BiSikkel: A Multimode Logical Framework in Agda

JORIS CEULEMANS, KU Leuven, Belgium
ANDREAS NUYTS, KU Leuven, Belgium
DOMINIQUE DEVRIESE, KU Leuven, Belgium

Embedding Multimode Type Theory (MTT) as a library enables the usage of additional reasoning principles in off-the-shelf proof assistants without risking soundness or compatibility. Moreover, by interpreting embedded MTT terms in an internally constructed model of MTT, we can extract programs and proofs to the metalanguage and obtain interoperability between the embedded language and the metalanguage. The existing Sikkel library for Agda achieves this for Multimode *Simple* Type Theory (MSTT) with an internal presheaf model of *dependent* MTT. In this work, we add, on top of the simply-typed layer, a logical framework in which users can write multimode proofs about multimode Sikkel programs, still in an off-the-shelf proof assistant. To this end, we carve out of MTT a new multimode logical framework μLF over MSTT and implement it on top of Sikkel, interpreting both in the existing internal model. In the process, we further extend and improve the original codebase for each of the three layers (syntax, semantics and extraction) of Sikkel. We demonstrate the use of μLF by proving some properties about functions manipulating guarded streams and by implementing an example involving parametricity predicates.

CCS Concepts: • **Theory of computation → Type theory**; *Programming logic*; *Modal and temporal logics*; Categorical semantics.

Additional Key Words and Phrases: multimode type theory (MTT), presheaf semantics, Agda

ACM Reference Format:
Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2025. BiSikkel: A Multimode Logical Framework in Agda. *Proc. ACM Program. Lang.* 9, POPL, Article 8 (January 2025), 31 pages. https://doi.org/10.1145/3704844

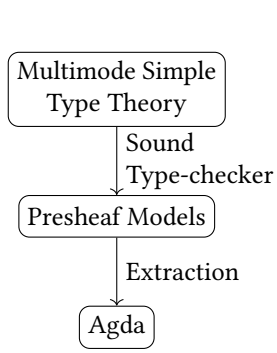
1 Introduction

In a modal type theory, all function types (and hence all variables in a context) are annotated with a modality, which enables the programmer to express aspects of a function's behavior that would be cumbersome or impossible to state in a standard type theory. Examples of such aspects include relational modalities [Nuyts and Devriese 2018] such as parametricity [Nuyts et al. 2017], irrelevance [Abel and Scherer 2012; Barras and Bernardo 2008; Miquel 2001; Mishra-Linger and Sheard 2008; Pfenning 2001; Reed 2003] and shape-irrelevance [Abel et al. 2017b]; guarded recursion [Atkey and McBride 2013; Clouston et al. 2017; Guatto 2018; Nakano 2000]; possibility and necessity [Pfenning and Davies 2001]; variance of functors [Abel 2006, 2008; Licata and Harper 2011; North 2018; Nuyts 2023a; Poiret et al. 2023]; freshness and transpension [Nuyts and Devriese 2024]; axiomatic cohesion [Licata and Shulman 2016]; and crispness/globality [Licata et al. 2018].

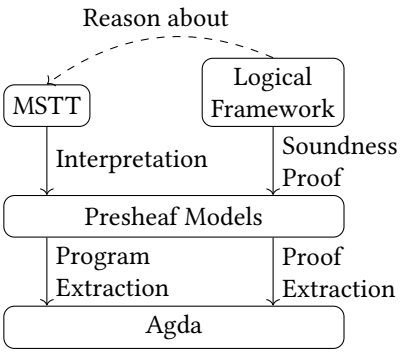
The integration of modalities in a proof assistant is an active area of research. We can generally distinguish two approaches. On the one hand, one can try to extend an existing proof assistant and provide support for some particular modal situations. Agda [Agda Development Team 2024], for

Authors' Contact Information: Joris Ceulemans, KU Leuven, Leuven, Belgium, joris.ceulemans@kuleuven.be; Andreas Nuyts, KU Leuven, Leuven, Belgium, andreas.nuyts@kuleuven.be; Dominique Devriese, KU Leuven, Leuven, Belgium, dominique.devriese@kuleuven.be.

ACM 2475-1421/2025/1-ART8
https://doi.org/10.1145/3704844



(a) Sikkel's architecture, figure taken from [Ceulemans et al. 2022].



(b) BiSikkel's architecture.

Fig. 1. Comparison between the architecture of Sikkel and BiSikkel.

example, has experimental support for applications such as guarded recursion [Veltri and Vezzosi 2020] and crisp type theory [Licata et al. 2018]. A downside of this approach is the fact that every new reasoning principle requires a new extension of the proof assistant's source code. Additionally, interactions between modal and other advanced language features often raise questions beyond the scope of existing theoretical research. On the other hand, `mitten` [Stassen et al. 2023] is, and Menkar [Nuyts and Devriese 2019] intended to be, a dedicated proof assistant specifically designed for modal type theory. They are both based on Multimode[1] Type Theory (MTT) [Gratzer et al. 2021], a family of modal type systems parameterized by a so-called *mode theory* which can be instantiated to obtain modal type systems for various applications. Although MTT puts a restriction on the modalities it can handle[2], the use of MTT has the advantage that important metatheoretic results such as canonicity, normalization [Gratzer 2022] and a substitution algorithm [Ceulemans et al. 2024c] can be dealt with for an arbitrary mode theory and will then apply to all instantiations of the framework, insofar as no further extensions are made.

In fact, for the implementation of modal type theories there is also a third approach recently explored by Ceulemans et al. [2022]. They developed Sikkel, a library for modal type theory written in Agda. Just like Menkar and `mitten`, Sikkel is based on MTT and is hence parameterized by a mode theory specifying the modal situation. Given such a mode theory, Sikkel's architecture is outlined in Fig. 1a. A library user can write programs in a syntactic layer, providing a deep embedding of Multimode Simple Type Theory (MSTT) in Agda. MSTT only differs from MTT in that it does not support dependent types.[3] Programs written at the syntactic layer cannot be evaluated immediately, but they can be interpreted in Sikkel's semantic layer, provided that they are well-typed. This semantic layer consists of a formalization in Agda of a broad class of models of type theory known as presheaf models [Hofmann 1997]. Finally, the denotation of certain Sikkel programs can be extracted to obtain actual Agda programs. As such, Sikkel combines the modal generality of dedicated modal languages (in fact, it can be regarded as one) with the benefit of working in a non-specialized proof assistant (Agda in this case) that has a more mature ecosystem.

[1]The names Multimode and Multimodal Type Theory are used interchangeably for the same system MTT which supports both multiple modes and multiple modalities.
[2]Semantically, every modality should have a left adjoint. Shulman [2023] addresses this limitation to some extent.
[3]As a consequence, type-checking does not rely on an equational theory and in fact, Ceulemans et al. [2022] do not consider an equational theory on the syntax at all.

Another advantage of the library approach is that modal type theories can be used modularly: since Sikkel programs can be extracted to ordinary Agda programs, a user only needs to program in the library when the modalities are actually needed. Other parts of the development can be performed in ordinary Agda. This also makes it possible to instantiate Sikkel twice with different mode theories (even incompatible ones) in the same development.

However, since Sikkel's syntactic layer is simply typed, the library does not provide the possibility to prove properties of programs that are written in it. Instead, such properties would have to be proven in plain Agda, breaking the abstraction that Sikkel provides and forcing the user to interact with the presheaf models of Sikkel. Of course, reasoning about functions implemented in a multimode theory is best done in a multimode proof assistant. For reasons of performance and usability (Sections 3.3 and 5.1), we chose a design where all dependent types are extrinsic, i.e. the syntax for dependently typed terms is untyped and needs to be type-checked. Meanwhile, in order to keep the type signature of this proof-checker tractable, we chose to make the programs mentioned in these dependent types, intrinsically typed. This design choice precluded a fully dependently typed system and led to a two-layered system called BiSikkel, consisting of an intrinsically typed adaptation of the (formerly extrinsically typed) Sikkel library – which embeds MSTT – and an extrinsically typed multimode logical framework on top. We name this framework μLF: a subset of MTT that acts as a logical framework for specifying and proving properties about MSTT terms. One can think of this as a syntactic layer for proofs: a user will prove properties of modal Sikkel programs using corresponding modal reasoning principles.

The resulting architecture is given in Fig. 1b. Since Sikkel's internally constructed presheaf model *is* fully dependently typed, it can interpret both layers (although many additional laws and coherence properties needed to be proven in order to handle modal type dependency). As dependency again played an important role in the extraction mechanism, we ended up with an architecture that reflects the two-layered syntax, with separate extraction functions for programs and for proofs (Section 5.3). Furthermore, BiSikkel's proof system is easy to extend with new proof rules or proposition formers, as long as they can be interpreted in the model.

Contributions.

- μLF: a novel multimode logical framework for MSTT, carved out of MTT (Section 4).
- BiSikkel: an implementation of MSTT (intrinsically typed, Section 2.1) and μLF (extrinsically typed with a type-checker, Section 5) as an Agda library [Ceulemans et al. 2024a,b], enabling modal proofs about modal MSTT terms in an off-the-shelf proof assistant[4]. BiSikkel is parametrized by a mode theory, just like MTT, MSTT, μLF and Sikkel.
  In order to accommodate dependency of BiSikkel propositions on Sikkel terms:
  - We make Sikkel intrinsically typed and extend it with an equational theory.
  - We give the first implementation of Ceulemans et al.'s modal substitution algorithm [2024c], now on intrinsically typed MSTT rather than on untyped syntax. Moreover, we prove its substitution lemma vis-à-vis Sikkel's presheaf model. (Section 5.4)
  - We implement a fueled normalization algorithm (in itself not novel [Stassen et al. 2023]), which outputs not just the normalized expression but also semantic evidence of its equality to the input expression. (Section 5.4)
- We prove BiSikkel sound by modeling μLF in a refined version of Sikkel's internal presheaf model of MTT, extended with additional proofs of various coherence and other laws. The proof checker outputs denotations of all well-typed proofs (Section 5.2).

[4]For the metatheory, we do assume function extensionality.

- We implement a proof extraction mechanism for the dependent layer. In order to do so, we extend Sikkel's program extraction mechanism to non-empty contexts (Section 5.3).
- We demonstrate the use of BiSikkel by proving properties about functions manipulating guarded streams (Section 4.3). Both the functions and proofs make use of BiSikkel's support for guarded recursive modes and modalities. The extraction mechanism for guarded streams requires the use of an axiom conflating bisimilarity and equality of Agda streams. Furthermore, we illustrate BiSikkel's flexibility with respect to the mode theory by also implementing an example involving unary parametricity (Section 4.4).

Overview of the Paper. We start with an overview of Ceulemans et al.'s Sikkel library [2022] in Section 2, mainly to keep this paper as self-contained as possible but also to highlight some differences with BiSikkel. Section 3 then introduces a running example with the concrete application of guarded recursive type theory and motivates BiSikkel's architecture. The core of the paper is formed by Sections 4 and 5, in which we describe the proof system μLF and its Agda implementation. We conclude with future and related work in Section 6.

2 A Brief Overview of Sikkel

In this section we describe all aspects of Sikkel that are needed to understand the rest of the paper. For more details we refer to [Ceulemans et al. 2022]. Note however, that we have reworked some parts of the library, as will be discussed more extensively here.

