to access their own code. Löb induction says if we can prove A from the proof of A, then A is true. Löb induction is incompatible with the Axiom T ($\square A \to A$), but it still has interesting computational behaviors, including an example for computer viruses. Chen and Ko [14] resolve the incompatibility between the Löb induction and the Axiom T by supporting them in two separate modalities.

7.4 Layering in Type Theories

Layering can also be found in other type theories. Logical Framework (LF) [24] is essentially a layered system. LF is a dependently typed framework to define object languages. These object languages live at one layer. LF as their meta-language live at a higher layer. Isabelle [40] is one example for modern proof assistants based on LF. There are two layers in Cocon [47]. At the lower layer is LF, which defines an object language. At the higher layer is a Martin-Löf type theory (MLTT) for computation. Two layers are connected by contextual types. Cocon supports induction in MLTT on the syntax of an object language in LF. Cocon's structure leads to a similar semantics to our 2-layered modal type theory. The main difference is that in 2-layered modal type theory, the core language at layer 0 is a sub-language of the computational language at layer 1. Consequently, all terms at layer 0 can be lifted to layer 1 (Lemma 3) for free and be run as programs. The conversion from code to programs is done implicitly in the semantics. Contrarily, in Cocon, since the object language is defined freely, an embedding to MLTT must be given explicitly and is only possible if the object language has strictly weaker expressive power. A categorical semantics for Cocon is given by Hu et al.[46,29]. Kovács [36] and Allais [4] demonstrate applications of 2-level type theory which focuses more on code composition and does not support intensional analysis.

Our system uses layers to account for the number of nested $\Box$'s, which shares some similarity with graded and quantitative systems [8,2,38]. The latter systems use grades to keep track of uses of variables. We believe that it would be interesting to have a universal framework to contain all these different uses of modalities, though it requires further investigations.

Our approach is also similar to GuTT [21], a guarded type theory supporting Löb induction. GuTT has two layers. The first layer excludes dynamics of Löb induction (but not for other terms) and enjoys normalization. The lost dynamics is recovered at the second layer, at the cost of normalization. We are similar in that we both take advantage of differences between layers and one layer is the extension of the other.

7.5 Conclusion and Future Work

In this paper, we introduce the layered style to support intensional analysis in type theory. In the layered view, meta-programming is done in an extended language of a chosen core language. Pattern matching on code at a higher layer only needs to handle code at lower layers, hence circumventing the complications in previous work. We investigate the layered style in 2-layered modal type theory, which supports pattern matching on code where we guarantee coverage by construction. We provide a constructive proof of normalization by evaluation using a presheaf model. The normalization algorithm extracted from the model is proven complete and sound and is implemented in Agda.

Layering provides a controlled and modular way to introduce meta-programming with intensional analysis to type theory. As a first step, we plan to add context abstraction following the approach taken for example in Beluga to support more general recursion principles [43,44]. We see abstracting over contexts as an orthogonal issue. As a next step, we will adapt layering to Martin-Löf type theory (MLTT). Both extensions will create the first dependent type theory that is supporting intensional analysis of code within MLTT. In the long term, we hope that this type theory will also provide a foundation for extending the core language of Coq, Agda, or other proof assistants with a meta-language of tactics that can reuse *all* definitions from the core language while the normalization of the overall system is retained.

References

1. Abel, A.: Normalization by evaluation: dependent types and impredicativity. Habilitation thesis, Ludwig-Maximilians-Universität München (2013), `https://www.cse.chalmers.se/~abela/habil.pdf`
2. Abel, A., Bernardy, J.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP), 90:1–90:28 (2020), `https://doi.org/10.1145/3408972`
3. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs, Second Edition. MIT Press (1996)
4. Allais, G.: Scoped and typed staging by evaluation (2024), `https://arxiv.org/abs/2310.13413`

