

# mitten: a flexible multimodal proof assistant

Philipp Stassen ✉ 

Aarhus University

Daniel Gratzer ✉ 

Aarhus University

Lars Birkedal ✉ 

Aarhus University

---

## Abstract

Recently, there has been a growing interest in type theories which include *modalities*, unary type constructors which need not commute with substitution. Here we focus on MTT [12], a general modal type theory which can internalize arbitrary collections of (dependent) right adjoints [5]. These modalities are specified by mode theories [17], 2-categories whose objects corresponds to modes, morphisms to modalities, and 2-cells to natural transformations between modalities. We contribute a defunctionalized NbE algorithm which reduces the type-checking problem for MTT to deciding the word problem for the mode theory. The algorithm is restricted to the class of *preordered* mode theories—mode theories with at most one 2-cell between any pair of modalities. Crucially, the normalization algorithm does not depend on the particulars of the mode theory and can be applied without change to any preordered collection of modalities. Furthermore, we specify a bidirectional syntax for MTT together with a type-checking algorithm. We further contribute *mitten*, a flexible experimental proof assistant implementing these algorithms which supports all decidable preordered mode theories without alteration.

**2012 ACM Subject Classification** Theory of computation → Type theory; Theory of computation → Modal and temporal logics

**Keywords and phrases** Dependent type theory, guarded recursion, modal type theory, proof assistants

**Digital Object Identifier** 10.4230/LIPIcs...

## 1 Introduction

A fundamental benefit of using type theory is the possibility of working within a proof assistant, which can check and even aid in the construction of complex theorems. Implementing a proof assistant, however, is a highly nontrivial task. In addition to a solid theoretical foundation for the particular type theory, numerous practical implementation issues must be addressed.

Recently, interest has gathered around type theories with *modalities*, unary type constructors which need not commute with substitution. Unfortunately, the situation for modal type theories is even more fraught; the theory for modalities is poorly understood in general, and it is unknown whether standard implementation techniques extend to support them.

Despite these challenges, mainstream proof assistants have begun to experiment with modalities [23], but these implementations are costly and only apply to a particular modal type theory. In practice, a type theorist may use a particular collection of modalities for only one proof or construction and it is impractical to invest in a specialized modal proof assistant each time. This churn has pushed type theorists to define *general* modal type theories which can be instantiated to a variety of modal situations [18, 13].

We choose to focus on MTT [12], a general modal type theory which can internalize an arbitrary collection of modalities so long as they behave like *right adjoints* [5]. Despite limiting consideration to right adjoints, MTT can be used to model a variety of existing modal type theories including calculi for guarded recursion, internalized parametricity, and axiomatic cohesion. Better still, MTT has a robustly developed metatheory [12, 10] which



© Philipp Stassen, Daniel Gratzer, and Lars Birkedal;  
licensed under Creative Commons License CC-BY 4.0



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

applies *irrespective* of the chosen modalities. An implementation of MTT could therefore conceivably be designed to allow the user to freely change the collection of modalities without re-implementing the entire proof assistant each time. This, in turn, enables the kind of specialized modal proof assistants previously impractical for one-off modal type theories.

## 1.1 MTT: a general modal type theory

As mentioned, MTT can be instantiated with a collection of modalities. More precisely, MTT is parameterized by a mode theory, a strict 2-category which describes a modal situation. Intuitively, objects  $(m, n, o)$  of this mode theory represent distinct type theories which are then connected by 1-cells  $(\mu, \nu, \xi)$  which describe the modalities. The categorical structure ensures that modalities compose and that there is an identity modality. In order to describe more intricate connections and structure, the mode theory also contains 2-cells  $(\alpha, \beta)$ . A 2-cell induces a ‘natural transformation’ between modalities. By carefully choosing 2-cells we can force a modality to e.g. become a comonad, a monad, or an adjoint.

To give a paradigmatic example, consider the mode theory  $\mathcal{M}$  with a single object  $m$ , a single non-identity morphism  $\mu : m \rightarrow m$  and a 2-cell  $\epsilon : \mu \rightarrow \text{id}_m$  subject to the equations  $\mu \circ \mu = \mu$  and  $\epsilon \star \mu = \mu \star \epsilon$ . This description defines  $\mathcal{M}$  as a 2-category with a strictly idempotent comonad  $\mu$ . Instantiating MTT with this mode theory yields a modality  $\langle \mu \mid - \rangle$  together with definable operations shaping  $\langle \mu \mid - \rangle$  into an idempotent comonad:

$$\text{extract}_A : \langle \mu \mid A \rangle \rightarrow A \qquad \text{dup}_A : \langle \mu \mid A \rangle \simeq \langle \mu \mid \langle \mu \mid A \rangle \rangle$$

Even this simple modal type theory is quite useful; it can serve as a replacement for the experimental version of Agda [23] used to formalize a construction of univalent universes [16].

Given the generality, it is natural to wonder whether instantiating MTT yields a calculus which is feasible to work with in practice. Fortunately, prior Fitch-style type theories have been highly workable [2, 3, 22] and this trend has continued with MTT [12, 10, 11].

## 1.2 From theory to practice

Unfortunately, converting the theoretical guarantee of normalization into an executable program is not a small step. A first obstacle is the syntax of MTT itself: prior work has exclusively considered an algebraic presentation of the syntax as a generalized algebraic theory. While mathematically elegant, a proof assistant requires a more streamlined and ergonomic syntax. Once a more convenient syntax has been designed, one must adapt the normalization proof to a normalization algorithm. Normalization is proven by a sophisticated *gluing* argument, and while the proof is reminiscent of normalization-by-evaluation [1] it remains to extract such an algorithm. Finally, the normalization algorithm does not give any insight into representing common mode theories or solving their word problems.

### Restriction to preordered mode theories

Many difficulties flow not from the modalities per se, but from the 2-cells of our mode theory, which induce a new primitive type of substitutions. During normalization these *key substitutions* accumulate at variables. Unfortunately, they disrupt a crucial property of modern NbE algorithms: variables can no longer be presented in a way that is *invariant* under weakening. Therefore, we restrict our attention to mode theories that are *preordered*, with at most one 2-cell between any pair of modalities.

This allows us to present a syntax that never talks about 2-cells and relies entirely on the elaboration procedure to insert and check 2-cells. In addition to avoiding annotations, this simplifies the normalization algorithm since the troublesome key substitutions trivialize.

Although such a restriction does preclude some examples, preordered mode theories are still expressive enough to model guarded recursion together with an *everything now* modality similar to the one introduced by Clouston et al. [8].

## A surface syntax for MTT

As a generalized algebraic theory, MTT is presented with explicit substitutions and fully annotated connectives [12]. In order to avoid this bureaucracy, we introduce a bidirectional version of MTT which allows a user of `mitten` to omit almost all type annotations [9].

## Normalization-by-evaluation

The normalization proof for MTT follows the structure of a normalization-by-evaluation proof. Rather than fixing a rewriting system, a term is *evaluated* into a carefully chosen semantics equipped with a quotation function reifying an element of the semantic domain to a normal form. The entire normalization function is then a round-trip from syntax to semantics and then back to normal forms. While the proof of normalization uses a traditional denotational model for a semantic domain, this approach is unsuitable for implementation.

Instead `mitten` follows the literature on normalization-by-evaluation and uses a *defunctionalized* and *syntactic* semantic domain [1]. This approach has previously been adapted to work with particular a modal type theories [14, 15].

## Mode theories

As mentioned previously, normalization for MTT does not immediately imply the decidability of type equality. Terms (and therefore types) mention both 1- and 2-cells from the mode theory, and deciding the mode theory is a necessary precondition for deciding type equality. Moreover, deciding the equality of 1- and 2-cells, even in a finitely presented 2-category, is well-known to be undecidable.<sup>1</sup> For us, this situation is slightly improved since for preordered mode theories at least 2-cell equality is trivial. Unfortunately, the undecidability of 1-cell equality remains. Special attention is therefore necessary for each mode theory to ensure that the normalization algorithm for MTT is sufficient to yield a type-checker.

While this rules out a truly generic proof assistant for MTT which works regardless of the choice of mode theory, `mitten` shows that the best theoretically possible result is obtainable. We implement `mitten` to be parameterized by a module describing the mode theory so that the type-checker relies only on the existence of such a decision procedure. In particular, there is no need to alter the entire proof assistant when changing the mode theory; only a new mode theory module is necessary. Crucially, while the user must write a small amount of code, no specialized knowledge of proof assistants is required.

We have implemented several mode theories commonly used with MTT in this way, showing that in practice decidability is no real obstacle. For instance, we have configured `mitten` to support guarded recursion with a combination of two modalities  $\square$  and  $\blacktriangleright$ . This is the first proof assistant to support this combination of modalities.

---

<sup>1</sup> The word problem is well-known to be undecidable for finitely presented groups which can be realized as finitely-presented categories and therefore locally discrete finitely-presentable 2-categories.

### 1.3 Contributions

We contribute a bidirectional syntax for MTT (restricted to preordered mode theories) together with a defunctionalized normalization-by-evaluation algorithm which reduces the type-checking problem to deciding the word problem for the mode theory. We have put these results into practice with `mitten`, an prototype implementation of MTT based on this algorithm. `mitten` also supports the replacement of the underlying mode theory with minimal alterations, allowing a user to construct specialized proof assistants for modal type theories by merely supplying a single module specifying the mode theory together with equality functions for 0-, 1-, and 2-cells.