2.1 Syntactic Layer: Multimode Simple Type Theory (MSTT)

Sikkel's syntactic layer consists of a deep embedding in Agda of Multimode Simple Type Theory (MSTT), which is Multimodal Type Theory (MTT) [Gratzer et al. 2021] restricted to simple types.

Mode Theory. Just like MTT, MSTT is parameterized by a mode theory consisting of various components. First of all, a mode theory should specify a set of modes.[5] In MSTT, every type, context and term lives at a certain mode. All standard type theory constructions (functions, products, natural numbers, …) are available at every mode. However, some specific new type or term formers might only exist in one particular mode. For example, in a system for guarded type theory, Löb induction (Section 3.1) will be available only at a mode for time-dependent types and terms.

Next, a mode theory specifies for every two modes m and n a set of modalities from m to n. These will enable a programmer to transport types and terms from mode m to mode n. The modes and modalities should form a category, i.e. there should be a designated unit modality for every mode, modalities should compose and these operations should obey the category laws.

Finally, for every two modalities μ and ρ from m to n, there is a set of two-cells from μ to ρ, which are coercions between modalities. This makes a mode theory a strict 2-category (as we also require a vertical and horizontal composition of two-cells and a unit two-cell for every modality).

In the Agda implementation of Sikkel, a mode theory is a record collecting the different components described above. In other words, for a given mode theory there are Agda types Mode of modes, Modality m n of modalities from m to n for every m, n : Mode, and TwoCell μ ρ of two-cells from μ to ρ for every two μ, ρ : Modality m n. The unit modality is written as $\mathbb{1}$ and composition of

[5] Often, there is just a single mode. In that case, all modalities are endomodalities and form a monoid. Additionally, often, two-cells between given modalities are unique when they exist, in which case the endomodalities form an ordered monoid.

```
data Ty (m : Mode) : Set where
  Nat' : Ty m
  Bool' : Ty m
  ⟨_|_⟩⇛_ : Modality n m → Ty n → Ty m
            → Ty m
  _⊠_ : Ty m → Ty m → Ty m
  ⟨_|_⟩ : Modality n m → Ty n → Ty m
```

```
data Ctx (m : Mode) : Set where
  ◇ : Ctx m
  _„_|_∈_ : (Γ : Ctx m) (μ : Modality n m)
            (x : String) (T : Ty n) → Ctx m
  _,lock⟨_⟩ : (Γ : Ctx n) (μ : Modality m n)
              → Ctx m
```

Fig. 2. Definition of Sikkel types and contexts.[7]

modalities μ and ρ is written as $\mu \mathbin{ⓜ} \rho$.[6] Furthermore, a Sikkel mode theory contains proofs of *some* 2-category laws, but it is of course good practice to satisfy all of them.

Types and Contexts. When a mode theory is fixed, Sikkel provides an Agda type `Ty m` of MSTT types that live at mode m. It is defined in Fig. 2. For every mode this contains at least product types $A \boxtimes B$, and types `Bool'` and `Nat'` of Booleans and natural numbers. Furthermore, every modality μ from m to n induces a type constructor transforming a type T : `Ty m` to a boxed type $\langle\ \mu \mid T\ \rangle$: `Ty n`. Function types are annotated with a modality and have the form $\langle\ \mu \mid A\ \rangle \Rrightarrow B$, where μ is a modality from m to n, A is a type at mode m and B a type at mode n. The function type itself then lives at mode n. In case the modality μ is the unit modality, we will write this function type as $A \Rightarrow B$. Finally, Sikkel can be easily extended with custom types that are specific for certain applications. We will see an example of this in Section 3.1.

All MSTT contexts also live at a certain mode, and Sikkel provides the Agda type `Ctx m` whose implementation is presented in Fig. 2. Basically, a context is a possibly empty list of variables and locks. Every variable has a name of type `String` and is annotated with a modality (note that the domain of this modality has to match the mode of the variable's type). Locks will play an important role in the typing rules for modal (boxed) types, as they serve in some sense as a left adjoint to the modal type former. We point out that locks act contravariantly on contexts, i.e. a modality from m to n induces a lock operation from `Ctx n` to `Ctx m`.

Terms. Figure 3 lists the most interesting typing rules of MSTT. Note that all rules make sure that the context, term and type within one judgment live at the same mode. This mode is indicated at the end of the judgment, e.g. a judgment at mode m ends in "$@\ m$". Variables carry a form of De Bruijn index i (which we will omit), so the string x is only there for human readability and shadowing is immaterial. As in MTT, variables (Tm-Var) can only be used when there is a two-cell from the modality the variable is annotated with to the composition of all the locks in the context to the right of that variable, as computed by the function 'locks' (Fig. 3). Most of the times, this will just be the unit two-cell and then we will write `svar x` instead of `var x id-cell`. Modal functions can be introduced via lambda abstraction (Tm-Lam); the bound variable will be annotated with the provided modality. When applying a μ-modal function, the argument will be checked in a context locked with μ (Tm-App). In order to construct a term of the boxed type $\langle\ \mu \mid T\ \rangle$, it is sufficient to construct a term of type T after locking the context with μ (Tm-Mod-Intro). The modal elimination rule Tm-Mod-Elim

[6] The actual implementation of a mode theory in BiSikkel is slightly more complicated than presented here. It makes sure that e.g. the unit modality $\mathbb{1}$ is a left unit for the composition definitionally rather than propositionally. Moreover, the representation of mode theories in BiSikkel differs at some points from the one by Ceulemans et al. [2022]. However, these details only matter for the implementers of new mode theories and are not important to understand the rest of the paper.

[7] (1) Here `Set` is the Agda universe of types. (2) Identifiers with underscores are *mixfix* operators and the underscores indicate where the explicit arguments are expected.

$$\textsc{Tm-Var}\quad \frac{\alpha \in \mu \Rightarrow \mathsf{locks}(\Delta) \qquad \mathsf{idx}(\Delta) = i}{\Gamma \mathbin{,,} \mu \mid x \in T \mathbin{,} \Delta \vdash \mathsf{var}_i\ x\ \alpha : T \mathbin{@} m}$$

$$\mathsf{locks}(\diamond) = \mathbb{1}$$
$$\mathsf{locks}(\Gamma \mathbin{,,} \mu \mid x \in T) = \mathsf{locks}(\Gamma)$$
$$\mathsf{locks}(\Gamma \mathbin{,} \mathsf{lock}\langle\ \mu\ \rangle) = \mathsf{locks}(\Gamma)\ \textcircled{m}\ \mu$$

$$\textsc{Tm-Lam}\quad \frac{\mu : \mathsf{Modality}\ m\ n \qquad \Gamma \mathbin{,,} \mu \mid x \in T \vdash s : S \mathbin{@} n}{\Gamma \vdash \mathsf{lam}[\ \mu \mid x \in T\ ]\ s : \langle\ \mu \mid T\ \rangle \Rrightarrow S \mathbin{@} n}$$

$$\textsc{Tm-App}\quad \frac{\mu : \mathsf{Modality}\ m\ n \qquad \Gamma \vdash f : \langle\ \mu \mid T\ \rangle \Rrightarrow S \mathbin{@} n \qquad \Gamma \mathbin{,} \mathsf{lock}\langle\ \mu\ \rangle \vdash t : T \mathbin{@} m}{\Gamma \vdash f \cdot t : S \mathbin{@} n}$$

$$\textsc{Tm-Mod-Intro}\quad \frac{\mu : \mathsf{Modality}\ m\ n \qquad \Gamma \mathbin{,} \mathsf{lock}\langle\ \mu\ \rangle \vdash t : T \mathbin{@} m}{\Gamma \vdash \mathsf{mod}\langle\ \mu\ \rangle\ t : \langle\ \mu \mid T\ \rangle \mathbin{@} n}$$

$$\textsc{Tm-Mod-Elim}\quad \frac{\mu : \mathsf{Modality}\ m\ n \qquad \rho : \mathsf{Modality}\ n\ o \qquad \Gamma \mathbin{,} \mathsf{lock}\langle\ \rho\ \rangle \vdash t : \langle\ \mu \mid T\ \rangle \mathbin{@} n \qquad \Gamma \mathbin{,,} \rho\ \textcircled{m}\ \mu \mid x \in T \vdash s : S \mathbin{@} o}{\Gamma \vdash \mathsf{let}\langle\ \rho\ \rangle\ \mathsf{mod}\langle\ \mu\ \rangle\ x \leftarrow t\ \mathsf{in'}\ s : S \mathbin{@} o}$$

Fig. 3. Selected MSTT typing rules and definition of the function 'locks'.

allows in some sense to pattern match on a term of a boxed type: when constructing a term of type S, any term t of type $\langle\ \mu \mid T\ \rangle$ can be used as if it were of the form $\mathsf{mod}\langle\ \mu\ \rangle\ x$ for some variable x.

Intrinsically Typed Representation of Terms. When formalizing MSTT in Agda, there are several options for encoding its terms and typing rules. This is one of the aspects where BiSikkel differs significantly from Sikkel. Sikkel uses a so-called extrinsically typed encoding of terms, which means that it has a data type Tm that is indexed by a mode, but not by an MSTT context or type. In other words, values of type Tm m are only guaranteed to respect the mode system, but they are not necessarily well-typed. Sikkel therefore needs a type checker, as indicated in Fig. 1a.

In an intrinsically typed encoding, as adopted by BiSikkel, the typing rules from Fig. 3 are directly integrated into the Agda type of terms, which looks as follows:[8]

```
data Tm : {m : Mode} → Ctx m → Ty m → Set where
  mod⟨_⟩_ : (μ : Modality m n) → Tm (Γ ,lock⟨ μ ⟩) T → Tm Γ ⟨ μ ∣ T ⟩
  lam[_∣_∈_]_ : (μ : Modality m n) (x : String) (T : Ty m) →
    Tm (Γ ,, μ ∣ x ∈ T) S → Tm Γ (⟨ μ ∣ T ⟩⇛ S)
  ...
```

Apart from being unable to represent ill-typed terms, this encoding also has the benefit of integrating well with Agda's interactive development features, as the expected context and type of a program will be visible in an Agda goal. A further advantage of intrinsic typing will be covered in Section 4.1.

A downside of the intrinsically typed approach is that we cannot silently coerce terms along type equalities. This is not too consequential in MSTT, where the only source of non-trivial type equalities are equalities between modalities. In an extrinsically typed system, we could define modality equality in a specific mode theory to be any (semi-)decidable relation we like. In an intrinsically typed system, we either have to introduce a term constructor for conversion, or explicitly invoke transport along Agda's propositional equality, which could be stricter than the relation we had envisioned. In particular, the extrinsic approach makes it easier to write modality-polymorphic programs, as they only need to be well-typed after instantiation.

Unlike Ceulemans et al. [2022], we opted for the intrinsically typed approach to keep the type signature of the proof-checker tractable (Sections 3.3 and 5.2).

[8]Curly braces indicate implicit arguments, which can be omitted or passed again in curly braces.

Table 1. Interpretation of the MSTT syntax in the semantic layer.

Syntax layer	Semantic layer
mode	base category
context	presheaf
type	dependent presheaf
term	dependent presheaf morphism
modality	dependent adjunction
two-cell	natural transformation

2.2 Semantic Layer: Presheaf Models

One of the motivations for developing modal type theory as a library is to be able to integrate programs that use modal primitives inside projects written in "mainstream" proof assistants. However, we cannot directly interpret the syntax from the previous section as normal Agda programs because they contain constructs like modalities that are not present in standard Agda. Sikkel therefore has a semantic layer, consisting of a formalization of presheaf models [Hofmann 1997] as indicated in Fig. 1a. We will not discuss the details of this formalization as they are not important for the rest of the paper and, apart from some technicalities, nothing much has changed with respect to [Ceulemans et al. 2022]. An overview of the interpretation can be found in Table 1.