5. Altenkirch, T., Hofmann, M., Streicher, T.: Categorical reconstruction of a reduction free normalization proof. In: Pitt, D.H., Rydeheard, D.E., Johnstone, P.T. (eds.) Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings. Lecture Notes in Computer Science, vol. 953, pp. 182–199. Springer (1995), https://doi.org/10.1007/3-540-60164-3_27
6. Altenkirch, T., Kaposi, A.: Normalisation by evaluation for dependent types. In: Kesner, D., Pientka, B. (eds.) 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal. LIPIcs, vol. 52, pp. 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), https://doi.org/10.4230/LIPIcs.FSCD.2016.6
7. Anand, A., Boulier, S., Cohen, C., Sozeau, M., Tabareau, N.: Towards certified meta-programming with typed template-coq. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10895, pp. 20–39. Springer (2018), https://doi.org/10.1007/978-3-319-94821-8_2
8. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018), https://doi.org/10.1145/3209108.3209189
9. Berger, U., Eberl, M., Schwichtenberg, H.: Normalization by Evaluation. In: Möller, B., Tucker, J.V. (eds.) Prospects for Hardware Foundations: ESPRIT Working Group 8533 NADA — New Hardware Design Methods Survey Chapters, pp. 117–137. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (1998), https://doi.org/10.1007/3-540-49254-2_4
10. Berger, U., Eberl, M., Schwichtenberg, H.: Term rewriting for normalization by evaluation. Inf. Comput. **183**(1), 19–42 (2003), https://doi.org/10.1016/S0890-5401(03)00014-2
11. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda -calculus. In: [1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science. pp. 203–211 (Jul 1991). https://doi.org/10.1109/LICS.1991.151645
12. Boespflug, M., Pientka, B.: Multi-level contextual type theory. In: Geuvers, H., Nadathur, G. (eds.) Proceedings Sixth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2011, Nijmegen, The Netherlands, August 26, 2011. EPTCS, vol. 71, pp. 29–43 (2011), https://doi.org/10.4204/EPTCS.71.3
13. Borghuis, V.A.J.: Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus. PhD Thesis, Mathematics and Computer Science (1994), https://doi.org/10.6100/IR427575
14. Chen, L., Ko, H.: Realising intensional S4 and GL modalities. In: Manea, F., Simpson, A. (eds.) 30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference). LIPIcs, vol. 216, pp. 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), https://doi.org/10.4230/LIPIcs.CSL.2022.14
15. Christiansen, D.R., Brady, E.C.: Elaborator reflection: extending idris in idris. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan,

September 18-22, 2016. pp. 284–297. ACM (2016), https://doi.org/10.1145/2951913.2951932
16. Clouston, R.: Fitch-style modal lambda calculi. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10803, pp. 258–275. Springer (2018), https://doi.org/10.1007/978-3-319-89366-2_14
17. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM **48**(3), 555–604 (May 2001), https://dl.acm.org/doi/10.1145/382780.382785
18. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. **1**(ICFP), 34:1–34:29 (2017), https://doi.org/10.1145/3110278
19. Gabbay, M.J., Nanevski, A.: Denotation of contextual modal type theory (CMTT): syntax and meta-programming. J. Appl. Log. **11**(1), 1–29 (2013), https://doi.org/10.1016/j.jal.2012.07.002
20. Gratzer, D.: Normalization for multimodal type theory. In: Baier, C., Fisman, D. (eds.) LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022. pp. 2:1–2:13. ACM (2022), https://doi.org/10.1145/3531130.3532398
21. Gratzer, D., Birkedal, L.: A stratified approach to löb induction. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel. LIPIcs, vol. 228, pp. 23:1–23:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), https://doi.org/10.4230/LIPIcs.FSCD.2022.23
22. Gratzer, D., Sterling, J., Birkedal, L.: Implementing a modal dependent type theory. Proceedings of the ACM on Programming Languages **3**(ICFP), 107:1–107:29 (Jul 2019), https://doi.org/10.1145/3341711
23. de Groote, P.: On the Strong Normalization of Natural Deduction with Permutation-Conversions. In: Narendran, P., Rusinowitch, M. (eds.) Rewriting Techniques and Applications. pp. 45–59. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (1999), https://doi.org/10.1007/3-540-48685-2_4
24. Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. J. ACM **40**(1), 143–184 (1993), https://doi.org/10.1145/138027.138060
25. Hu, J.Z.S., Jang, J., Pientka, B.: Normalization by evaluation for modal dependent type theory. Journal of Functional Programming **33**, e7 (2023). https://doi.org/10.1017/S0956796823000060
26. Hu, J.Z.S., Pientka, B.: A Categorical Normalization Proof for the Modal Lambda-Calculus. Electronic Notes in Theoretical Informatics and Computer Science **Volume 1 - Proceedings of MFPS XXXVIII** (Feb 2023), https://entics.episciences.org/10360
27. Hu, J.Z.S., Pientka, B.: Layered modal type theories (2023), https://arxiv.org/abs/2305.06548
28. Hu, J.Z.S., Pientka, B.: Agda mechanization (2024), https://doi.org/10.5281/zenodo.10492818
29. Hu, J.Z.S., Pientka, B., Schöpp, U.: A category theoretic view of contextual types: From simple types to dependent types. ACM Trans. Comput. Log. **23**(4), 25:1–25:36 (2022), https://doi.org/10.1145/3545115