In Section 2 we provide a guided tour of MTT. This section also introduces the bidirectional syntax for MTT and shows how even in this general setting the modalities introduce minimal overhead. Section 3 introduces the defunctionalized normalization algorithm for non-specialists and Sections 4 and 5 completes the description of the core components of `mitten` by describing the type-checking algorithm. In so doing, we also describe the novel interface `mitten` uses to represent modalities and show how this interface is implemented.

In Section 6 we discuss the realization of mode theories with a representative example: guarded recursion. As previously mentioned, this is the first proof assistant able to support this pair of modalities simultaneously.

## 2 A surface syntax for MTT

Prior to specifying a type-checking algorithm for MTT, we must specify the surface syntax for the language. This question is not satisfactorily addressed in the prior work on MTT; the *generalized algebraic* version of syntax is too verbose to be workable, but the informal pen-and-paper syntax which omits all type annotations cannot be type-checked. Our surface syntax is formulated with an eye towards the type-checking algorithm we will eventually use: a version of Coquand’s semantic type-checker [9]. In particular, we will employ a bidirectional surface syntax which minimizes the number of mandatory annotations while still ensuring the decidability of type-checking.

To a first approximation, the surface syntax is divided into two components: checkable and synthesizable terms. Checkable terms include introduction forms while synthesizable terms include elimination forms and variables. By carefully controlling where checkable and synthesizable terms are used, we thereby avoid unnecessary type annotations.

We present the grammar for surface syntax in Section 2.1. While we will defer presenting the actual type-checking algorithm until Section 5, in order to make this account as self-contained as possible we provide an example-based introduction to MTT in Section 2.2.

### 2.1 Bidirectional Syntax

As previously mentioned, MTT is parameterized by a mode theory [17] which specifies the modes and modalities of the type theory. We begin by more precisely defining a mode theory in our situation.

► **Definition 1.** *A mode theory is a category whose objects  $m, n, o$  we refer to as modes and whose morphisms  $\mu, \nu$  we refer to as modalities. We further require that each hom-set be equipped with a pre-order  $\leq$  compatible with composition. Explicitly, given  $\mu, \nu \in \text{Hom}(m, n)$  and  $\rho, \sigma \in \text{Hom}(n, o)$  with  $\mu \leq \nu$  and  $\rho \leq \sigma$  we require  $\rho \circ \mu \leq \sigma \circ \nu$ .*

*Equivalently, a mode theory is a preorder-enriched category.*

For the remainder of this subsection, we fix a mode theory  $\mathcal{M}$ . The grammar of the surface syntax is presented below:

$$\begin{array}{ll}
\text{(Checkable)} & A, M, N, C ::= R \mid (\mu \mid A) \rightarrow B \mid A \times B \mid \text{Nat} \mid \mathbb{U} \mid \lambda(M) \mid (M, N) \mid \text{zero} \\
& \mid \text{succ}(M) \mid \langle \mu \mid A \rangle \mid \text{mod}_\mu(M) \mid \text{Id}_A(M, N) \mid \text{refl}_M \\
\text{(Synthesizable)} & R, S ::= M : A \mid \mathbf{v}_k \mid R(M)_\mu \mid \text{pr}_1(R) \mid \text{pr}_2(R) \\
& \mid \text{let}_\mu \text{ mod}_\nu(\_) \leftarrow R \text{ in } M \text{ over } C \mid \text{rec}(C, M_{\text{zero}}, M_{\text{suc}}, N) \\
& \mid \text{J}(C, c_{\text{refl}}, M)
\end{array}$$

As mentioned previously, checkable terms consist essentially of introduction forms while synthesizable terms are elimination principles. For instance, the presentation of dependent sums above includes  $A \times B$  and  $(M, N)$  as checkable terms while  $\text{pr}_i(R)$  is synthesizable.

By stratifying terms in this way we ensure that annotations are required exactly where ambiguity would arise during type-checking. For instance, this stratification prevents unannotated  $\beta$ -redexes from occurring. Consider again the case of dependent sums. In order to apply a projection to an element  $(M, N)$  of dependent sum type, the element must be synthesizable. However, since  $(M, N)$  is checkable, the only way to represent  $\text{pr}_1((M, N))$  in this discipline is to promote  $(M, N)$  to a synthesizable term by annotating it:  $(M, N) : A \times B$ .

► **Remark 2.** In particular, terms in  $\beta$ -normal and  $\eta$ -long form fit into this surface syntax with no additional annotations. Consequently, the normalization theorem for MTT [10] ensures that any term is convertible to one expressible in the surface syntax.

► **Remark 3.** We have made a concession to simplicity and used de Bruijn indices for variables rather than names. This makes the normalization and type-checking algorithms far easier to specify and it is well-known how to pass between syntax with named variables and de Bruijn indices. We will use named variables in examples e.g.,  $\text{let}_\mu \text{ mod}_\nu(y) \leftarrow R \text{ in } M \text{ over } x. C$  or  $(\mu \mid x : A) \rightarrow B$  for modal elimination and dependent products respectively.

## 2.2 The surface syntax by example

We will crystallize when a term in the surface syntax is well-formed in Section 5 when presenting the type-checking algorithm. In order to cultivate intuition for the theory before this, we will now work through several examples in the language.

► **Remark 4.** We refer the reader to Gratzer et al. [12] for a long form explanation of MTT.

### MTT with one mode and one generating modality

Consider MTT instantiated with the mode theory with one mode  $m$  and one modality  $\phi$  with no non-trivial equations or inequalities. Then each modality  $\mu$  is uniquely expressible as  $\phi^n$ , the composition of  $n$  copies of  $\phi$ . Just as in ordinary type theory, MTT then has dependent sums, natural numbers, identity types, and their behavior is unchanged.

Unlike in ordinary type theory, each variable is annotated with a modality  $x : (\mu \mid A)$  (pronounced  $x : A$  annotated by  $\mu$ ). Variables annotated with the identity modality behave like ‘ordinary’ variables; they can be used freely when working with e.g. natural numbers. Conversely, variables annotated with  $\phi^{n+1}$  cannot be used except in the construction of an element the modal type  $\langle \phi \mid A \rangle$ .

An element of  $\langle \phi \mid A \rangle$  is introduced by  $\text{mod}_\phi(M)$ , where  $M$  is an element of  $A$ , subject to the restriction that  $M$  may only use variables with annotation  $\phi^{n+1}$ . More concretely, when we construct  $M$  we (1) lose access to all id-annotated variables and (2) replace a variable  $x : (\phi^{n+1} \mid A)$  with  $x : (\phi^n \mid A)$ . As only variables with identity annotation can be used with the variable rule, this means that within  $\text{mod}_\phi(-)$  we may use  $\phi$ -annotated variables freely.

For instance, in the context with variables  $x_0 : (\text{id} \mid \text{Nat})$ ,  $x_1 : (\phi \mid \text{Nat})$ , and  $x_2 : (\phi \circ \phi \mid \text{Nat})$  the following programs are well-typed:

$$x_0 : \text{Nat} \quad \text{mod}_\phi(x_1) : \langle \phi \mid \text{Nat} \rangle \quad \text{mod}_\phi(\text{mod}_\phi(x_2)) : \langle \phi \mid \langle \phi \mid \text{Nat} \rangle \rangle$$

On the other hand, both  $x_1 : \text{Nat}$  and  $\text{mod}_\phi(x_0) : \langle \phi \mid \text{Nat} \rangle$  are ill-typed as the annotations on variables do not match their usage.

This idea generalizes: to construct an element of  $\langle \phi^k \mid A \rangle$  we use  $\text{mod}_{\phi^k}(M)$  where  $M : A$  in a context where we have (1) lost access to variables with annotations  $\phi^l$  where  $l < k$  (2) replaced each variable  $x : (\phi^{n+k} \mid A)$  with  $x : (\phi^n \mid A)$ . In the same context as the example above therefore,  $\text{mod}_{\phi \circ \phi}(x_2) : \langle \phi \circ \phi \mid \text{Nat} \rangle$ . We refer to the modification to the context given by (1) and (2) as  $\phi^k$ -restricting the context.

Let us now consider the modal function type  $(\mu \mid A) \rightarrow B$ . An element of  $(\mu \mid A) \rightarrow B$  is precisely a function which binds a variable of type  $A$  with annotation  $\mu$ . Application for these function types  $R(M)_\mu$  takes  $\mu$  into account in the following way:  $R(M)_\mu : B$  if (1)  $R$  has type  $(\mu \mid A) \rightarrow B$  and (2) after  $\mu$ -restricting the context,  $M$  has type  $A$ .

One feature remains to be discussed, the elimination principle for modal types:

$$\text{let}_\mu \text{ mod}_\nu(y) \leftarrow R \text{ in } M \text{ over } x. C$$

To a first approximation, this principle allows us to replace a variable  $x : (\nu \mid \langle \mu \mid A \rangle)$  with  $y : (\nu \circ \mu \mid A)$ . More precisely,  $\text{let}_\mu \text{ mod}_\nu(y) \leftarrow R \text{ in } M \text{ over } C : C[M/x]$  if (1) after binding  $x : (\nu \mid \langle \mu \mid A \rangle)$ ,  $C$  is a type (2) after  $\nu$ -restricting the context  $M$  has type  $\langle \mu \mid A \rangle$  and (3) after binding  $y : (\nu \circ \mu \mid A)$ ,  $R$  has type  $C[\text{mod}_\mu(y)/x]$ .