MSTT contexts and types are interpreted as presheaves.[9] Very broadly speaking, this means that they are interpreted as diagrams consisting of Agda types and Agda functions. The shape of such a diagram is determined by a parameter of the presheaf model called the base category. In (Bi)Sikkel, a mode theory should provide a mapping from modes to base categories. In other words, types from different modes will be interpreted as diagrams of different shapes. The interpretation of a term then provides an Agda value of every type in this diagram, in such a way that these values are stable under the Agda functions of the diagram.

Implementers of a new mode theory also have to provide interpretations of modalities and two-cells. Semantically, a modality corresponds to a dependent adjunction: a left adjoint functor acting on semantic contexts and substitutions (i.e. presheaves and presheaf morphisms) modeling the lock operation, supplemented with a dependent right adjoint (DRA) [Birkedal et al. 2020] acting on types and modeling the modal box type formers.[10] Two-cells are then interpreted as natural transformations between the left adjoint functors, from which transformations between the right adjoints can be derived.

2.3 Extraction to the Metalevel

Every mode theory in BiSikkel has a distinguished trivial mode called $\star$, which is interpreted as the trivial base category with only one object and its identity morphism. The resulting presheaf model is equivalent (but not equal) to the set model, so MSTT types at mode $\star$ are interpreted as very simple diagrams, namely ordinary Agda types, and terms of these types are interpreted ordinary Agda values. This makes it very easy to extract the types and terms in mode $\star$ to normal Agda programs. Terms in other modes can usually be extracted by first using a modality to transport them to the trivial mode. This approach nicely decouples program extraction from conversion between different models.

[9]Dependent ones in the case of types. The reason for this is that Sikkel's original model was already dependently typed; a fact which we will use when proving the soundness of the proof system.

[10]We remark that, since presheaf categories are democratic, a dependent adjunction is essentially the same as an adjunction whose right adjoint is a weak CwF morphism [Birkedal et al. 2020, lemma 17 and corr. 23].

However, the constructions in a presheaf model do not always lead to the desired Agda types for extraction. For example, the semantic type of standard streams in guarded type theory (Section 3.1) will be a highly involved type that is non-trivially isomorphic to the Agda type of streams. Sikkel therefore has a type class on types at mode $\star$, which provides the extraction mechanism with the intended target and proves it isomorphic to the type extracted from the presheaf model.

The extraction mechanism thus has two tasks: to convert between the trivial presheaf model and the set (i.e. type) model, and to convert between the extracted Agda type and the intended type. We build further upon this extraction mechanism as described in Section 5.3.

Note that, contrary to the situation in BiSikkel, the existence of a trivial mode is not required for a mode theory in MTT. However, this extra requirement does not pose a problem in practice for two reasons. First, since mode $\star$ is interpreted as the trivial (i.e. final) base category, the unique functor from any base category to this trivial category gives rise to two interesting DRAs between any presheaf model and the trivial presheaf model [nLab authors 2024].[11] Moreover, the trivial mode is actually only required in BiSikkel for extraction. If that feature is not necessary, one can always soundly add a trivial mode to an existing mode theory without adding interesting modalities to or from this mode, and then not use this mode in programs or proofs.

3 Why a Dedicated Logical Framework?

Before introducing the proof system μLF, we will take a look at why such a framework is necessary. In Section 3.1 we introduce a concrete instantiation of MSTT for guarded recursive type theory, also covered by Ceulemans et al. [2022]. When attempting to prove a property of some programs written in this MSTT instantiation, we will see in Section 3.2 why we need at least a framework for modal reasoning and what are the required features for a proof system like μLF. In Section 3.3, we discuss why we did not directly go for full-blown dependent types.

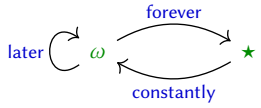
3.1 Motivating Example: Guarded Recursion

Proof assistants like Agda and Coq have support for manipulating potentially infinite objects, such as infinite streams. This works via corecursion: when constructing an infinite stream s, a programmer may corecursively refer to s itself. The stream of zeros is for instance uniquely characterized by $\mathsf{zeros} = 0 :: \mathsf{zeros}$. However, just like recursive definitions must pass a termination check in order for the proof assistant to be sound, corecursive definitions have to be productive, meaning that one should be able to compute every element of an infinite stream in a finite number of steps.

In order to ensure productivity, Agda and Coq use a syntactic check: basically, a corecursive call needs to be exactly located under a non-zero number of applications of the constructor $::$. However, this restriction is non-compositional and rejects many programs that are actually productive. For instance, the definition of the stream of natural numbers as $\mathsf{nats} = 0 :: \mathsf{map}\ \mathsf{suc}\ \mathsf{nats}$ is productive but not accepted.

To mitigate these problems, Nakano [2000] proposed to express the guarded behavior of functions in their type by means of a modal operator called the later modality. His theory also contains a fixpoint operator that explicitly takes this modality into account. This was later turned into a practical system for productive programming by Atkey and McBride [2013]. Gratzer et al. [2021] then fitted guarded recursive type theory into the MTT framework, elegantly using the ability to program in different modes. We can use their mode theory to obtain an instance of guarded recursion in Sikkel, as was also demonstrated by Ceulemans et al. [2022].

[11] Note however that the context functor for one of these DRAs can only be implemented in general in a metatheory with quotient types, though it is implementable in Agda for specific base categories (including all examples in this paper).



forever ⓜ later = forever
forever ⓜ constantly = 𝟙
𝟙 ≤ later
constantly ⓜ forever ≤ 𝟙

(a) Mode theory.

GStream : Ty ★ → Ty ω

Tm-GCons
$$\frac{\Gamma, \text{lock}\langle\ \text{constantly}\ \rangle \vdash a : A\ @\ \star \qquad \Gamma, \text{lock}\langle\ \text{later}\ \rangle \vdash s : \text{GStream}\ A\ @\ \omega}{\Gamma \vdash \text{g-cons}\ a\ s : \text{GStream}\ A\ @\ \omega}$$

Tm-GHead
$$\frac{\Gamma \vdash s : \text{GStream}\ A\ @\ \omega}{\Gamma \vdash \text{g-head}\ s : \langle\ \text{constantly}\ |\ A\ \rangle\ @\ \omega}$$

Tm-GTail
$$\frac{\Gamma \vdash s : \text{GStream}\ A\ @\ \omega}{\Gamma \vdash \text{g-tail}\ s : \langle\ \text{later}\ |\ \text{GStream}\ A\ \rangle\ @\ \omega}$$

Tm-Löb
$$\frac{\Gamma, \text{later} \mid x \in T \vdash t : T\ @\ \omega}{\Gamma \vdash \text{löb[later}|\ x \in T\ ]\ t : T\ @\ \omega}$$

(b) Additional type and term constructors.

Fig. 4. Mode theory and new type and term formers for guarded recursive type theory.

The mode theory in Fig. 4a contains two modes: a mode ω whose values can intuitively be seen as unfolding gradually over time, and a mode $\star$ whose values are intuitively ordinary Agda values and that do not have a time dependence. Consequently, the mode $\star$ will be interpreted as the trivial base category at the semantic level, whereas mode ω gets interpreted in the topos of trees (i.e. the presheaf category over the poset of natural numbers, [Birkedal et al. 2012]). The intuitive understanding of the base modalities is as follows: later delays the unfolding process with one time step, constantly embeds an ordinary value into the time-dependent world as a value that actually does not unfold, and forever creates an ordinary value by applying the unfolding process infinitely many times, making everything available at once. The mode theory for guarded recursion has at most one two-cell between any two modalities, and hence can be seen as a poset-enriched category rather than a 2-category.

Figure 4b illustrates that we can extend (Bi)Sikkel with type and term constructors that do not arise from modalities but which can mention modalities in their types. Guarded streams intuitively unfold over time by revealing one element at every time step. As a consequence, their tail is only available as a GStream one time step from now and hence has type ⟨ later | GStream A ⟩. Löb induction is a fixpoint operator; the variable x it binds corresponds to a corecursive call and the fact that it is annotated with later makes sure that definitions are productive.

```
g-map : Tm Γ ( ⟨ constantly | A ⇛ B ⟩⇛
                 GStream A ⇛ GStream B)
g-map = lam[ constantly | "f" ∈ A ⇛ B ]
  löb[later| "map" ∈ GStream A ⇛ GStream B ]
  lam[ "s" ∈ GStream A ]
    let' mod⟨ constantly ⟩ "s-head" ←
                          g-head (svar "s") in'
    let' mod⟨ later ⟩ "s-tail" ←
                          g-tail (svar "s") in'
    g-cons (svar "f" · svar "s-head")
           (svar "map" · svar "s-tail")
```

(a) Mapping a function over a guarded stream.

```
g-iterate g-iterate' : Tm Γ (⟨ later ⓜ constantly | A ⇛ A ⟩⇛
                              ⟨ constantly | A ⟩⇛ GStream A)
g-iterate = lam[ later ⓜ constantly | "f" ∈ A ⇛ A ]
  lam[ constantly | "a" ∈ A ] löb[later| "s" ∈ GStream A ]
  g-cons (svar "a")
         (g-map · svar "f" · svar "s")

g-iterate' = lam[ later ⓜ constantly | "f" ∈ A ⇛ A ]
  löb[later| "iter" ∈ ⟨ constantly | A ⟩⇛ GStream A ]
  lam[ constantly | "a" ∈ A ] g-cons (svar "a")
    (svar "iter" · (svar "f" · var "a" (𝟙≤ltr ⓣ-hor id-cell)))
```

(b) Two versions of `iterate` for guarded streams.

Fig. 5. Guarded stream examples in BiSikkel.

We can now write some definitions of functions that manipulate guarded streams. Figure 5a shows an implementation of the classic `map` function. Note that the elements of a guarded stream

live at mode $\star$, so we use the constantly modality to embed the mapped function into mode ω. Before constructing the resulting stream via g-cons, we use the modal eliminator (rule Tm-Mod-Elim from Fig. 3) twice to bind the head and tail of the argument stream to variables `"s-head"` and `"s-tail"`, annotated respectively with the constantly and later modality.[12] These variables can then be accessed via the unit two-cell because g-cons locks the context of its arguments with the right modalities. In Fig. 5b we find two alternative ways to produce a stream whose n-th element is computed by applying a given function n times to a given starting value (starting to count from $n = 0$). Note that in both versions, the generating function is only required when constructing the tail of the resulting stream. Hence it is not needed at the current time step, which explains the later modality in the type signature. The version g-iterate is defined in terms of g-map and intuitively corresponds to the corecursive definition

$$\text{iterate}\ f\ a = a :: \text{map}\ f\ (\text{iterate}\ f\ a), \tag{1}$$

which would not be accepted by a syntactic check employed by Agda or Coq. The alternative g-iterate' would be allowed, as it corresponds to

$$\text{iterate'}\ f\ a = a :: \text{iterate'}\ f\ (f\ a). \tag{2}$$

In the BiSikkel implementation, the variable `"a"` is annotated with modality constantly. When it is used to construct the tail of the resulting stream, the locks to the right of this variables compose to later ⓜ constantly so we access it via the horizontal composition of the two-cell witnessing that $\mathbb{1} \leq$ later and the trivial two-cell at constantly.