30. Jang, J., Gélineau, S., Monnier, S., Pientka, B.: Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022), https://doi.org/10.1145/3498700
31. Kaposi, A., Altenkirch, T.: Normalisation by Evaluation for Type Theory, in Type Theory. Logical Methods in Computer Science **Volume 13, Issue 4** (Oct 2017), https://lmcs.episciences.org/4005/pdf, publisher: Episciences.org
32. Kavvos, G.A.: Dual-context calculi for modal logic. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017. pp. 1–12. IEEE Computer Society (2017), https://doi.org/10.1109/LICS.2017.8005089
33. Kavvos, G.A.: Intensionality, intensional recursion and the gödel-löb axiom. FLAP **8**(8), 2287–2312 (2021), https://collegepublications.co.uk/ifcolog/?00050
34. Kiselyov, O.: The design and implementation of BER metaocaml - system description. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8475, pp. 86–102. Springer (2014), https://doi.org/10.1007/978-3-319-07151-0_6
35. Kokaji, Y., Kameyama, Y.: Polymorphic multi-stage language with control effects. In: Yang, H. (ed.) Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7078, pp. 105–120. Springer (2011), https://doi.org/10.1007/978-3-642-25318-8_11
36. Kovács, A.: Staged compilation with two-level type theory. Proc. ACM Program. Lang. **6**(ICFP), 540–569 (2022), https://doi.org/10.1145/3547641
37. Martin-Löf, P.: An Intuitionistic Theory of Types: Predicative Part. In: Rose, H.E., Shepherdson, J.C. (eds.) Studies in Logic and the Foundations of Mathematics, Logic Colloquium '73, vol. 80, pp. 73–118. Elsevier (Jan 1975), https://www.sciencedirect.com/science/article/pii/S0049237X08719451
38. Moon, B., III, H.E., Orchard, D.: Graded modal dependent type theory. In: Yoshida, N. (ed.) Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12648, pp. 462–490. Springer (2021), https://doi.org/10.1007/978-3-030-72019-3_17
39. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic **9**(3), 23:1–23:49 (Jun 2008), https://doi.org/10.1145/1352582.1352591
40. Paulson, L.C.: Natural deduction as higher-order resolution. J. Log. Program. **3**(3), 237–258 (1986), https://doi.org/10.1016/0743-1066(86)90015-4
41. Pfenning, F.: Intensionality, extensionality, and proof irrelevance in modal type theory. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 221–230. IEEE Computer Society (2001), https://doi.org/10.1109/LICS.2001.932499
42. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science **11**(04) (Aug 2001), http://www.journals.cambridge.org/abstract_S0960129501003322
43. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Pro-