## Multiple modalities

The above approach for  $\phi$ -restriction based on decrementing modal annotations provides a simple mental model for MTT. To extend these ideas to more complex mode theories, however, a more refined approach is necessary. We begin by discussing a small adjustment to the concepts introduced previously.

Rather than eagerly decrementing the annotation on a variable when we restrict a context, we instead *lazily* perform this update. Accordingly, we annotate each variable with a pair of modalities and write  $x :_{\mu/\nu} A$  for a  $\mu$ -annotated variable with a  $\nu$ -restriction lazily performed upon it. The rule for applying a restriction to a variable now becomes more uniform: to restrict  $x :_{\mu/\nu} A$  by  $\xi$  we replace it with  $x :_{\mu/\nu \circ \xi} A$ . The variable rule applies only when the fraction ‘cancels’ i.e.,  $x :_{\mu/\mu} A \vdash x : A$ .

For the mode theory under consideration, this is merely a change in notation as the behavior of the annotations of  $x :_{\phi^l/\phi^k} A \vdash x : A$  is entirely determined by the difference  $l - k$ . We therefore introduce the following mode theory to illustrate the need for the ‘lazy’ approach:

► **Definition 5.** Denote by  $\mathcal{M}_1^{\text{ex}}$  the mode theory with one mode and two generating modalities  $\psi$  and  $\phi$ . The preorder is generated by the inequality  $\psi \circ \psi \leq \phi$ .

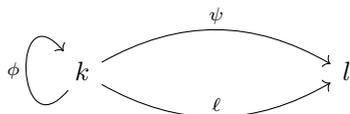
This mode theory introduces two new concepts simultaneously: multiple modalities and non-trivial inequalities between those modalities. Fortunately, to refine the idea explained above of  $\mu$ -restricting a context, only one rule must be altered: To account for the preorder on modalities, we relax the variable rule slightly:  $x :_{\mu/\nu} A \vdash x : A$  if  $\mu \leq \nu$ . With this modified rule, we can construct a coercion  $\langle \psi \circ \psi \mid A \rangle \rightarrow \langle \phi \mid A \rangle$ :

$$\text{coerce} = \lambda x. \text{let}_{\text{id}} \text{ mod}_{\psi \circ \psi}(y) \leftarrow x \text{ in } \text{mod}_\phi(y) \text{ over } \_. \langle \mu \mid A \rangle$$

## Multiple modes and multiple modalities

Only one generalization is required at this point to provide a complete description of MTT: multiple modes. While thus far we have confined ourselves to discussing multiple modalities on one mode, we are allowed to have multiple modes in MTT as well. Consider the following mode theory:

► **Definition 6.**  $\mathcal{M}_2^{\text{ex}}$  is the mode theory equipped with two modes  $k$  and  $l$  whose modalities are generated by  $\phi : k \rightarrow k$  and  $\psi, \ell : k \rightarrow l$ . The preorder on hom-sets are generated by the inequalities  $\text{id}_k \leq \phi$  and  $\ell \leq \psi$ :



We note that now  $\mathcal{M}_2^{\text{ex}}$  now has two different modes  $k$  and  $l$ . Each mode in MTT gives rise to a separate type theory so that we must check not only that some term has a type, but also that the term, type, and all variables in scope live at the correct mode.

All of the standard constructions do not change the mode; thus, e.g.,  $\text{succ}(n)$  will be well-typed at type  $\text{Nat}$  at mode  $m$  just when the same is true of  $n$ . We will notate “ $M$  has type  $A$  at mode  $m$ ” by  $M : A @ m$ . Prior to discussing the two type constructors involving modalities, we must explain what it means for a context to be well-formed at mode  $m$ .

► **Definition 7.** A variable declaration  $x :_{\mu/\nu} A$  is well-formed at mode  $m$  if the following hold:

1.  $\mu : n \rightarrow o$  and  $\nu : m \rightarrow o$  for some  $o$ .
2.  $A$  is a type at mode  $n$ .

The context is well-formed at mode  $m$  if all variables in scope are well-formed at mode  $m$ .

► **Example 8.** Restricting a well-formed context at  $m$  by  $\mu : n \rightarrow m$  yields a well-formed context in mode  $n$ .

It is worth emphasizing the contravariant nature of the restriction  $\nu$  in  $x :_{\mu/\nu} A$ . This is crucial for the rules governing  $\langle \mu | A \rangle$ . The type  $\langle \mu | A \rangle$  is well-formed at mode  $m$  if (1)  $\mu : n \rightarrow m$  for some  $n$  and (2) after  $\mu$ -restricting the context,  $A$  is well-formed at mode  $n$ . In particular,  $\langle \mu | - \rangle$  sends types at mode  $n$  to types at mode  $m$  so restriction must move contexts contravariantly from mode  $n$  to mode  $m$ . We remark, however, that aside from the additional checks to ensure that types are well-moded, this is the same rule as given previously. Likewise, the rules for introduction and elimination along with all of those for modal dependent products are merely instrumented with additional checks to ensure that types and terms live at the correct mode.

We conclude with a few examples.

► **Example 9.**  $\lambda x.x : (\ell | A) \rightarrow \langle \psi | A \rangle @ l$  is well typed. In particular, since  $\ell \leq \psi$  we conclude  $x :_{\ell/\psi} A \vdash x : A @ k$ .

► **Example 10.** We will define a function of the following type:

$$f : \langle \psi | \langle \phi | \text{Nat} \rangle \rangle \rightarrow \langle \psi \circ \phi | \text{Nat} \rangle @ l$$

We begin by binding a variable  $x :_{\text{id}/\text{id}} \langle \psi | \langle \phi | \text{Nat} \rangle \rangle$  so it now suffices to construct a term  $\langle \psi \circ \phi | \text{Nat} \rangle @ l$ . To this end, we use the modal elimination principle on  $x$  to obtain a new variable  $y :_{\psi/\text{id}} \langle \phi | \text{Nat} \rangle$ . Applying modal elimination to  $y$ , we obtain  $z :_{\psi \circ \phi/\text{id}} \text{Nat}$ .

## XX:8 mitter: a flexible multimodal proof assistant

We still wish to construct a term  $\langle \psi \circ \phi \mid \text{Nat} \rangle$ . Applying the modal introduction rule, we  $\psi \circ \phi$  restrict the context (so  $y$  becomes  $y :_{\psi \circ \phi / \psi \circ \phi} \text{Nat}$ ). Our goal is then  $\text{Nat}$ , so  $y$  suffices. All told, the term final term is as follows:

```
λx.  
  letidk modψ(y) = x in  
    letψ modφ(z) = y in  
      modψ ∘ φ(z)  
    over  $\langle \psi \circ \phi \mid \text{Nat} \rangle$   
  over  $\langle \psi \circ \phi \mid \text{Nat} \rangle$ 
```

### 3 Normalization by Evaluation

A crucial ingredient of any type checker is a procedure for determining when two types are equal. In `mitter`, we have implemented this decision procedure through a normalization algorithm: a function which sends a term to a corresponding *normal form*. The precise definition of normal form is then less important than the fact that definitional equality for normal forms is straightforward to decide. Writing `NfTerms` for the collection of normal forms, we view our normalization algorithm as a function:

**`normΓ`** : Syntax → NfTerms

Merely having a function from syntax to normal forms, however, is insufficient to decide definitional equality. Accordingly, we are interested in normalization functions which satisfy the following properties:

► **Definition 11.** A normalization function is called *complete* if  $\Gamma \vdash A = B @ m$  implies  $\text{norm}_{\Gamma}(A) = \text{norm}_{\Gamma}(B)$

► **Definition 12.** A normalization function is *sound* if  $\Gamma \vdash A @ m$  implies  $\Gamma \vdash \text{norm}_{\Gamma}(A) = A @ m$ .

Completeness states that normalization lifts to a function on syntax quotiented by definitional equality while soundness states that this induced function has a section. Taken together, therefore, we have the following:

► **Corollary 13.** Let  $\text{norm}_{\Gamma}$  be sound and complete then  $\Gamma \vdash A = B @ m$  if and only if  $\text{norm}_{\Gamma}(A) = \text{norm}_{\Gamma}(B)$ .

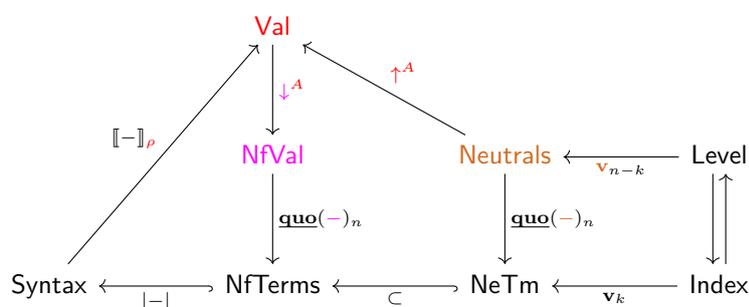
The traditional approach to constructing a normalization function is to specify an abstract rewriting system which directs and presents the equational theory. Equality of terms is then convertibility within this rewriting system so that strong normalization ensures both soundness and completeness. This approach, however, turns out to be unworkable for more elaborate dependent type theories with type-directed rules. We therefore adopt an entirely different approach to associating terms to normal forms: *normalization by evaluation* (NbE).