Guarded streams are somewhat inflexible with respect to their "time dependence". It is for instance impossible to write a function that only retains the values of a guarded stream at an even position. This issue can be very elegantly addressed by considering a type of standard streams, defined as $\text{Stream'}\ A = \langle\ \text{forever} \mid \text{GStream}\ A\ \rangle$. Since these standard streams live at the trivial mode $\star$, they can also be extracted to ordinary Agda streams. For more details we refer to Gratzer et al. [2021] and Ceulemans et al. [2022].

3.2 Proving a Property of g-iterate

In the previous section, we have seen two different implementations of the iterate function for guarded streams. As scientists caring about formal verification, we now want to rigorously prove that these two functions indeed produce the same result when applied to the same arguments. Although it is not a formal proof, we can get an idea of what such a proof entails by taking a look at the intuitive characterizations of iterate and iterate' from Equations (1) and (2). We start with a lemma about iterate, namely that map f (iterate f a) is equal to iterate f (f a). This can informally be argued as follows.

$$\begin{aligned}
\text{map}\ f\ (\text{iterate}\ f\ a) &= \text{map}\ f\ (a :: \text{map}\ f\ (\text{iterate}\ f\ a)) && \text{(3a)}\\
&= f\ a :: (\text{map}\ f\ (\text{map}\ f\ (\text{iterate}\ f\ a))) && \text{(3b)}\\
&= f\ a :: (\text{map}\ f\ (\text{iterate}\ f\ (f\ a))) && \text{(3c)}\\
&= \text{iterate}\ f\ (f\ a) && \text{(3d)}
\end{aligned}$$

Steps (3a) and (3d) are just an unfolding of the intuitive "definition" of iterate from (1) and step (3b) is the intuitive definition of map. The crucial step is (3c), where we coinductively apply the lemma we are proving. Making use of this lemma, a similar style of reasoning allows us to deduce that

$$\text{iterate}\ f\ a = a :: \text{map}\ f\ (\text{iterate}\ f\ a) = a :: \text{iterate}\ f\ (f\ a)\ =_{(4)}\ a :: \text{iterate'}\ f\ (f\ a)\ = \text{iterate'}\ f\ a. \tag{4}$$

[12] The notation for the modal eliminator used here is syntactic sugar for the one in Fig. 3 where ρ is the unit modality $\mathbb{1}$.

An obvious question is now whether and when it is actually permitted to coinductively apply a proposition in the proof of that same proposition. Agda does not allow coinduction in proofs of propositional equality, as equality is not a coinductive type, but one could define a coinductive bisimilarity relation on streams as a dependent stream of equality proofs, which can then be inhabited coinductively.[13] However, even if the definition of iterate from (1) were accepted by Agda's productivity checker, we would still not be able to prove that iterate and iterate' produce bisimilar streams. One problem is in the proof of step (3c) of the lemma, where we do not directly use the coinductive application of the lemma to prove the bisimilarity of the tails, but still have to apply a proof that map f preserves bisimilarity. As a result, this proof would not be accepted by Agda's syntactic productivity check. A similar problem arises in (4), where the result of coinduction gets combined with the lemma to prove the bisimilarity of the streams' tails.

Since this problem seems to be similar to the one solved by guarded recursion in Section 3.1, we might expect that the later modality could also play a role in the solution here. Indeed, we can prove the result above for guarded streams if we can apply the following two reasoning principles.

- In order to prove that g-cons a s equals g-cons b t, we need to show now that a equals b, but we only need to show that s is equal to t one time step from now. In other words, the second requirement would be a proof of $\langle$ later $\mid s = t \rangle$.
- We have a version of Löb induction for proofs: in order to prove a proposition φ, we may assume that $\langle$ later $\mid \varphi \rangle$ holds.

Note that both principles involve the application of a modality to a proposition. This illustrates the fact that, in order to prove useful properties of Sikkel programs, we need modal reasoning principles, and hence a modal logical framework.

3.3 A Logical Framework vs. Dependent Types

One could argue that logical frameworks have been obsoleted by the contemporary understanding of dependent types and indeed MSTT extended with our modal logical framework μLF is in its entirety a subset of MTT. In fact, the original goal of this work was to make Sikkel fully dependently typed, which would involve formalizing the syntax of MTT in Agda. For such a formalization of dependent type theory in dependent type theory, one has the same choice as in Section 2.1 between intrinsic and extrinsic typing.

In a non-modal setting, the intrinsically typed approach has for example been explored by Chapman [2009] and Altenkirch and Kaposi [2016], and is generally regarded as viable when it comes to *formalization* of syntax. The latter formalize the syntax as a quotient-inductive-inductive type (QIIT) [Altenkirch et al. 2018], which is essentially the same thing as a generalized algebraic theory (GAT) [Cartmell 1986]. This implies that definitional equality is not a separate judgment but rather coincides with the metalanguage's propositional equality, so that the conversion rule is not actually an inference rule but can be proven by transport. This is fine in *theory*, but makes the approach infeasible for a syntax in which users are actually expected to write programs, as they would be forced to explicitate every invocation of the conversion rule (which allows one to silently cast a term of type A to type B when $A = B$ definitionally or *even syntactically* up to propositional Agda equality), *with* an equality proof. Worse, these invocations will then proceed to haunt the user through type dependency. Bense et al. [2024] propose to at least compute away substitutions in intrinsic types. Furthermore, we observed that in a multimodal setting, the usage of intrinsic dependent types causes performance problems (Section 5.1).

[13] Note that, despite its mention here, we will not explicitly use bisimilarity of streams in the rest of this paper, except when proving that stream extraction is an isomorphism. In particular, the interpretation of a μLF proposition asserting equality of guarded streams will not state the bisimilarity of the streams' interpretations. See Section 5.3 for further details.

Abel et al. [2017a] give an extrinsically typed formalization of dependent type theory. Concretely, they provide a conversion algorithm, internally proven sound and complete, which would form the core part of a type-checker for the untyped syntax. However, rather than outputting a derivation (which can be regarded as intrinsically typed syntax indexed the extrinsically typed one), we want our type-checker to output a denotation in the presheaf model (Section 5.2). Hence Abel et al.'s approach was not readily usable for us. Perhaps, however, we could have taken a two-step approach where the type-checker produces a derivation which can then be recursively interpreted in the model (Section 6.1).

In particular, in order to keep the type signature of the proof-checker (the statement of what it does) tractable, we want a proof's context and proposition ('type') to be intrinsically typed, so that *their* denotation is already clear.

For all of the above reasons, for BiSikkel we opted for a two-layer architecture with a program layer of intrinsically typed MSTT and a proof layer of extrinsically typed μLF. This solves all of our problems: (1) The need for conversion in MSTT is limited as equality of modalities is the only source of non-trivial type equalities. (2) The performance problems of intrinsic modal types also did not occur for simple types. (3) The proof-checker knows the denotation of a proof's context and proposition beforehand.

4 μLF, A Proof System for MSTT

In this section we introduce the proof system μLF. The core of this system is the set of axioms and inference rules discussed in Section 4.2. The proof judgments in these rules mention propositions and proof contexts, which are introduced in Section 4.1. We illustrate the use of our framework in Section 4.3 with a formal proof in μLF of the equivalence of g-iterate and g-iterate' from Fig. 5b. Furthermore, we show that μLF can be applied to different modal situations by additionally constructing an example involving unary parametricity in Section 4.4.

4.1 Propositions & Proof Contexts

Although we only present μLF in this section and defer its Agda implementation to Section 5, the propositions and proof contexts of this system are introduced via their Agda definitions as this is the most concise way of presenting them.

Propositions. μLF has an Agda type of propositions – to be regarded as a separate judgment form – called bProp (for BiSikkel propositions). Its definition can be found in Fig. 6a. Since a proposition may mention terms, which in their turn can contain variables, the type bProp is indexed by a context specifying the variables in scope. Note that μLF propositions also always live at a certain mode, which is left implicit in the Agda type bProp Γ as it is exactly the mode of Γ.

There are standard proposition constructors for truth ($\top^{\mathsf{b}}$), falsehood ($\bot^{\mathsf{b}}$) and conjunction ($\wedge$).[14] Given two terms t and s of the same type T in a context Γ, we also have a proposition $t \equiv^{\mathsf{b}} s$ expressing that t and s are equal. Just like MSTT function types, bProp implications $\langle\ \mu \mid \varphi\ \rangle \supset \psi$ are annotated with a modality μ. When this modality μ is the unit modality $\mathbb{1}$, we write the implication as $\varphi \supset \psi$. Unsurprisingly, a universally quantified proposition $\forall[\ \mu \mid x \in T\ ]\ \varphi$ expresses that φ holds for all values x of type T. Here the proposition φ can mention x as it is bound in its context, annotated with the modality μ. Finally, there is a proposition equivalent of boxed types, resulting in propositions of the form $\langle\ \mu \mid \varphi\ \rangle$.

Proof Contexts. When proving a proposition, we do not only need to keep track of the MSTT variables that are in scope, but also of any assumptions that have already been made during the

[14] We had no need for disjunction, but it could be easily added.

```
data bProp : {m : Mode} → Ctx m → Set where
  ⊤ᵇ ⊥ᵇ : bProp Γ
  _∧_ : bProp Γ → bProp Γ → bProp Γ
  _≡ᵇ_ : Tm Γ T → Tm Γ T → bProp Γ
  ⟨_|_⟩⊃_ : (μ : Modality m n)
    → bProp (Γ ,lock⟨ μ ⟩) → bProp Γ → bProp Γ
  ∀[_|_∈_]_ : (μ : Modality m n) (x : String)
    (T : Ty m) → bProp (Γ „ μ | x ∈ T) → bProp Γ
  ⟨_|_⟩ : (μ : Modality m n) → bProp (Γ ,lock⟨ μ ⟩)
    → bProp Γ
```

(a) μLF propositions.

$$
\begin{aligned}
\mathsf{locks}(\diamond) &= \mathbb{1} \\
\mathsf{locks}(\Xi \mathbin{„^{v}} \mu \mid x \in T) &= \mathsf{locks}(\Xi) \\
\mathsf{locks}(\Xi \mathbin{„^{b}} \mu \mid x \in \varphi) &= \mathsf{locks}(\Xi) \\
\mathsf{locks}(\Xi \mathbin{,\mathsf{lock}\langle} \mu \rangle) &= \mathsf{locks}(\Xi) \mathbin{ⓜ} \mu
\end{aligned}
$$

(b) Definition of the 'locks' function for proof contexts.

```
data ProofCtx (m : Mode) : Set
to-ctx : {m : Mode} → ProofCtx m → Ctx m

data ProofCtx m where
  ◇ : ProofCtx m
  _„ᵛ_|_∈_ : ProofCtx m → (μ : Modality n m)
    (x : String) (T : Ty n) → ProofCtx m
  _„ᵇ_|_∈_ : (Ξ : ProofCtx m) (μ : Modality n m)
    → String → bProp (to-ctx Ξ ,lock⟨ μ ⟩)
    → ProofCtx m
  _,lock⟨_⟩ : ProofCtx n → Modality m n
    → ProofCtx m

to-ctx ◇ = ◇
to-ctx (Ξ „ᵛ μ | x ∈ T) = (to-ctx Ξ) „ μ | x ∈ T
to-ctx (Ξ „ᵇ _ | _ ∈ _) = to-ctx Ξ
to-ctx (Ξ ,lock⟨ μ ⟩) = (to-ctx Ξ) ,lock⟨ μ ⟩
```

(c) μLF proof contexts.