gramming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 371–382. ACM (2008), https://doi.org/10.1145/1328438.1328483
44. Pientka, B., Abel, A.: Well-Founded Recursion over Contextual Objects. In: Altenkirch, T. (ed.) 13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 38, pp. 273–287. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015), http://drops.dagstuhl.de/opus/volltexte/2015/5169
45. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In: Antoy, S., Albert, E. (eds.) Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain. pp. 163–173. ACM (2008), https://doi.org/10.1145/1389449.1389469
46. Pientka, B., Schöpp, U.: Semantical Analysis of Contextual Types. In: Goubault-Larrecq, J., König, B. (eds.) Foundations of Software Science and Computation Structures. pp. 502–521. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-45231-5_26
47. Pientka, B., Thibodeau, D., Abel, A., Ferreira, F., Zucchini, R.: A type theory for defining logics and proofs. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–13. IEEE (2019), https://doi.org/10.1109/LICS.2019.8785683
48. Schürmann, C., Despeyroux, J., Pfenning, F.: Primitive recursion for higher-order abstract syntax. Theor. Comput. Sci. **266**(1-2), 1–57 (Sep 2001), http://dx.doi.org/10.1016/S0304-3975(00)00418-7
49. Sheard, T., Jones, S.P.: Template meta-programming for haskell. In: Chakravarty, M.M.T. (ed.) Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, Pittsburgh, Pennsylvania, USA, October 3, 2002. pp. 1–16. ACM (2002), https://doi.org/10.1145/581690.581691
50. Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., Winterhalter, T.: The metacoq project. J. Autom. Reason. **64**(5), 947–999 (2020), https://doi.org/10.1007/s10817-019-09540-0
51. Sterling, J.: First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory. Ph.D. thesis, Carnegie Mellon University, USA (2022), https://doi.org/10.1184/r1/19632681.v1
52. Taha, W.: A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of metaml is non-trivial (extended abstract). In: Lawall, J.L. (ed.) Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22-23, 2000. pp. 34–43. ACM (2000), https://doi.org/10.1145/328690.328697
53. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 203–217. PEPM '97, Association for Computing Machinery, New York, NY, USA (Dec 1997), https://doi.org/10.1145/258993.259019
54. Taha, W., Sheard, T.: Metaml and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2), 211–242 (2000), https://doi.org/10.1016/S0304-3975(00)00053-0
55. Valliappan, N., Ruch, F., Tomé Cortiñas, C.: Normalization for fitch-style modal calculi. Proc. ACM Program. Lang. **6**(ICFP), 772–798 (2022), https://doi.org/10.1145/3547649

56. Viera, M., Pardo, A.: A multi-stage language with intensional analysis. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. pp. 11–20. ACM (2006), https://doi.org/10.1145/1173706.1173709
57. van der Walt, P., Swierstra, W.: Engineering proof by reflection in agda. In: Hinze, R. (ed.) Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8241, pp. 157–173. Springer (2012), https://doi.org/10.1007/978-3-642-41582-1_10
58. Xie, N., Pickering, M., Löh, A., Wu, N., Yallop, J., Wang, M.: Staging with class: a specification for typed template haskell. Proc. ACM Program. Lang. **6**(POPL), 1–30 (2022), https://doi.org/10.1145/3498723
59. Yallop, J.: Staged generic programming. Proc. ACM Program. Lang. **1**(ICFP), 29:1–29:29 (2017), https://doi.org/10.1145/3110273
60. Zeilberger, N.: Focusing and higher-order abstract syntax. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 359–369. ACM (2008), https://doi.org/10.1145/1328438.1328482
61. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A monad for typed tactic programming in coq. J. Funct. Program. **25** (2015), https://doi.org/10.1017/S0956796815000118