Normalization by evaluation breaks the process of normalizing a term into two distinct phases: evaluation and quotation. The first *evaluates* a term into a *semantic domain*. Historically and in certain contexts, semantic domains were drawn from genuine semantic models. For the purposes of implementation, however, the semantic domain is simply a more restrictive form of syntax which disallows  $\beta$ -reducible terms. The process of evaluation

boils down to placing a term in  $\beta$ -normal form while crucially retaining various pieces of type information for the next phase. The second phase, quotation, takes an element of the semantic domain and *quotes* it back to syntax. In the process, however, it  $\eta$  expands terms wherever possible. As a result, the full loop of evaluation and quotation sends a term to its  $\beta$ -normal  $\eta$ -long form as required. Figure 1 gives a graphical overview of the process.

We describe the semantic domain in detail in Section 3.1. The actual algorithm is described over the following three sections (Sections 3.2–3.4). Our algorithm is inspired by Gratzer’s gluing-based argument for normalization [10] and we conjecture that this link can be made sufficiently precise to establish the soundness and completeness of our code.

► **Remark 14.** The version of normalization-by-evaluation we use is robust enough to require only local modifications in order to accommodate modal types. Accordingly, we focus primarily on connectives like dependent products and modal types whose behavior is impacted and refer the reader to, e.g., Abel [1] for a description how the algorithm works on the remaining connectives.



■ **Figure 1** Overview of the algorithm inspired by [14] and [1].

### 3.1 The Domain

We start by a brief overview of the semantic domains described in Figure 1:

(values)	$A, u$	$::=$	$\uparrow^A e \mid \lambda(f) \mid (\mu \mid A) \rightarrow B \mid \text{zero} \mid \text{suc}(v) \mid \text{Nat} \mid (v_1, v_2) \mid A \times B$ $\mid \langle \mu \mid A \rangle \mid \text{mod}_\mu(v)$
(neutrals)	$e$	$::=$	$\mathbf{v}_k \mid \text{app}[\mu](e, d) \mid \text{pr}_1(e) \mid \text{pr}_2(e) \mid \text{letmod}(\mu, \nu, C, c, A, e)$ $\mid \text{rec}(C, u, v, e)$
(environments)	$\rho$	$::=$	$\cdot \mid \rho.v$
(closures)	$C, f$	$::=$	$\text{clo}(M, \rho)$
(normals)	$d$	$::=$	$\downarrow^A v$

Informally, *neutral forms* are generated by variables and elimination forms stuck on other neutrals. To a first approximation, a neutral is a chain of eliminations which are stuck on a variable. On the other hand, *values*—the codomain of the evaluation function—are primarily generated by introduction forms. In particular, there are no elimination forms directly available on values and there is no uniform way to turn a value into a neutral form. Consequently,  $\beta$ -reducible terms cannot be expressed in this grammar. One can, however, lift a neutral into a value after annotating the neutral form with its type. Tersely, values are  $\beta$ -short but not necessarily  $\eta$ -long.

A defining aspect of our approach to NbE is the handling of open terms. Rather than directly evaluating under a binder, when we reach, e.g., a lambda, we suspend the computation

and store the intermediate result in a *closure*. The evaluation is resumed as soon as further information is gathered. In the case of a function, for instance, the evaluation of the body is resumed only after the function is applied. A closure is a combination of the term being evaluated and “the state of the evaluation algorithm.” The latter amounts to the environment of variables which is reified and stored in the closure alongside the term.

Normal forms have only one constructor, *reification*. Values are lifted to normals by annotating them with a type. This type annotation is used during the quotation process in Section 3.3 in order to deal with the  $\eta$ -laws.

We emphasize that while terms use De Bruijn indices, neutral forms use De Bruijn *levels* to represent variables. This small maneuver ensures that values, neutral forms, and normal forms can be silently weakened and we will capitalize on this fact throughout our algorithm. See Abel [1] for further explanation.

### 3.2 Evaluation

Evaluation is the procedure of interpreting syntax into the semantic domains, specifically values. At a high-level, this amounts to  $\beta$ -reducing all terms (recall  $\beta$ -reducible terms cannot be represented as values). The presence of variables, however, will ensure that some elimination forms will become stuck. These stuck terms are evaluated into neutrals and annotated with a type to embed them as values.

We single out a few interesting cases of the evaluation algorithm shown in Figure 2.

$$\boxed{\llbracket \_ \rrbracket : \text{Syntax} \rightarrow \text{Env} \rightarrow \text{Val}}$$

$$\begin{array}{c}
 \text{EVAL/VAR} \\
 \frac{\rho(i) = v}{\llbracket v_i \rrbracket_\rho = v}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{EVAL/PI} \\
 \frac{\llbracket A \rrbracket_\rho = A}{\llbracket (\mu \mid A) \rightarrow B \rrbracket_\rho = (\mu \mid A) \rightarrow \text{clo}(B, \rho)}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{EVAL/MODIFY} \\
 \frac{\llbracket A \rrbracket_\rho = A}{\llbracket \langle \mu \mid A \rangle \rrbracket_\rho = \langle \mu \mid A \rangle}
 \end{array}$$

$$\begin{array}{c}
 \text{EVAL/MOD} \\
 \frac{\llbracket M \rrbracket_\rho = v}{\llbracket \text{mod}_\mu(M) \rrbracket_\rho = \text{mod}_\mu(v)}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{EVAL/APP} \\
 \frac{\llbracket M \rrbracket_\rho = u \quad \llbracket N \rrbracket_\rho = v}{\llbracket M(N)_\mu \rrbracket_\rho = \text{app}(u, v)}
 \end{array}$$

$$\begin{array}{c}
 \text{EVAL/LETMOD} \\
 \frac{\llbracket M \rrbracket_\rho = v}{\llbracket \text{let}_\nu \text{ mod}_\mu(\_ ) \leftarrow M \text{ in } N : A \rrbracket_\rho = \text{letmod}_{\mu;\nu}(\text{clo}(A, \rho), \text{clo}(N, \rho), v)}
 \end{array}$$

$$(\rho.v)(0) = v \qquad (\rho.v)(i + 1) = \rho(i)$$

■ **Figure 2** Evaluation function, selected cases.

The work of evaluation is done around eliminators. Therefore, we single these cases out and define ‘helper’ functions for this portion of the algorithm. The interesting new cases are **letmod** and **app**, but generally for every syntax elimination form we define a suggestively named function that automatically beta-reduces eliminators applied to an introduction form, or returns a neutral and annotates it with its type.

$$\boxed{\text{app}(u, v) : \text{Val} \quad \text{proj}_i(v) : \text{Val} \quad \text{letmod}_{\mu;\nu}(C, c, v) : \text{Val} \quad \mathbf{J}(C, c_{\text{refl}}, p) : \text{Val}}$$

$$\begin{array}{c}
\frac{u = \lambda(\text{clo}(M, \rho)) \quad \llbracket M \rrbracket_{\rho, v} = w}{\text{app}(u, v) = w} \quad \frac{u = \uparrow^{A_0} e \quad A_0 = (\mu \mid A) \rightarrow C \quad \text{inst}(C, v) = B}{\text{app}(u, v) = \uparrow^B \text{app}[\mu](e, \downarrow^A v)} \\
\\
\frac{v = \text{mod}_\mu(v') \quad \text{inst}(c, v') = u}{\text{letmod}_{\mu; \nu}(C, c, v) = u} \quad \frac{v = \uparrow^{A_0} e \quad A_0 = \langle \mu \mid A \rangle \quad \text{inst}(C, \uparrow^{\langle \mu \mid A \rangle} e) = B}{\text{letmod}_{\mu; \nu}(C, c, v) = \uparrow^B \text{letmod}(\mu, \nu, C, c, A, e)} \\
\\
\frac{}{\text{inst}(\text{clo}(M, \rho), v) = \llbracket M \rrbracket_{\rho, v}}
\end{array}$$

As mentioned previously, we use closures to represent syntax that cannot be evaluated in the present environment. Once we have found the value to complete the environment, we *instantiate* the closure with it and continue the evaluation in the extended environment as in e.g.,  $\text{app}(\lambda(\text{clo}(M, \rho)), v)$ .

### 3.3 Quotation

Quotation is the process of turning normals into terms. We will ensure that the results of quotation are always *normal form terms*, that is,  $\beta$ -short and  $\eta$ -long terms.

To account for the fact that normal forms mention values and neutral forms, quotation is split into three mutually recursive functions. Quotation must perform  $\eta$ -expansion and is therefore type-directed. Accordingly, while we have a quotation procedure which applies to values, this portion of the algorithm can only be used for quoting types where there is no associated  $\eta$ -expansions. All three of these functions take a natural number in addition to the actual term being quoted. This number represents the next available De Bruijn level for a free variable; it is used to quote terms with binders.

We present the novel cases of quotation of normal forms—those with modalities—below:

$$\begin{array}{c}
\frac{A_0 = (\mu \mid A) \rightarrow B \quad \text{inst}(B, \uparrow^A \mathbf{v}_k) = B \quad \text{quo}(\downarrow^B \text{app}(v, \uparrow^A \mathbf{v}_k))_{k+1} = M}{\text{quo}(\downarrow^{A_0} v)_k = \lambda(M)} \\
\\
\frac{A_0 = \langle \mu \mid A \rangle \quad v = \text{mod}_\mu(w)}{\text{quo}(\downarrow^{A_0} v)_k = \text{mod}_\mu(\text{quo}(\downarrow^A w)_k)} \quad \frac{A_0 = \langle \mu \mid A \rangle \quad v = \uparrow^B e}{\text{quo}(\downarrow^{A_0} v)_k = \text{quo}(e)_k} \\
\\
\frac{A_0 = \uparrow^A e \quad v = \uparrow^{A'} e}{\text{quo}(\downarrow^{A_0} v)_k = \text{quo}(e)_k}
\end{array}$$

We draw attention to one aspect of the first rule. This rule quotes a function, so consider the case where  $v = \lambda(\text{clo}(M, \rho))$ . We create a fresh variable  $\uparrow^A \mathbf{v}_k$  and make the semantic application  $\text{app}(\lambda(\text{clo}(M, \rho)), \uparrow^A \mathbf{v}_k)$ . This last step is only sensible because values are closed under silent weakening; the environment  $\rho$  would otherwise need to be weakened over the freshly created variable  $\mathbf{v}_k$ .