Fig. 6. Definition of μLF propositions and proof contexts.

proof. For example, a proof by induction on a natural number gives us an induction hypothesis that we may use. Therefore, the proof judgments discussed in the next section mention what we call proof contexts. They are basically lists of MSTT variables, bProp assumptions (that are also named by a string), and locks. The precise definition is shown in Fig. 6c.

As we can see, a proof context lives at a certain mode and hence the Agda type ProofCtx m is indexed by a mode m. The constructors for the empty proof context ($\diamond$), extension with an MSTT variable (_„ᵛ_|_∈_), and locks (_,lock⟨_⟩) are very similar to the ones for MSTT contexts of type Ctx m from Fig. 2. Adding an assumption to a proof context is somewhat more involved, since we should mention the program variables that that proposition may contain. These should be the MSTT variables that had already been bound in the proof context. Therefore, mutually with the definition of ProofCtx we define a function to-ctx that turns a proof context into an MSTT context by dropping all bProp assumptions.[15] This function can then be used in the type of the constructor _„ᵇ_|_∈_, making the definition of ProofCtx and to-ctx an example of induction-recursion. As we can see, assumptions are also named by a string in the proof context. This name will not really play a role in μLF, but it will be useful in the BiSikkel implementation.

4.2 Axioms & Inference Rules

Before we can present the μLF proof system, we first need to discuss some aspects of substitution and β-equivalence of MSTT terms.

Substitution. In some propositions and terms of the μLF inference rules, we need to substitute a variable by a term. The presence of modality annotations and locks in contexts makes this a considerably less easy task than in non-modal type theory [Ceulemans et al. 2024c]. Details about how representing substitutions as a data type and about the implementation of the substitution

[15] Note that the implementation of to-ctx contains two different uses of the symbol $\diamond$: the first one refers to a proof context, the second one to an MSTT context. Agda is smart enough to handle this kind of constructor overloading. Something similar happens in the case of locks.

algorithm for terms and propositions are not necessary for the understanding of μLF, so we defer them to Section 5.4. For now, it suffices to know that there is an Agda type Sub Γ Δ of substitutions from Γ to Δ. Such a substitution σ can be applied to a term t or a proposition φ in context Δ to obtain a term t [σ]tm (of the same type) or a proposition φ [σ]bprop in Γ. There are various ways to construct substitutions, but we will most often use the following Agda functions (which are actually implemented in terms of more basic constructors).

$$\begin{aligned} &\_/\_ : \mathsf{Tm}\ (\Gamma\ ,\mathsf{lock}\langle\ \mu\ \rangle)\ T \to (x : \mathsf{String}) \to \mathsf{Sub}\ \Gamma\ (\Gamma\ „\ \mu \mid x \in T) \\ &\_//\_ : \mathsf{Tm}\ (\Gamma\ „\ \rho \mid y \in S\ ,\mathsf{lock}\langle\ \mu\ \rangle)\ T \to (x : \mathsf{String}) \to \mathsf{Sub}\ (\Gamma\ „\ \rho \mid y \in S)\ (\Gamma\ „\ \mu \mid x \in T) \end{aligned} \tag{5}$$

Note in the first argument that when we use these functions to substitute a term t for a variable x that is annotated with a modality μ, the term t has to live in the context locked with μ.

β-equivalence for Terms. One of the key components of a type checker for a dependently typed language is its conversion checker. During type checking, it will automatically handle definitional equalities of the underlying type theory, which typically include β- and η-equivalence of terms and expansion of function definitions. Although BiSikkel is not dependently typed, requiring a user to mention all proof steps that involve β-equivalence quickly becomes impractical, even when writing proofs of small results. For μLF, we therefore decided to have automated β-equivalence of terms in the form of a β-normalization function that can be used in proofs. See Section 5.4 for more details. There is no automated η-expansion (as this is used significantly less frequently than β-reduction), but there are proof rules that can be manually used.

TmEq-Fun

$$\frac{\begin{array}{l}\mu : \mathsf{Modality}\ m\ n \\ \Gamma\ „\ \mu \mid x \in T \vdash s : S\ @\ n \\ \Gamma\ ,\mathsf{lock}\langle\ \mu\ \rangle \vdash t : T\ @\ m\end{array}}{\Gamma \vdash (\mathsf{lam}[\ \mu \mid x \in T\ ]\ s) \cdot t =^{\beta} s\ [\ t\ /\ x\ ]\mathsf{tm}\ @\ n}$$

TmEq-Mod

$$\frac{\begin{array}{l}\mu : \mathsf{Modality}\ m\ n \\ \rho : \mathsf{Modality}\ n\ o \\ \Gamma\ ,\mathsf{lock}\langle\ \rho\ \rangle\ ,\mathsf{lock}\langle\ \mu\ \rangle \vdash t : T\ @\ m \\ \Gamma\ „\ \rho \circledcirc \mu \mid x \in T \vdash s : S\ @\ o\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\langle\ \rho\ \rangle\ \mathsf{mod}\langle\ \mu\ \rangle\ x \leftarrow \mathsf{mod}\langle\ \mu\ \rangle\ t\ \mathsf{in'}\ s \\ \qquad =^{\beta} s\ [\ t\ /\ x\ ]\mathsf{tm}\ @\ o\end{array}}$$

Fig. 7. Selected rules for β-equivalence of MSTT terms.

For now, instead of considering a normalization function, we take an axiomatic approach for β-equivalence with some of the rules shown in Fig. 7. We omitted all rules making sure that $=^{\beta}$ is an equivalence relation and is respected by all term formers. The standard β-reduction for functions is expressed in TmEq-Fun. Note that the fact that t lives in a context locked by μ is exactly what is needed for the substitution on the right-hand side (see the type signature of _/_ in (5)) as well as for the function application on the left-hand side (see Tm-App in Fig. 3) of the conclusion. The rule TmEq-Mod covers what happens when the modal eliminator meets a modal constructor. The term on the right-hand side in the conclusion does actually not type-check: for that to be the case, t should actually live in the context Γ ,lock⟨ ρ ⓜ μ ⟩. In Gratzer et al.'s original formulation of MTT [2021], this is solved by having a notion of definitional context equalities, which makes Γ ,lock⟨ ρ ⓜ μ ⟩ equal to Γ ,lock⟨ ρ ⟩ ,lock⟨ μ ⟩. It has since been argued [Ceulemans et al. 2024c; Nuyts 2023b] that having such equalities is not necessarily a good idea, so MSTT does not have context equalities, but does provide operations to transfer terms between these two contexts. For simplicity of the exposition, we omit those in the paper.

As propositions may contain terms but do not themselves compute, we extend β-equivalence to bProp by asking that all proposition formers propagate β-equivalence.

The Proof System. We can now finally discuss the core of the proof system μLF. It consists of a set of inference rules that allow to derive judgments of the form $\Xi \vdash \varphi \mathbin{@} m$ where Ξ is a proof context and φ is a proposition, both living at mode m. The intended meaning of such a judgment is that φ is provable in context Ξ. An important invariant that we maintain in the system, is that the variables that are in scope in φ are exactly the MSTT variables of Ξ. In other words, whenever we can derive $\Xi \vdash \varphi \mathbin{@} m$, we have that $\varphi : \mathsf{bProp}\ (\mathsf{to\text{-}ctx}\ \Xi)$.

The built-in rules of μLF can be found in Fig. 8. Note that the BiSikkel implementation is extensible and users can add more application-specific reasoning principles, as long as they are able to model them in the presheaf model. The notion of β-equality is introduced in the proof system via Prf-Beta. Note that this non-algorithmic rule is not explicitly supported by the BiSikkel proof-checker. In fact, unlike most type- or proof-checking algorithms, for performance reasons BiSikkel will never automatically check for β-equality except when encountering an invocation of a special version of Prf-$\equiv^{\mathsf{b}}$-Refl. However, in combination with Prf-$\equiv^{\mathsf{b}}$-Subst, we retain the same expressivity.

Assumptions work similar to MSTT variables: to use one in a proof we need a two-cell from the modality it is annotated with to the composite of the locks to the right of the assumption (Prf-Assumption, the 'locks' function for proof contexts is defined in Fig. 6b). However, since the proposition φ lives at $\mathsf{to\text{-}ctx}\ \Xi$, it is not necessarily well-formed in $\Xi \mathbin{„^{\mathsf{b}}} \mu \mid x \in \varphi$, Θ as the telescope Θ might contain MSTT variables and locks. The inference rule therefore allows us to deduce $\varphi^{\Theta,\alpha}$, which is basically φ but with the two-cell annotations modified so that they make sense in the new proof context. Internally, we apply a renaming to φ that inserts weakenings and two-cell modifications based on Θ and α. This phenomenon also applies to MTT, we refer to Ceulemans et al. [2024c] for more details.

The next batch of rules provide ways to prove and use equality of terms (Prf-$\equiv^{\mathsf{b}}$-…). Of course, Prf-$\equiv^{\mathsf{b}}$-Sym and Prf-$\equiv^{\mathsf{b}}$-Trans follow from Prf-$\equiv^{\mathsf{b}}$-Refl and Prf-$\equiv^{\mathsf{b}}$-Subst but given that BiSikkel currently lacks user-friendly declarations, we include these lemmas as primitives for reasons of usability. Important to note is that the rule Prf-$\equiv^{\mathsf{b}}$-Subst is stronger than the elimination principle for propositional equality in MTT. A translation of the MTT principle to μLF would give rise to the current rule specialized to the case where $\mu = \mathbb{1}$. One benefit of our rule is that it can be used to prove that modal functions of type $\langle\ \mu \mid A\ \rangle\!\Rrightarrow B$ respect the bProp equality $\equiv^{\mathsf{b}}$. In fact, since $\equiv^{\mathsf{b}}$ is modeled as propositional equality in the presheaf model, the principle Prf-$\equiv^{\mathsf{b}}$-Subst is sound for any instantiation of our library, and hence we decide to add it to μLF. The principle is also discussed by Gratzer [2022]; Gratzer and Birkedal [2022] who call it crisp identity induction.[16]

The following rules are introduction and elimination rules for many of the proposition formers from Fig. 6a. The μLF treatment of conjunction, truth and falsehood is entirely standard (Prf-∧-…, Prf-$\top^{\mathsf{b}}$-Intro, Prf-$\bot^{\mathsf{b}}$-Elim). Modal implications follow the same pattern as modal functions in Fig. 3 (Prf-⊃-…). The rules for modal propositions Prf-Mod-Intro and Prf-Mod-Elim are similar to their MSTT counterparts Tm-Mod-Intro and Tm-Mod-Elim. Finally, the rules for universal quantification (Prf-∀-…) are inspired by those for modal dependent functions in MTT.

The built-in definitional equality of μLF does not cover η-expansion. However, a user of the system can manually employ this principle for functions and products by making use of the rules Prf-⇛-η and Prf-⊠-η. In the η-rule for functions, the term f has to be weakened as it lives in context $\mathsf{to\text{-}ctx}\ \Xi$ and not $\mathsf{to\text{-}ctx}\ \Xi \mathbin{„} \mu \mid x \in T$. This can be done via the substitution $\pi : \mathsf{Sub}\ (\Gamma \mathbin{„} \mu \mid x \in T)\ \Gamma$.