Finally, we record the novel cases of quotation for neutral forms:

$$\begin{array}{c}
\frac{}{\text{quo}(\text{app}[\mu](e, d))_k = \text{quo}(e)_k(\text{quo}(d)_k)_\mu} \\
\\
\frac{\text{inst}(C, \text{mod}_\mu(\uparrow^A \mathbf{v}_k)) = B \quad \text{inst}(c, \uparrow^A \mathbf{v}_k) = v}{\text{quo}(\text{letmod}(\mu, \nu, C, c, A, e))_k = \text{let}_\nu \text{ mod}_\mu(\_ ) \leftarrow \text{quo}(e)_k \text{ in } \text{quo}(\downarrow^B v)_{k+1}}
\end{array}$$

### 3.4 The NbE function

Having defined both evaluation and quotation, we are almost in a position to define the complete normalization algorithm. The only missing step is the construction of the *initial environment* from a context. This portion of the algorithm takes a context  $\Gamma$  and produces an environment consisting of the variables bound in  $\Gamma$ . We then use this environment to kick off the evaluation of terms in context  $\Gamma$ :

$$\mathbf{reflect}(\mathbf{1}) = \cdot \quad \mathbf{reflect}(\Gamma.(\mu \mid A)) = \mathbf{reflect}(\Gamma).\uparrow^{\llbracket A \rrbracket_{\mathbf{reflect}(\Gamma)}} \mathbf{v}_{|\Gamma|}$$

Finally, the complete normalization algorithm evaluates a term  $\Gamma \vdash M : A @ m$  in the initial environment specified by  $\Gamma$  and quotes it back:

$$\mathbf{norm}_{\Gamma,A}(M) = \mathbf{quo}(\downarrow^{\llbracket A \rrbracket_{\mathbf{reflect}(\Gamma)}} \llbracket M \rrbracket_{\mathbf{reflect}(\Gamma)})_{|\Gamma|}$$

## 4 Implementing a Mode Theory

Thus far we have been somewhat vague about which mode theory we were instantiating MTT with. The normalization algorithm given in Section 3, for instance, did not need to manipulate or compare modalities and so this point was easy to gloss over. The type-checker, on the other hand, must manipulate and scrutinize modalities and its definition requires a precise specification of a mode theory. Accordingly, we now present a representation of mode theories and operations upon them suitable for implementing a type-checker.

Concretely, our presentation closely follows the actual representation of mode theories used in `mitten`, our implementation of MTT. In `mitten`, all information specific to a mode theory is confined to a single OCaml module on which the type-checker depends. In particular, to configure `mitten` to work with a new mode theory, it is only necessary to implement that single module. Our signature for mode theories is comprised of three essential parts (summarized in Figure 3):

1. Two abstract types; one for modes and one for modalities.
2. Various operations to compose modalities, extract the domain or codomain mode from a modality, or construct the identity modality.
3. Three operations to compare modes for equality and modalities for (in)equality.

It is these last two operations which are particularly crucial. Recall that not all mode theories admit decidable (in)equality and without it, type-checking MTT is undecidable. Accordingly, any implementation of MTT will require the user to supply a decision procedure for the mode theory. Our implementation shows that this information is both necessary and essentially sufficient. We note that the decision procedures for mode theories are completely separate from the terms and types of MTT and no knowledge of e.g., normalization-by-evaluation is required for their implementation.<sup>2</sup>

► **Remark 15.** The reader might wonder why `idm` is not parametrized over `mode`. This is because `idm` internally is a placeholder for some identity modality, whose mode is elaborated. This alleviates practitioners of some tedious bookkeeping obligations in their proofs. This approach necessitates that the boundary projections `dom_mod` and `cod_mod` take an additional argument of type `mode`, which is returned on input `idm`. Essentially, we assume that always one part of the boundary is known so `dom_mod` gets a modality and its codomain as argument whereas `cod_mod` a gets modality and its domain as argument.

<sup>2</sup> See the following for examples: [https://github.com/logsem/mitten\\_preorder/blob/main/src/lib/](https://github.com/logsem/mitten_preorder/blob/main/src/lib/)

```

type mode
val eq_mode : mode → mode → bool

type m
val idm : m
val compm : m → m → m
val dom_mod : m → mode → mode
val cod_mod : m → mode → mode
val (=) : m → m → bool
val (≤) : m → m → bool

```

■ **Figure 3** A fragment of the signature for mode theories used in `mitten`.

## 5 Semantic Type-Checking Algorithm

Having defined the normalization algorithm, we now define the type-checking algorithm for MTT. As mentioned in Sections 1 and 2, the algorithm is a variant of Coquand’s semantic type checking algorithm for dependent type theory [9]. Accordingly, the algorithm breaks into two distinct phases: checking and synthesis. The checking portion of the algorithm accepts a context  $\Gamma$ , a term  $M$ , and a type  $A$  and checks that  $M$  has type  $A$  in context  $\Gamma$ . The synthesis phase accepts only the context and term, and synthesizes the type of the term in this context.

This simple picture is slightly complicated in the case of MTT, where various side conditions must be managed. For instance, we must ensure that the modalities a user writes in modal types are well-formed and that the term and type exist at the same mode as the context. These same considerations also require us to form a more intricate notion of a *semantic context* specifically for the type-checking algorithm.

We discuss the definition of semantic contexts in Section 5.1 and present a representative fragment of the type-checking algorithm itself in Section 5.2.

### 5.1 Semantic Contexts

In Section 2.2, we explained the intuitions behind MTT while working informally with the collection of variables in scope. Prior to discussing the type checker, we must describe the precise notion of context to organize these variables. Two factors complicate this otherwise standard structures: the modal annotations and restrictions and the need to evaluate terms during type-checking.

To a first approximation, contexts are still lists of variables with types but with additional bells and whistles added in order to support these two requirements. In order to record the necessary modal information, each variable is annotated by a modality. Deviating from Section 2.2, we add a new context operation  $\Xi.\{\mu\}$  to ‘lazily’ restrict all entries in a context  $\Xi$  by  $\mu$  rather than storing this information on each variable separately.

For the second requirement, recall that type-checking must repeatedly test when two types are equal for the *conversion rule*. Accordingly, the context must store enough information to support this conversion test. We follow Coquand [9] and represent each type in the context by the corresponding *value* (in the sense of Section 3) and pair each variable with a corresponding value. This value may just be  $\uparrow^A v_i$ , but it may also store the term associated definition. By storing information in this form, we can easily project out a *semantic environment* of a context and use that to evaluate a term and check for convertibility during type checking.

The grammar of *semantic contexts* is presented below:

(semantic contexts)  $\Xi ::= \cdot \mid \Xi.(v :_{\mu} A @ m) \mid \Xi.\{\mu\}$

We now define two functions: The partial *lookup function*, which displays the type with its annotation and restriction as well as the mode it lives at, and the *stripping* function, which returns an environment by projecting out only the value components of the semantic context. The lookup function is undefined whenever a De Bruijn index is larger than the length of the context.

$$\begin{aligned} (\Xi.(v :_{\mu} A @ m))(0) &= (\mu|A)_m, \{\text{id}\} \\ (\Xi.(w :_{\xi} B @ o))(i+1) &= (\mu|A)_m, \{\nu'\} && \text{where } (\mu|A)_m, \nu = \Xi(i) \\ (\Xi.\{\nu'\})(i) &= (\mu|A)_m, \{\nu \circ \nu'\} && \text{where } (\mu|A)_m, \nu = \Xi(i) \\ |\cdot| &= \cdot \\ |\Xi.(v :_{\mu} A @ m)| &= |\Xi|.v \\ |\Xi.\{\mu\}| &= |\Xi| \end{aligned}$$

► **Notation 16.** *If we extend a semantic context with a type where the value is a fresh variable, we hide it to make the expression more readable.*

$$\Xi.(\mu|A) \triangleq \Xi.(\uparrow^A \mathbf{v}_0 :_{\mu} A @ m)$$

*If the modality  $\mu$  is furthermore the identity modality, we omit it and write*

$$\Xi.A \triangleq \Xi.(\uparrow^A \mathbf{v}_0 :_{\text{id}} A @ m)$$

## 5.2 Checking and Synthesis

We now come to the type-checking algorithm which is split into a pair of judgments:  $\Xi \vdash M \Leftarrow A @ m$  and  $\Xi \vdash R \Rightarrow A @ m$ . The first,  $\Xi \vdash M \Leftarrow A @ m$ , handles type checking which tests if  $M$  has type  $A$  in  $\Xi$ . The second,  $\Xi \vdash R \Rightarrow A @ m$ , implements type synthesis and accordingly takes only the semantic context  $\Xi$  and term  $M$  and returns type  $A$  of  $M$  in context  $\Xi$  if one can be inferred.

We present a few representative rules for these judgments (and explain them below). To ensure that terms and types are well-formed, we utilize the functions exposed by the signature presented in Section 4. In particular,  $n \stackrel{?}{=} m$  checks whether two modes are equal and  $\mu \leq \nu$  is the modality ordering relation. Furthermore, with  $\mu.\text{dom}$  and  $\mu.\text{cod}$  we denote the respective domain and codomain of a modality—denoted `dom_mod` and `cod_mod` respectively in Section 4. For readability, we leave the second argument of  $\mu.\text{dom}$  and  $\mu.\text{cod}$  implicit.

$$\begin{array}{c} \text{PI} \\ \frac{\Xi.\{\mu\} \vdash A \Leftarrow \mathbf{U} @ \mu.\text{dom} \quad \Xi.(\mu|A) \vdash B \Leftarrow \mathbf{U} @ m \quad \mu.\text{cod} \stackrel{?}{=} m}{\Xi \vdash (\mu | A) \rightarrow B \Leftarrow \mathbf{U} @ m} \\ \\ \text{MOD-FORM} \qquad \qquad \qquad \text{MOD-INTRO} \\ \frac{\Xi.\{\mu\} \vdash A \Leftarrow \mathbf{U} @ \mu.\text{dom} \quad \mu.\text{cod} \stackrel{?}{=} m}{\Xi \vdash \langle \mu | A \rangle \Leftarrow \mathbf{U} @ m} \qquad \frac{\Xi.\{\mu\} \vdash M \Leftarrow A @ \mu.\text{dom} \quad \mu.\text{cod} \stackrel{?}{=} m}{\Xi \vdash \text{mod}_{\mu}(M) \Leftarrow \langle \mu | A \rangle @ m} \\ \\ \text{CONV} \qquad \qquad \qquad \text{VAR} \\ \frac{\Xi \vdash R \Rightarrow B @ m \quad A \equiv_{|\Xi|} B}{\Xi \vdash R \Leftarrow A @ m} \qquad \frac{\Xi(k) = (\mu|A)_m, \nu \quad \mu \leq \nu \quad m \stackrel{?}{=} n}{\Xi \vdash \mathbf{v}_k \Rightarrow A @ n} \end{array}$$

$$\begin{array}{c}
\text{MOD-ELIM} \\
\frac{\nu.\text{cod} \stackrel{?}{=} m \quad \Xi.\{\nu\} \vdash R \Rightarrow \langle \mu \mid A \rangle @ \nu.\text{dom} \quad \Xi.(\nu, \langle \mu \mid A \rangle) \vdash C \Leftarrow \mathbf{U} @ m \\
\quad \Xi.(\nu \circ \mu, A) \vdash N \Leftarrow \llbracket C \rrbracket_{|\Xi|. \text{mod}_\mu(\uparrow^A \mathbf{v}_k)} @ m}{\Xi \vdash \text{let}_\nu \text{ mod}_\mu(\_) \leftarrow R \text{ in } N \text{ over } C \Rightarrow \llbracket C \rrbracket_{|\Xi|. \llbracket M \rrbracket_{|\Xi|}} @ m}
\end{array}$$

We first consider the formation rule for dependent products. First we verify that indeed  $\mu.\text{cod} \stackrel{?}{=} m$  to ensure that the modality  $\mu$  can be used at this mode. Recall that  $\Pi$ -types in MTT go from a  $\mu$ -restricted type  $A$  to a non restricted type  $B$ . Accordingly, we check that  $A$  is a type in the  $\mu$ -restricted semantic context  $\Xi.\{\mu\}$  and that  $B$  is well-formed in the context  $\Xi.(\mu \mid A)$ . Note that when checking  $A$  we change the mode to  $\mu.\text{dom}$ .

Since the modal formation and introduction rules follow a similar pattern we will only look at the modal introduction rule. To validate that  $\text{mod}_\mu(M)$  has type  $\langle \mu \mid A \rangle$  at mode  $m$  we first verify that  $\mu.\text{cod} \stackrel{?}{=} m$ . Next we check that  $M$  has type type  $A$  in the  $\mu$ -restricted environment  $\Xi.\{\mu\}$  at mode  $\mu.\text{dom}$ .

Next, we discuss the conversion rule. When considering a synthesizable term  $R$ , the type-checking algorithm proceeds somewhat differently. We first synthesize the type of  $R$  and then compare the result to the type we were given to check  $R$  against. It is this comparison which uses the normalization algorithm of Section 3 to compute the normal forms of  $A$  and  $B$  and decides afterwards the equality of the normalized expressions.

To synthesize a variable  $\mathbf{v}_k$  in a semantic context  $\Xi$  at mode  $m$  we first compute the type of the variable together with its annotation and restriction  $(\mu \mid A)_m, \{\nu\}$ , using the lookup function defined in Section 5.1. Before we return  $A$  as the type of  $\mathbf{v}_k$ , we must also perform an additional check to ensure that  $\mu \leq \nu$  so that this occurrence of the variable is valid.

Finally, we consider the modal elimination case. Recall from Section 2.2 that the modal elimination principle allows us to ‘pattern-match’ on a term a  $M : \langle \mu \mid A \rangle$  in a  $\nu$ -restricted context and replace it with a variable  $x :_{\nu \circ \mu} A$ . To synthesize  $\text{let}_\nu \text{ mod}_\mu(\_) \leftarrow R \text{ in } N \text{ over } C$ , we take advantage of the fact that the user provides the motive  $C$  already; if this term is well-typed, its type must be  $\llbracket C \rrbracket_{|\Xi|. \llbracket M \rrbracket_{|\Xi|}}$ .

There are, however, several checks to perform to ensure that the term is actually well-typed. First, we check that  $\nu.\text{cod} \stackrel{?}{=} m$ . Next, we synthesize the type of  $R$  in the  $\nu$ -restricted context and check that the result is of the form  $\langle \mu \mid A \rangle$ . Having computed  $\langle \mu \mid A \rangle$ , we then check that both  $C$  and  $N$  are well-formed. The motive  $C$  must be a type in the extended context  $\Xi.(\nu, \langle \mu \mid A \rangle)$  while  $N$  must have type  $\llbracket C \rrbracket_{|\Xi|. \llbracket M \rrbracket_{|\Xi|}}$  in context  $\Xi.(\nu, \langle \mu \mid A \rangle)$ .

A complete implementation of the algorithm can be found at [https://github.com/logsem/mitten\\_preorder/blob/main/src/lib/check.ml](https://github.com/logsem/mitten_preorder/blob/main/src/lib/check.ml).

## 6 Case study: guarded recursion in mitten

We now discuss an extended example using `mitten` with a particular choice of mode theory. By instantiating `mitten` appropriately, we convert it into a proof assistant for *guarded recursion* and use it to reason about classical examples from the theory.

### 6.1 Guarded recursion

Guarded recursion provides a discipline for managing recursive definitions within type theory without compromising soundness. In particular, guarded type theory extends type theory with a handful of modalities ( $\blacktriangleright$ ,  $\Gamma$  and  $\Delta$ ) along with a modified version of the fixed-point combinator:

$$\text{loeb} : (\blacktriangleright A \rightarrow A) \rightarrow A$$

By placing the recursive call under a  $\blacktriangleright$ , this weakened fixed-point combinator does not result in inconsistencies. Together with the other modalities, moreover, it can be used to define and reason about coinductive types and gives rise to a *synthetic* form of domain theory.

Following [6], we are interested in using `mitten` as a tool to reason about a particular *model* of guarded recursion:  $\mathbf{PSh}(\omega)$ . In fact, using MTT's capacity to reason about multiple categories at once, we will work with a slightly richer model which includes both  $\mathbf{PSh}(\omega)$  and  $\mathbf{Set}$ . In this model, the aforementioned modalities are all interpreted by right adjoints:

$$\begin{array}{lll} \Gamma : \mathbf{PSh}(\omega) \rightarrow \mathbf{Set} & \Delta : \mathbf{Set} \rightarrow \mathbf{PSh}(\omega) & \blacktriangleright : \mathbf{PSh}(\omega) \rightarrow \mathbf{PSh}(\omega) \\ \Gamma(X) = [\mathbf{1}, X] & \Delta(S) = \lambda \_ . S & \blacktriangleright(X)(0) = \{\star\} \quad \blacktriangleright(X)(n+1) = X(n) \end{array}$$

In particular, the composite of  $\Gamma$  and  $\Delta$  is the global sections comonad  $\square$ . The fixed-point operator `loeb` in  $\mathbf{PSh}(\omega)$  is definable using induction over  $\omega$ .

Gratzer et. al [12] have shown that MTT with a mode theory axiomatizing these three modalities is modeled by these two categories and therefore provides a suitable basis for guarded recursion. We recall their mode theory in Figure 4.



■ **Figure 4**  $\mathcal{G}$ : a mode theory for guarded recursion

The equalities represented in Figure 4 together with the equational theory of MTT ensure that  $\square = \delta \circ \gamma$  is an idempotent comonad and that the following equivalence is definable:

$$\langle \square \mid \langle \ell \mid A \rangle \rangle \simeq \langle \square \mid A \rangle.$$

In order to actually reason about guarded definitions, however, we still must add Löb induction to the system. Adding Löb induction primitively raises substantial issues [11], so we opt to axiomatize it along with a (propositional) equation specifying its unfolding principle:

$$\text{loeb} : ((\ell \mid A) \rightarrow A) \rightarrow A @ t \quad \text{unfold} : (f : (\ell \mid A) \rightarrow A) \rightarrow \text{Id}_A(\text{loeb } f, f(\text{loeb } f)) @ t$$

As to be expected, these new constants disrupt canonicity but crucially cause no issues for type checking. We now discuss how to instantiate `mitten` with this particular mode theory.