Next are induction principles for some of the MSTT types (Prf-…-Induction). They correspond to the dependent eliminators for Booleans, natural numbers and modal types in MTT. For example, the last premise of Prf-Nat-Induction is the classical induction step in a proof by induction for natural

[16] Licata et al.'s flat modality ♭ [2018] disrespects this rule *if we read* $\equiv^{\mathsf{b}}$ *as the path type in HoTT*. However, this path type is not modeled as propositional equality, so there is no contradiction.

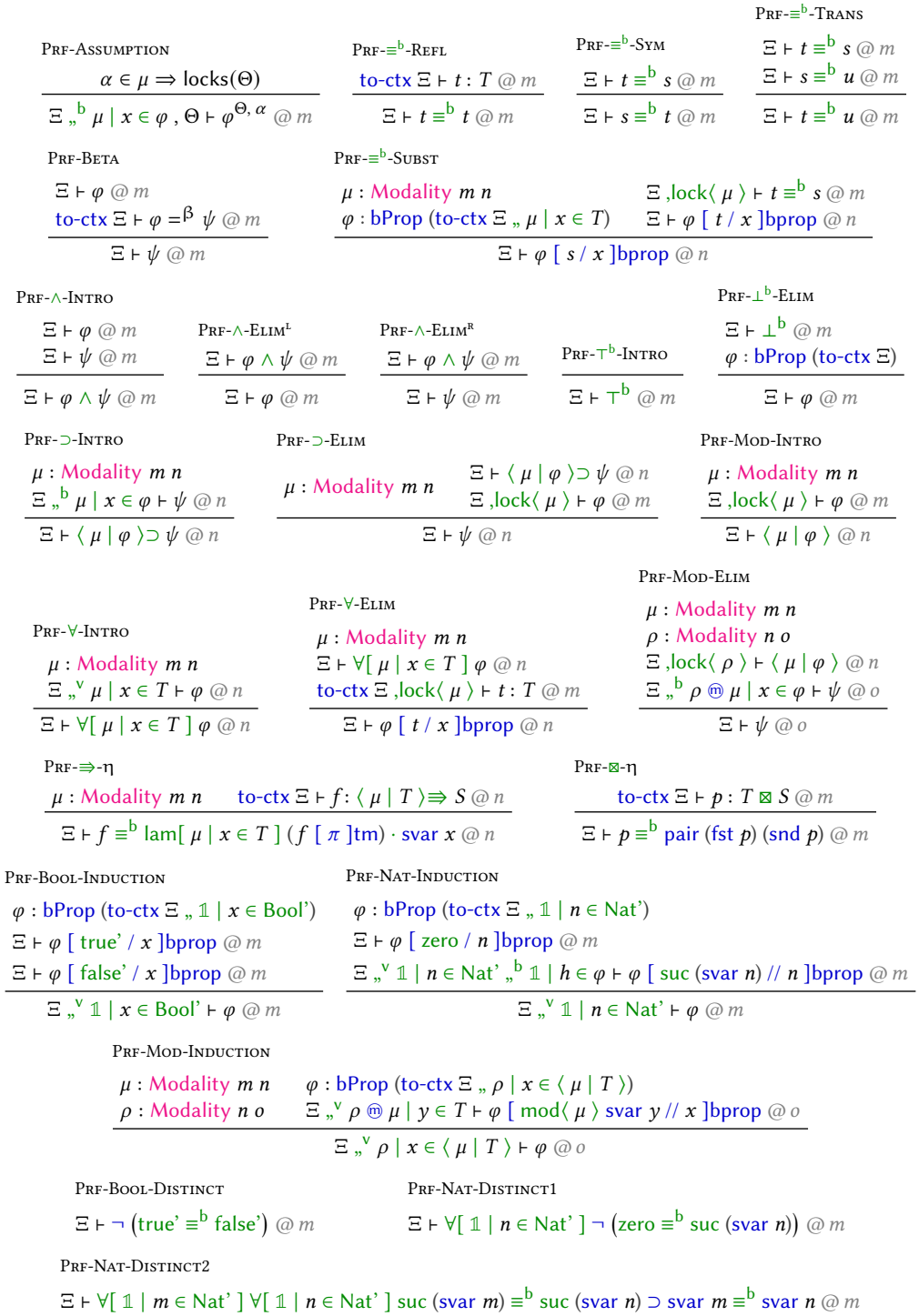


Fig. 8. Inference rules and axioms of the μLF proof system.

$$\textsc{TmEq-GStream-Head}\quad \frac{\Gamma\ ,\mathsf{lock}\langle\ \mathsf{constantly}\ \rangle \vdash a : A\ @\ \star \qquad \Gamma\ ,\mathsf{lock}\langle\ \mathsf{later}\ \rangle \vdash s : \mathsf{GStream}\ A\ @\ \omega}{\Gamma \vdash \mathsf{g\text{-}head}\ (\mathsf{g\text{-}cons}\ a\ s) =^{\beta} \mathsf{mod}\langle\ \mathsf{constantly}\ \rangle\ a\ @\ \omega}$$

$$\textsc{TmEq-GStream-Tail}\quad \frac{\Gamma\ ,\mathsf{lock}\langle\ \mathsf{constantly}\ \rangle \vdash a : A\ @\ \star \qquad \Gamma\ ,\mathsf{lock}\langle\ \mathsf{later}\ \rangle \vdash s : \mathsf{GStream}\ A\ @\ \omega}{\Gamma \vdash \mathsf{g\text{-}tail}\ (\mathsf{g\text{-}cons}\ a\ s) =^{\beta} \mathsf{mod}\langle\ \mathsf{later}\ \rangle\ s\ @\ \omega}$$

$$\textsc{Prf-TmLöb-Beta}\quad \frac{\mathsf{to\text{-}ctx}\ \Xi\ „^{\mathsf{b}}\ \mathsf{later}\ |\ x \in T \vdash t : T\ @\ \omega}{\Xi \vdash \mathsf{löb[later|}\ x \in T\ \mathsf{]}\ t \equiv^{\mathsf{b}} t\ [\ (\mathsf{löb[later|}\ x \in T\ \mathsf{]}\ t)^{\mathbb{1} \le \mathsf{ltr}}\ /\ x\ ]\mathsf{tm}\ @\ \omega}$$

$$\textsc{Prf-Löb}\quad \frac{\Xi\ „^{\mathsf{b}}\ \mathsf{later}\ |\ x \in \varphi^{\mathbb{1} \le \mathsf{ltr}} \vdash \varphi\ @\ \omega}{\Xi \vdash \varphi\ @\ \omega}$$

Fig. 9. β-equivalence and proof rules specific to guarded recursive type theory.

numbers: we assume the proposition φ that we try to prove, and are then required to prove φ with n substituted by $\mathsf{suc}\ n$. The modal induction principle allows us to see any variable of type $\langle\ \mu\ |\ T\ \rangle$ as a term of the form $\mathsf{mod}\langle\ \mu\ \rangle\ \mathsf{svar}\ y$ for some variable y. Note that all induction principles have conclusions of the form $\Xi\ „^{\mathsf{v}}\ \mu\ |\ x \in T \vdash \varphi\ @\ m$ (possibly with a specific value for μ), i.e. the context is extended with a variable. This is typically avoided when designing dependent type systems, as it is unclear how substitutions should be propagated through term formers like this. Therefore, it is more common to encounter an induction principle, for example for natural numbers, that would take an extra premise $\mathsf{to\text{-}ctx}\ \Xi \vdash m : \mathsf{Nat'}\ @\ o$ and produce a conclusion of the form $\Xi \vdash \varphi\ [\ m\ /\ n\ ]\mathsf{bprop}\ @\ o$. However, since we are not interested in the computational behavior of proofs and μLF does not even have a notion of substitution between proof contexts, the induction rules in Fig. 8 are unproblematic.

Finally, there are some axioms that assert the distinctness of the constructors for the types Bool' and Nat' (Prf-Bool-Distinct, Prf-Nat-Distinct1 and Prf-Nat-Distinct2). Here we make use of the negation of propositions defined as $\neg\ \varphi\ =\ \varphi \supset \bot^{\mathsf{b}}$. These axioms would be provable from big elimination principles, which would let us construct predicates over Bool' (and Nat') by case distinction (recursion). However, for simplicity, rather than adding big elimination, we choose to add just a few corollaries that we are interested in.

4.3 Continuing the g-iterate Example

Now that we have the proof system μLF at our disposal, we can return to the example from Section 3.2. Before working out the proof, we have to extend the framework with specific proof rules related to guarded recursion. These can be found in Fig. 9.

First of all, we extend the β-equivalence of our MSTT instance for guarded recursion with computation rules for guarded streams (TmEq-GStream-Head and TmEq-GStream-Tail). There should also be a computation rule associated to Löb induction for terms, expressing that it indeed generates fixpoints. This can however not be part of the β-equivalence for terms because that would lead to loss of decidability of conversion as shown by Gratzer and Birkedal [2022]. As a result, we add this principle as an axiom Prf-TmLöb-Beta that has to be manually invoked in proofs. In the right-hand side of the conclusion, we need to substitute x with a term in the context $\mathsf{to\text{-}ctx}\ \Xi\ ,\mathsf{lock}\langle\ \mathsf{later}\ \rangle$ but $\mathsf{löb[later|}\ x \in T\ \mathsf{]}\ t$ is a term in context $\mathsf{to\text{-}ctx}\ \Xi$. We can however transform a term from the latter context to the former since there is a two-cell from $\mathbb{1}$ to later. This is what the construction $\_^{\mathbb{1} \le \mathsf{ltr}}$ does: internally this is again a renaming (similar to the conclusion of Prf-Assumption) that modifies the two-cells that accompany the variables in a term. Finally, we also have a version of Löb induction for propositions: in order to prove φ, we may assume φ under a later modality (Prf-Löb).