## 6.2 Implementation

In order to use `mitten` to reason about guarded MTT, we must construct an implementation of the mode theory module corresponding to Figure 4 and extend `mitten` with constants for Löb induction. The latter point is routine; `mitten` supports adding axioms to a development. We therefore focus on the first step: the implementation of the mode theory.

The main challenge when implementing Figure 4 is to show that the relation  $\leq$  is decidable. We have done so by using a (simple) form of normalization-by-evaluation to reduce modalities in this mode theory to normal forms which can be directly compared.

► **Remark 17.** We leave the modes during the evaluation implicit and assume, without loss of generality, that we are only considering well-formed modalities.<sup>3</sup>

By studying the category generated by Figure 4, it becomes clear that  $\mathcal{G}$  is far from a free mode theory. In fact, many possible compositions trivialize; in a chain of composable modalities we can freely remove any  $\gamma \circ \delta$  as well as any  $\ell$  to the right of a  $\gamma$ . Accordingly, there are only four kinds of expressions remaining which thus constitute *normal modalities*:

$$(Normal\ modalities) \quad \mu, \nu ::= \ell^k \mid \ell^k \circ \delta \mid \ell^k \circ \delta \circ \gamma \mid \gamma \mid id_s$$

Note that  $k = 0$  is allowed and thus in particular  $\delta \circ \gamma = \ell^0 \circ \delta \circ \gamma$  as well as  $id = \ell^0$ . There is an evident map  $i$  sending a normal form  $\mu$  to a modality in  $\mathcal{G}$ . We now construct an inverse to this map:

$$\begin{array}{ll} \underline{\text{eval}}(id_t) = \ell^0 & \underline{\text{comp}}(\ell, \ell^k) = \ell^{k+1} \\ \underline{\text{eval}}(id_s) = id_s & \underline{\text{comp}}(\ell, \ell^k \circ \delta) = \ell^{k+1} \circ \delta \\ \underline{\text{eval}}(\ell \circ \nu) = \underline{\text{comp}}(\ell, \underline{\text{eval}}(\nu)) & \underline{\text{comp}}(\ell, \ell^k \circ \delta \circ \gamma) = \ell^{k+1} \circ \delta \circ \gamma \\ \underline{\text{eval}}(\gamma \circ \nu) = \underline{\text{comp}}(\gamma, \underline{\text{eval}}(\nu)) & \underline{\text{comp}}(\gamma, \ell^k) = \gamma \\ \underline{\text{eval}}(\delta \circ \nu) = \underline{\text{comp}}(\delta, \underline{\text{eval}}(\nu)) & \underline{\text{comp}}(\gamma, \ell^k \circ \delta \circ \gamma) = \gamma \\ & \underline{\text{comp}}(\delta, id_s) = \ell^0 \circ \delta \\ & \underline{\text{comp}}(\delta, \gamma) = \ell^0 \circ \delta \circ \gamma \end{array}$$

► **Theorem 18.** For any modality  $\mu$  we have that  $\mu = i(\underline{\text{eval}}(\mu))$ .

Next, we define a (decidable) partial ordering on normal modalities:

$$\frac{m \leq n}{\ell^m \sqsubseteq \ell^n} \quad \frac{}{\gamma \sqsubseteq \gamma} \quad \frac{m \leq n}{\ell^m \circ \delta \circ \gamma \sqsubseteq \ell^n \circ \delta \circ \gamma} \quad \frac{m \leq n}{\ell^m \circ \delta \sqsubseteq \ell^n \circ \delta} \quad \frac{}{id_s \sqsubseteq id_s}$$

► **Theorem 19.** For any normal modalities  $\mu$  and  $\nu$  we have  $\mu \sqsubseteq \nu$  if and only if  $i(\mu) \leq i(\nu)$ .

► **Corollary 20.** Equality of modes and inequality of modalities are both decidable.

### 6.3 Streams in guarded mitten

We now illustrate the use of this instantiation of *mitten* by defining the types of guarded and coinductive streams and constructing various examples.

► **Remark 21.** In the following we deviate from our surface syntax to enhance readability of the derivations. Thus, we leave many arguments implicit and alter certain notations. In particular, propositional identities are denoted by  $a \equiv b$  instead of  $\text{Id}_A(a, b)$  and implicit arguments are omitted. We furthermore hide the type family  $C$  of the modal elimination rule in the following constructions.

We begin with the type of *guarded streams*.

$$\begin{array}{ll} \text{gstream\_fun} : \mathbb{U} \rightarrow (\ell \mid \mathbb{U}) \rightarrow \mathbb{U} @ t & \text{gstream} : \mathbb{U} \rightarrow \mathbb{U} @ t \\ \text{gstream\_fun } A \ X = A \times \langle \ell \mid X \rangle & \text{gstream } A = \text{loeb}(\text{gstream\_fun } A) \end{array}$$

► **Notation 22.** We will make use of several standard functions for intensional identity types such as the functions  $\text{transport} : A \equiv B \rightarrow A \rightarrow B$  and  $-^{-1} : a \equiv b \rightarrow b \equiv a$ .

<sup>3</sup> This assumption is justified since *mitten* checks all modalities prior to normalization and type-checking.

## XX:18 mitten: a flexible multimodal proof assistant

Recall that we have added Löb induction only with a *propositional* unfolding rule. Accordingly, we must use `transport` along this equality to obtain the folding and unfolding operations for `gstream`:

$$\begin{aligned} \text{gfold} &: (A : \mathbb{U}) \rightarrow A \times \langle \ell \mid \text{gstream } A \rangle \rightarrow \text{gstream } A @ t \\ \text{gfold } A &= \text{transport } (\text{unfold}(\text{gstream\_fun } A))^{-1} \end{aligned}$$

$$\begin{aligned} \text{gunfold} &: (A : \mathbb{U}) \rightarrow \text{gstream } A \rightarrow A \times \langle \ell \mid \text{gstream } A \rangle @ t \\ \text{gunfold } A &= \text{transport } (\text{unfold}(\text{gstream\_fun } A)) \end{aligned}$$

We are able to deduce the following equalities by using the fact that `transport  $p$`  is inverse to `transport  $p^{-1}$` :

$$\begin{aligned} \text{fold\_unfold} &: (s : \text{gstream } A) \rightarrow \text{gfold } A (\text{gunfold } A s) \equiv s @ t \\ \text{unfold\_fold} &: (s : A \times \langle \ell \mid \text{gstream } A \rangle) \rightarrow \text{gunfold } A (\text{gfold } A s) \equiv s @ t \end{aligned}$$

Using this we can define the familiar operations on guarded streams and prove their expected equations.

$$\begin{array}{ll} \text{ghead} : \text{gstream } A \rightarrow A & \text{gtail} : \text{gstream } A \rightarrow \langle \ell \mid \text{gstream } A \rangle \\ \_ : \text{gtail}(\text{gcons } a s) \equiv s & \text{gcons} : A \rightarrow \langle \ell \mid \text{gstream } A \rangle \rightarrow \text{gstream } A \\ \_ : \text{ghead}(\text{gcons } a s) \equiv a & \_ : \text{gcons } (\text{gheads } s) (\text{gtails } s) \equiv s \end{array}$$

With Löb induction, these definitions and equalities allow us to construct and work with *guarded* streams, which differ from coinductive streams in several important ways. For instance, the tail operation on guarded streams produces a guarded stream under a later which prevents us from writing an operation dropping every element of a guarded stream.

By making use of the other modalities of Figure 4, we are able to define the type of *coinductive* streams. To do so, we will use the following operations:

$$\text{comp}_{\gamma,\delta} : \langle \gamma \mid \langle \delta \mid A \rangle \rangle \rightarrow A \quad \text{comp}_{\gamma,\ell} : \langle \gamma \mid \langle \ell \mid A \rangle \rangle \rightarrow \langle \gamma \mid A \rangle$$

Both of these are instances of the general composition principle for modalities available in MTT. We now define streams as follows:

$$\begin{aligned} \text{stream} &: \mathbb{U} \rightarrow \mathbb{U} @ s \\ \text{stream } A &= \langle \gamma \mid \text{gstream } \langle \delta \mid A \rangle \rangle \end{aligned}$$

$$\begin{array}{ll} \text{head} : \text{stream } A \rightarrow A & \text{tail} : \text{stream } A \rightarrow \text{stream } A \\ \text{head } s = & \text{tail } s = \\ \text{let}_{\text{id}} \text{mod}_{\gamma}(g) = s \text{ in} & \text{let}_{\text{id}} \text{mod}_{\gamma}(g) = s \text{ in} \\ \text{comp}_{\gamma,\delta}(\text{mod}_{\gamma}(\text{ghead } g)) & \text{comp}_{\gamma,\ell}(\text{mod}_{\gamma}(\text{gtail } g)) \end{array}$$

We emphasize that the type of coinductive streams lives at mode  $s$ , the mode modeled by sets. Intuitively, by taking the global sections of a guarded stream we obtain the normal coinductive stream [8]. Indeed, using guarded recursion in mode  $t$ , we are able to equip this type with a coiteration principle:

$$\begin{aligned} \text{go} &: (\delta \mid A : \mathbb{U})(\delta \mid S : \mathbb{U})(\delta \mid S \rightarrow A \times S) \rightarrow (\delta \mid S) \rightarrow \text{gstream } \langle \delta \mid A \rangle @ t \\ \text{go } A S f &= \text{loeb } \lambda g s. \text{gcons}(\text{mod}_{\delta}(\pi_1(f s)), \text{mod}_{\ell}(g(\pi_2(f s)))) \end{aligned}$$

$$\text{coiter} : (A : \mathbb{U})(S : \mathbb{U}) \rightarrow (S \rightarrow A \times S) \rightarrow S \rightarrow \text{stream } A @ s$$

$$\text{coiter } A S f s = \text{mod}_g(\text{go } A S f s)$$

Informally, this coiteration scheme induces a map from any  $(A \times -)$ -coalgebra to  $\text{stream } A$ .