We now start with the lemma from Section 3.2. For the sake of readability and space, we will from now on in this section just write a variable name "x" instead of svar "x" and similarly

"x"$^{\alpha}$ instead of var "x" α. We also define the abbreviations γ for the two-cell from constantly to later ⓜ constantly obtained as $\mathbb{1}\leq$ltr ⓣ-hor id-cell and δ for the two-cell $\mathbb{1}\leq$ltr ⓣ-hor γ from constantly to later^2 ⓜ constantly. Furthermore, in any context where it makes sense, we use the abbreviation

$$\textbf{lem} = \text{g-map} \cdot \text{"f"} \cdot (\text{g-iterate} \cdot \text{"f"}^{\gamma} \cdot \text{"a"}) \equiv^{\text{b}} \text{g-iterate} \cdot \text{"f"}^{\gamma} \cdot (\text{"f"} \cdot \text{"a"}).$$

We then want to prove in any context that $\forall[\ \text{constantly} \mid \text{"f"} \in A \Rrightarrow A\ ]\ \forall[\ \text{constantly} \mid \text{"a"} \in A\ ]\ \textbf{lem}$. Building the proof from bottom to top, we can proceed as follows.

$$\dfrac{\dfrac{\dfrac{\Xi \,„^{\text{v}}\, \text{constantly} \mid \text{"f"} \in A \Rrightarrow A \,„^{\text{v}}\, \text{constantly} \mid \text{"a"} \in A \,„^{\text{b}}\, \text{later} \mid \text{"ih"} \in \textbf{lem}^{\mathbb{1}\leq\text{ltr}} \vdash \textbf{lem}\ @\ \omega}{\Xi \,„^{\text{v}}\, \text{constantly} \mid \text{"f"} \in A \Rrightarrow A \,„^{\text{v}}\, \text{constantly} \mid \text{"a"} \in A \vdash \textbf{lem}\ @\ \omega}\ \textsc{Prf-Löb}}{\Xi \,„^{\text{v}}\, \text{constantly} \mid \text{"f"} \in A \Rrightarrow A \vdash \forall[\ \text{constantly} \mid \text{"a"} \in A\ ]\ \textbf{lem}\ @\ \omega}\ \textsc{Prf-∀-Intro}}{\Xi \vdash \forall[\ \text{constantly} \mid \text{"f"} \in A \Rrightarrow A\ ]\ \forall[\ \text{constantly} \mid \text{"a"} \in A\ ]\ \textbf{lem}\ @\ \omega}\ \textsc{Prf-∀-Intro} \tag{6}$$

The question remains how to prove the equality at the top of this derivation. This will be the following sequence of equality proofs, chained together with the rule Prf-≡$^{\text{b}}$-Trans.

$$\begin{aligned}
&\text{g-map} \cdot \text{"f"} \cdot (\text{g-iterate} \cdot \text{"f"}^{\gamma} \cdot \text{"a"}) \\
&\quad \equiv^{\text{b}} \text{g-map} \cdot \text{"f"} \cdot (\text{löb}[\text{later}\mid \text{"s"} \in \text{GStream}\ A\ ]\ \text{g-cons}\ \text{"a"}\ (\text{g-map} \cdot \text{"f"}^{\gamma} \cdot \text{"s"})) && (7)\\
&\quad \equiv^{\text{b}} \text{g-map} \cdot \text{"f"} \cdot (\text{g-cons}\ \text{"a"}\ (\text{g-map} \cdot \text{"f"}^{\gamma} \cdot (\text{g-iterate} \cdot \text{"f"}^{\delta} \cdot \text{"a"}^{\gamma}))) && (8)\\
&\quad \equiv^{\text{b}} \text{g-cons}\ (\text{"f"} \cdot \text{"a"})\ (\text{g-map} \cdot \text{"f"}^{\gamma} \cdot (\text{g-map} \cdot \text{"f"}^{\gamma} \cdot (\text{g-iterate} \cdot \text{"f"}^{\delta} \cdot \text{"a"}^{\gamma}))) && (9)\\
&\quad \equiv^{\text{b}} \text{g-cons}\ (\text{"f"} \cdot \text{"a"})\ (\text{g-map} \cdot \text{"f"}^{\gamma} \cdot (\text{g-iterate} \cdot \text{"f"}^{\delta} \cdot (\text{"f"}^{\gamma} \cdot \text{"a"}^{\gamma}))) && (10)\\
&\quad \equiv^{\text{b}} \text{g-iterate} \cdot \text{"f"}^{\gamma} \cdot (\text{"f"} \cdot \text{"a"}) && (11)
\end{aligned}$$

Step (7) is just an unfolding of the definition of g-iterate (these are always transparent in BiSikkel), combined with two β-reductions for functions. Hence this step can be proved via Prf-Beta and TmEq-Fun, and will not be included in a mechanized version of the proof. In step (8) we make use of Prf-TmLöb-Beta where "s" gets substituted with the löb term itself. For readability, we also rewrote this löb term again to an application of g-iterate, which is just an instance of β-conversion. Note that the operation $\_^{\mathbb{1}\leq\text{ltr}}$ from Prf-TmLöb-Beta changed the two-cell annotations for the variables "f" and "a". Strictly speaking, we also made use of the fact that g-map respects the equality $\equiv^{\text{b}}$, a principle that can be proved using Prf-≡$^{\text{b}}$-Subst. Step (9) applies a lemma that we do not prove in the paper: it is the principle that mapping a function over a guarded stream constructed via g-cons amounts to applying the function to the head and mapping it over the stream's tail. This can essentially be proved via one application of the rule Prf-TmLöb-Beta. The most interesting step is (10), where we apply the induction hypothesis "ih" that we obtained via Löb induction in the derivation (6) above. Since this use of Löb induction is one of the motivations from Section 3.2 for having a modal framework, let us examine in more detail why we are allowed to apply the induction hypothesis here. First of all, we use the following principle for proving the equality of two streams constructed via g-cons.

Admissible-Gcons-Cong

$$\dfrac{\Xi\ ,\text{lock}\langle\ \text{constantly}\ \rangle \vdash a \equiv^{\text{b}} a'\ @\ \star \qquad \Xi\ ,\text{lock}\langle\ \text{later}\ \rangle \vdash s \equiv^{\text{b}} s'\ @\ \omega}{\Xi \vdash \text{g-cons}\ a\ s \equiv^{\text{b}} \text{g-cons}\ a'\ s'\ @\ \omega}$$

This rule is not built into μLF, but it is admissible and can be proved via Prf-≡$^{\text{b}}$-Subst. In step (10), the heads are identical, so the first premise of the rule above can be discharged with Prf-≡$^{\text{b}}$-Refl. Interesting is the fact that we need to prove the equality of the tails in a proof context locked with later. This means that we want to apply the induction hypothesis in the following proof context,

$$\Xi \,„^{\text{v}}\, \text{constantly} \mid \text{"f"} \in A \Rrightarrow A \,„^{\text{v}}\, \text{constantly} \mid \text{"a"} \in A \,„^{\text{b}}\, \text{later} \mid \text{"ih"} \in \textbf{lem}^{\mathbb{1}\leq\text{ltr}}\ ,\text{lock}\langle\ \text{later}\ \rangle$$

and this is indeed allowed via Prf-Assumption because we have the trivial two-cell from later to later. Note that the two-cell $\mathbb{1} \leq \mathsf{ltr}$ that is applied to **lem** in the induction hypothesis modifies the two-cell annotations in **lem** so that they exactly match those in step (10). Finally, step (11) is again a combination of β-reduction and Prf-TmLöb-Beta, similar to steps (7) and (8).

Proving the final result, the equivalence of g-iterate and g-iterate' now follows a similar pattern. If we write

$$\mathbf{res} = \mathsf{g\text{-}iterate} \cdot \texttt{"f"}^{\gamma} \cdot \texttt{"a"} \equiv^{\mathsf{b}} \mathsf{g\text{-}iterate'} \cdot \texttt{"f"}^{\gamma} \cdot \texttt{"a"},$$

we want to prove $\forall[\ \mathsf{constantly} \mid \texttt{"f"} \in A \Rrightarrow A\ ]\ \forall[\ \mathsf{constantly} \mid \texttt{"a"} \in A\ ]\ \mathbf{res}$. Consequent applications of Prf-∀-Intro, Prf-Löb, and again Prf-∀-Intro give us the following proof goal.

$$\Xi\ „^{\mathsf{v}}\ \mathsf{constantly} \mid \texttt{"f"} \in A \Rrightarrow A\ „^{\mathsf{b}}\ \mathsf{later} \mid \texttt{"ih"} \in \forall[\ \mathsf{constantly} \mid \texttt{"a"} \in A\ ]\ \mathbf{res}\ „^{\mathsf{v}}\ \mathsf{constantly} \mid \texttt{"a"} \in A \vdash \mathbf{res}.$$

Equational reasoning similar to the lemma then gives us the following steps.

$$\begin{aligned}
&\mathsf{g\text{-}iterate} \cdot \texttt{"f"}^{\gamma} \cdot \texttt{"a"} \\
&\quad \equiv^{\mathsf{b}} \mathsf{g\text{-}cons}\ \texttt{"a"}\ (\mathsf{g\text{-}map} \cdot \texttt{"f"}^{\gamma} \cdot (\mathsf{g\text{-}iterate} \cdot \texttt{"f"}^{\delta} \cdot \texttt{"a"}^{\gamma}) && (12)\\
&\quad \equiv^{\mathsf{b}} \mathsf{g\text{-}cons}\ \texttt{"a"}\ (\mathsf{g\text{-}iterate} \cdot \texttt{"f"}^{\delta} \cdot (\texttt{"f"}^{\gamma} \cdot \texttt{"a"}^{\gamma})) && (13)\\
&\quad \equiv^{\mathsf{b}} \mathsf{g\text{-}cons}\ \texttt{"a"}\ (\mathsf{g\text{-}iterate'} \cdot \texttt{"f"}^{\delta} \cdot (\texttt{"f"}^{\gamma} \cdot \texttt{"a"}^{\gamma})) && (14)\\
&\quad \equiv^{\mathsf{b}} \mathsf{g\text{-}iterate'} \cdot \texttt{"f"}^{\gamma} \cdot \texttt{"a"} && (15)
\end{aligned}$$

Steps (12) and (15) are again the unfolding of definitions, β-reduction and the use of Prf-TmLöb-Beta. In step (13) we use the lemma we just proved, combined with the Admissible-Gcons-Cong principle. Since this lemma contains two universal quantifications, we apply the rule Prf-∀-Elim twice in this step. Step (14) is then an application of the induction hypothesis. Again this is possible because we locked the context with later when proving that the tails of the two streams are equal. Note that it is crucial that we have an induction hypothesis of the form $\forall[\ \mathsf{constantly} \mid \texttt{"a"} \in A\ ]\ \mathbf{res}$, as we now want to apply it to the term $\texttt{"f"}^{\gamma} \cdot \texttt{"a"}^{\gamma}$, which is possible thanks to Prf-∀-Elim.

From the equivalence of g-iterate and g-iterate', it is a small step to prove a similar result about the counterparts of these functions for standard streams of type Stream' A (which are implemented in terms of the functions for guarded streams) at mode ⋆. We refer to the Agda formalization for the details [Ceulemans et al. 2024a,b]. Subsequently, we can apply the extraction mechanism to obtain an equivalence result about the extracted versions of iterate and iterate' for standard Agda streams (although some performance issues occur here, see Section 5.3).

4.4 Another Example: Unary Parametricity

In order to demonstrate BiSikkel's relevance to more than just a single mode theory, here we demonstrate how its logical framework can be used to reason about parametricity predicates. Concretely, we consider a toy example where we implement boolean disjunction in terms of conjunction and negation, for booleans encoded as the natural numbers 0 and 1. We use parametricity to show that, if conjunction and negation send valid booleans to valid booleans, then so does disjunction.

We use a mode theory with two modes: the obligatory trivial mode ⋆ and a mode ↑ modeled as a base category with two objects and only one non-trivial morphism between them. A presheaf then consists of an Agda function between two types, where we regard the domain as a predicate over the codomain. We have two modalities forget and Σ from ↑ to ⋆: the former selects the codomain (the type of all values; and thus forgets the predicate) whereas the latter selects the domain (the type of values satisfying the predicate). There is a two-cell π from Σ to forget, which uses the function to get from its domain to its codomain. We have a type EncBool of booleans encoded as natural numbers, modeled as the function Bool → Nat sending false to 0 and true to 1, and we view this as the natural numbers equipped with the predicate 'encodes a valid boolean'.