It is natural to wonder whether  $\text{stream } A$  is the *final coalgebra* for  $(A \times -)$ . In the presence of equality reflection, this was established by Gratzer et al. [12]. To replay this proof in `mitten`, we would require two ingredients not presently available: function extensionality and modal extensionality. The first is unsurprising, so we focus on the second. Modalities do not necessarily preserve identity types and therefore in general we cannot have a function:

$$(\ell \mid \text{Id}_A(a, b)) \rightarrow \text{Id}_{\langle \ell | A \rangle}(\text{mod}_\ell(a), \text{mod}_\ell(b))$$

Such a map is crucial to establish arguments of equality by Löb induction like the finality of  $\text{stream } A$ . Having said this, we emphasize that without disrupting normalization we can extend MTT with a crisp induction principle which enables us to construct such a map and prove it to be an equivalence [10]. In the presence of this additional structure—or a postulate to the same effect—we conjecture that we can prove  $\text{stream } A$  to be a final coalgebra purely within `mitten`.

We conclude with a simple example of the coiteration: the stream of all natural numbers.

```
nats : stream Nat
nats = coiter (λn. (n, succ(n))) 0
```

## 7 Related Work

Modal proof assistants have seen a great deal of attention in the last several years. We compare our work on `mitten` to several of the most closely related lines of research.

### 7.1 Normalization for MTT

In [10], Gratzer proves that MTT enjoys a normalization algorithm. While his proof avoids a number of technicalities by adopting a synthetic approach to normalization, this obstructs extracting an actual algorithm for use in implementation. We have taken this next step and, inspired by the synthetic proof of normalization, obtained an actual algorithm suitable for implementation in the particular case of preordered mode theories. Furthermore, while Gratzer works relative to the assumption that the ambient mode theory is decidable, we have isolated the precise requirements necessary on the mode theory and shown that they are sufficiently flexible to accommodate common mode theories.

### 7.2 Guarded recursion in Agda

In Section 6 we discussed an instantiation of `mitten` for guarded recursion. For this specific case, an experimental Agda extension is available [21]. This extension implements a version of clocked cubical type theory [4]. This variant of guarded type theory offers finer-grained guarded programming by exposing multiple independent later modalities; these can be used to interleave guarded types without issue. Furthermore, clocked cubical type theory capitalizes on certain primitives of cubical type theory to expose some definitional equalities around Löb induction. Guarded cubical Agda builds upon Agda’s existing facilities for interactive proof developments and the system has been used for non-trivial developments [19, 22].

As a consequence of this more intricate theory, however, the metatheory of guarded cubical Agda is far less developed than the theory of `mitten`. Moreover, the infrastructure of guarded cubical Agda is (necessarily) specialized to just one modal situation. While `mitten`

is a more primitive system than guarded cubical Agda, it is therefore far more flexible and offers a theoretical framework for many modal systems rather than being specialized to one.

### 7.3 Sikkel

Recently, Ceulemans et al. [7] have explored an alternative strategy for implementing MTT in Sikkel. Rather than constructing a custom proof assistant like `mitten`, they have provided a DSL for a simply-typed version of MTT within Agda. Within this DSL, one may construct terms in MTT which then compile to elements of an appropriate denotational semantics expressed within Agda. A major advantage of such an approach is the low startup cost: the full resources of the Agda proof assistant are available when working within Sikkel. By embedding within Agda, however, Sikkel’s interface is less convenient and it is currently restricted only to simple types. Accordingly, we believe that a proof assistant designed for MTT from its inception offers a more promising route for serious modal programming.

### 7.4 Menkar

Menkar [20] is an earlier attempt at a proof assistant for multimodal programming developed by Nuyts. It predates—and in fact partially inspires—MTT, but contains both theoretical and practical deficiencies which led to its development being suspended in 2019. Inspired by the advances in proof theory for multimodal type theory obtained since Menkar’s development, both `mitten` and Sikkel are early attempts to develop a theoretically sound replacement for Menkar. While not as fully-featured as Menkar, `mitten` in particular is an attempt to develop a principled modal proof assistant.

## 8 Conclusions and future work

We contribute `mitten`, a flexible proof assistant which can be specialized to a wide range of modal type theories. We have designed normalization and type-checking algorithms for `mitten` based on recent advances in the metatheory of MTT [10]. Finally, we have argued for `mitten`’s utility by instantiating it to a mode theory suitable for guarded recursion and constructing various classical examples of guarded programs.

Thus far, `mitten` is restricted to working with preordered mode theories. While this constitutes a large and important class of examples, it would be desirable to implement full MTT and allow for arbitrary 2-categories as mode theories. Such an extension, however, would require a more refined normalization algorithm.

In particular, in our algorithm we have taken advantage of the absence of distinct 2-cells to avoid annotating variables with modal coercions. This, in turn, preserves a crucial invariant of NbE: it is never necessary to explicitly substitute within a value. Indeed, in our style of NbE such substitutions are not even possible; our representation of closures essentially precludes them. We hope to generalize our approach to cover full MTT by incorporating some techniques recently used by Hu and Pientka [15] in a normalization algorithm for a particular modal type theory. Essentially, they enable a small amount of substitution to occur during the normalization algorithm; by carefully structuring the necessary modal substitutions they are able to adapt the standard normalization-by-evaluation to their setting. We hope to do the same in `mitten` by generalizing their approach to support multiple interacting modalities.

---

**References**

---

- 1 Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity*. Habilitation, 2013.
- 2 Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017. URL: <http://www.itu.dk/people/mogel/papers/lics2017.pdf>, doi: 10.1109/LICS.2017.8005097.
- 3 Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Simply ratt: A fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.*, 3:109:1–109:27, 2019. doi:10.1145/3341713.
- 4 Magnus Baunsgaard Kristensen, Rasmus Ejlers Mogelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '22*, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3531130.3533359.
- 5 Lars Birkedal, Ranald Clouston, Bassel Manna, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science*, 30(2):118–138, 2020. arXiv:1804.05236, doi:10.1017/S0960129519000197.
- 6 Lars Birkedal, Rasmus Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:1)2012.
- 7 Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. Sikkel: Multimode simple type theory as an agda library. In *Electronic Proceedings in Theoretical Computer Science*, volume 360, pages 93–112. Munich, Germany, Open Publishing Association, 06 2022. doi:10.4204/EPTCS.360.5.
- 8 Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 407–421. Springer Berlin Heidelberg, 2015.
- 9 Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996. doi:[https://doi.org/10.1016/0167-6423\(95\)00021-6](https://doi.org/10.1016/0167-6423(95)00021-6).
- 10 Daniel Gratzer. Normalization for multimodal type theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '22*, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3531130.3532398.
- 11 Daniel Gratzer and Lars Birkedal. A Stratified Approach to Löb Induction. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:22, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/opus/volltexte/2022/16304>, doi:10.4230/LIPIcs.FSCD.2022.23.
- 12 Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. URL: <https://lmcs.episciences.org/7713>, doi:10.46298/lmcs-17(3:11)2021.
- 13 Daniel Gratzer, G.A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '20*. ACM, 2020. doi:10.1145/3373718.3394736.
- 14 Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a Modal Dependent Type Theory. *Proc. ACM Program. Lang.*, 3, 2019. doi:10.1145/3341711.
- 15 Jason Z. S. Hu and Brigitte Pientka. An investigation of kripke-style modal type theories, 2022. URL: <https://arxiv.org/abs/2206.07823>, doi:10.48550/ARXIV.2206.07823.
- 16 Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal Universes in Models of Homotopy Type Theory. In H. Kirchner, editor, *3rd International Conference on Formal*

- Structures for Computation and Deduction (FSCD 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 22:1–22:17. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. [arXiv:1801.07664](https://arxiv.org/abs/1801.07664), doi:10.4230/LIPIcs.FSCD.2018.22.
- 17 Daniel R. Licata and Michael Shulman. Adjoint Logic with a 2-Category of Modes. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, pages 219–235. Springer International Publishing, 2016. doi:10.1007/978-3-319-27683-0\_16.
  - 18 Daniel R. Licata, Michael Shulman, and Mitchell Riley. A Fibrational Framework for Substructural and Modal Logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.FSCD.2017.25.
  - 19 Rasmus Ejlers Møgelberg and Niccolò Veltri. Bisimulation as path type for guarded recursive types. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi:10.1145/3290317.
  - 20 Andreas Nuyts. Menkar. <https://github.com/anuyts/menkar>, 2019.
  - 21 The Agda Team. Agda, 2022. URL: <https://agda.readthedocs.io/en/latest/language/guarded-cubical.html>.
  - 22 Niccolò Veltri and Andrea Vezzosi. Formalizing  $\pi$ -calculus in guarded cubical agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 270–283, 2020.
  - 23 Vezzosi, Andrea. *agda-flat*, 2018. URL: <https://github.com/agda/agda/tree/flat>.