Prf-Parametricity

$$\frac{C : \mathsf{Ty}\uparrow}{\Xi \vdash \forall[\,\Sigma \mid \texttt{"c"} \in C\,]\ \mathsf{Pred}\ C\ (\mathsf{mod}\langle\ \mathsf{forget}\ \rangle\ \mathsf{var}\ \texttt{"c"}\ \pi)\ @\ \star}$$

Prf-Extent-From

$$\frac{\Xi \vdash \mathsf{Pred}\ (A \Rrightarrow B)\ f\ @\ \star}{\Xi \vdash \forall[\,\mathsf{forget} \mid \texttt{"a"} \in A\,]\ \mathsf{Pred}\ A\ (\mathsf{mod}\langle\ \mathsf{forget}\ \rangle\ \mathsf{svar}\ \texttt{"a"}) \supset \mathsf{Pred}\ B\ (f \circledast \mathsf{mod}\langle\ \mathsf{forget}\ \rangle\ \mathsf{svar}\ \texttt{"a"})\ @\ \star}$$

Fig. 10. Inference rules specific to unary parametricity.

Next, we extend BiSikkel with one proposition former and two inference rules (see Fig. 10 for the inference rules). For every $C : \mathsf{Ty}\uparrow$ and every term $\Gamma \vdash c : \langle\ \mathsf{forget} \mid C\ \rangle\ @\ \star$, we add $\mathsf{Pred}\ C\ c : \mathsf{bProp}\ \Gamma$. The semantics of $\mathsf{Pred}\ C\ c$ is simply that c satisfies the predicate of C. This is expressed by the rule Prf-Parametricity, which states that $\mathsf{Pred}\ C$ is satisfied by all values c of type C that are bound under modality Σ (so in other words, those values that satisfy the predicate associated to C). Following Reynolds's [1983] definition of the logical relation for the function type, we also assert one half of the logical equivalence of $\mathsf{Pred}\ (A \Rrightarrow B)\ f$ and the assertion that $\mathsf{Pred}\ A\ \texttt{"a"}$ implies $\mathsf{Pred}\ B\ (f \circledast \texttt{"a"})$ for all $\texttt{"a"}$, where $\circledast$ is the applicative operator of the modal type (Prf-Extent-From, since the quantification in that rule contains a forget modality, we have to apply modal constructors to the variable $\texttt{"a"}$).[17] Both rules from Fig. 10 have been proven sound in the model.

We add primitive conjunction $\wedge$' and negation $\neg$' operators for EncBool at mode $\uparrow$ to the MSTT part of the BiSikkel instance, with arbitrary behaviour on naturals greater than 1. Using these primitives, we can implement disjunction $\vee$' for EncBool. Making use of the rules from Fig. 10, we can then prove the proposition

$$\forall[\,\mathbb{1} \mid \texttt{"a"} \in \langle\ \mathsf{forget} \mid A\ \rangle\,]\ \forall[\,\mathbb{1} \mid \texttt{"b"} \in \langle\ \mathsf{forget} \mid B\ \rangle\,]\ \mathsf{Pred}\ \mathsf{EncBool}\ (\mathsf{svar}\ \texttt{"a"}) \supset \mathsf{Pred}\ \mathsf{EncBool}\ (\mathsf{svar}\ \texttt{"b"}) \supset \mathsf{Pred}\ \mathsf{EncBool}\ ((\mathsf{mod}\langle\ \mathsf{forget}\ \rangle\ \vee') \circledast \mathsf{svar}\ \texttt{"a"} \circledast \mathsf{svar}\ \texttt{"b"}).$$

We refer to the Agda code for details [Ceulemans et al. 2024a,b]. Extracting this proof, we should learn that the extraction of $\mathsf{mod}\langle\ \mathsf{forget}\ \rangle\ \vee'$, a function on naturals, really does send valid booleans to valid booleans. However, similar to proofs about streams, Agda is too slow to verify that the extracted proof has the expected type (see also Section 5.3).

5 Implementation of μLF in Agda

Now that we know the proof system μLF and how it can be used in practice, we take a look at how it is implemented as an Agda library. As announced in Sections 1 and 3.3, BiSikkel uses an extrinsically typed representation of proofs. The motivation for this choice is discussed in Section 5.1. As a result of this extrinsic typing, it is possible to write invalid BiSikkel proofs. The library therefore comes with a proof checker, presented in Section 5.2. We then continue in Section 5.3 with a discussion how BiSikkel proofs can be extracted to Agda proofs and conclude in Section 5.4 with details on the implementation of substitution and normalization for MSTT.

[17] As mentioned, the rule Prf-Extent-From actually expresses a logical equivalence so the rule where conclusion and premise are swapped can be added to the system as well. However, this is not necessary for the current example.

5.1 Agda Representation of Proofs

Just like in Section 2.1 where we had to consider the Agda representation for MSTT terms, we now have to choose a representation for μLF proofs. Again we can distinguish between an intrinsically and an extrinsically typed encoding, where we regard propositions as types and their proofs as terms. In the intrinsically typed approach, we would have an Agda data type of proofs that is indexed by a proof context and a proposition. The constructors of this type are direct encodings of the proof rules from Fig. 8:

```
data IProof : (Ξ : ProofCtx m) → bProp (to-ctx Ξ) → Set where
  subst : (φ : bProp (to-ctx Ξ „ μ | x ∈ T)) → IProof (Ξ ,lock⟨ μ ⟩) (t ≡ᵇ s)
        → IProof Ξ (φ [ t / x ]bprop) → IProof Ξ (φ [ s / x ]bprop)
  ⊃-intro : IProof (Ξ „ᵇ μ | x ∈ φ) ψ → IProof Ξ (⟨ μ | φ ⟩⊃ ψ)
  ∀-intro : IProof (Ξ „ᵛ μ | x ∈ T) φ → IProof Ξ (∀[ μ | x ∈ T ] φ)
  ∀-elim : IProof Ξ (∀[ μ | x ∈ T ] φ) → (t : Tm (to-ctx Ξ ,lock⟨ μ ⟩) T)
         → IProof Ξ (φ [ t / x ]bprop)
  ...
```

This intrinsically typed approach has advantages discussed before: We can leverage the Agda type checker for verifying that the proofs we write are valid, and we can use Agda holes to interact with the logical framework, as the Agda goal is a μLF judgment that needs to be derived. This means we do not have to implement our own type-checker, substitution or normalization algorithm, and interaction mechanism. We also mentioned the drawbacks: (1) invocations of the conversion rule are not silent, but become explicit invocations of Agda's transport lemma *with* explicit equality proof, and (2) we noticed poor performance for intrinsic dependent types in a modal setting.

We clarify this latter point here. When experimenting with the above encoding, we constructed a term of type IProof to prove the commutativity of natural number addition (so there were even no modal phenomena taking place). However it took Agda more than 30s to type-check this term, leaving us no hope to prove more interesting results in this way. We suspect that the issue is caused by computation taking place in indices. For example the last argument of the constructor subst has a substitution taking place in the proposition index. This should result in a proposition that is comparable to φ, but the implementation of substitution for MSTT is quite intricate [Ceulemans et al. 2024c] and it seems to be the case that Agda does not fully normalize this application of substitution when checking the sub-proof of subst. This results in a large Agda term as index, a problem that gets worse when many IProof constructors are being combined.

Because of the drawbacks of intrinsic typing, BiSikkel adopts an extrinsically typed representation of proofs. In other words, the Agda type of BiSikkel proofs is not indexed by a proof context or a proposition. It is however necessary to mention terms or propositions in some proof constructors (e.g. subst or ∀-intro), and their Agda types are indexed by an MSTT context. Hence we arrive at the following data type of proofs, indexed by an MSTT context describing the MSTT variables but not the proof assumptions that are in scope:

```
data Proof : Ctx m → Set where
  subst : (φ : bProp (Γ „ μ | x ∈ T)) (t1 t2 : Tm (Γ ,lock⟨ μ ⟩) T)
        → Proof (Γ ,lock⟨ μ ⟩) → Proof Γ → Proof Γ
  ⊃-intro : (x : String) → Proof Γ → Proof Γ
  ∀-intro : (μ : Modality n m) (x : String) (T : Ty n) → Proof (Γ „ μ | x ∈ T) → Proof Γ
  ∀-elim : (μ : Modality n m) (φ : bProp Γ) → Proof Γ → (t : Tm (Γ ,lock⟨ μ ⟩) T) → Proof Γ
  ...
```

Notice that we obtain the type signature of the constructors from the ones in IProof by applying to-ctx to all proof contexts and setting Γ = to-ctx Ξ. Moreover, some constructors now require extra annotations that are needed when type-checking a proof. For example, the constructor subst now also requires both sides of the equation we are eliminating to be given as arguments.

5.2 The Proof Checker

Since our proof syntax is now untyped, BiSikkel has a proof checker that, given a proof context Ξ, a proof p and a proposition φ, will verify whether p is really a valid proof of φ in context Ξ. Since the actual type signature of the proof checker is quite complex, we will build it up step by step.

As a first approximation, we might expect the proof checker to take a proof context, proof and proposition as argument, and return a Boolean representing the result of the checking process (success or failure). In other words, we would get the following Agda function.

```
check-proof : (Ξ : ProofCtx m) → Proof (to-ctx Ξ) → bProp (to-ctx Ξ) → Bool
```

Note that this type signature already enforces the invariant that we imposed on the proof judgment $\Xi \vdash \varphi \ @ \ m$ in Section 4.2: in order to check whether there might be a proof of φ in a proof context Ξ, it must already be the case that φ is a valid proposition in to-ctx Ξ.

If we look back at Fig. 1b depicting BiSikkel's architecture, we see that there is a soundness proof of the system μLF. This is actually part of the proof checker, which is sound by construction. By that we mean that whenever the proof checker declares a proof to be valid, it needs to provide evidence by outputting a semantic term of the semantic type corresponding to the proposition we are trying to prove. More concretely, BiSikkel provides functions that interpret proof contexts and bProps as semantic contexts and semantic types in a presheaf model (recall that these are already dependently typed, which explains why a semantic type depends on a semantic context).

```
⟦_⟧pctx : ProofCtx m → SemCtx ⟦ m ⟧mode
⟦_⟧bprop : bProp Γ → SemTy ⟦ Γ ⟧ctx
to-ctx-subst : (Ξ : ProofCtx m) → (⟦ Ξ ⟧pctx ⇒ ⟦ to-ctx Ξ ⟧ctx)
```

Given a proof context Ξ and a proposition φ, we want the proof checker to return upon success a semantic term in the semantic context ⟦ Ξ ⟧pctx of the semantic type ⟦ φ ⟧bprop. However, the latter semantic type lives in the semantic context ⟦ to-ctx Ξ ⟧ctx. To bridge this gap, BiSikkel implements a semantic substitution to-ctx-subst Ξ that goes from ⟦ Ξ ⟧pctx to ⟦ to-ctx Ξ ⟧ctx for any Ξ and that can be applied to ⟦ φ ⟧bprop to obtain a semantic type that lives in the right context. To deal with potential failure of the proof checking process, we also introduce the proof checking monad PCM.

```
data PCM (A : Set) : Set where
  ok : A → PCM A
  error : String → PCM A
```

This is essentially an error monad with strings as error messages. We now get the following approximation of the type signature for the proof checker.

```
check-proof : (Ξ : ProofCtx m) → Proof (to-ctx Ξ) → (φ : bProp (to-ctx Ξ))
                → PCM (SemTm ⟦ Ξ ⟧pctx (⟦ φ ⟧bprop M.[ to-ctx-subst Ξ ]))
```

Here the operation _M.[_] applies a substitution to a type at the semantic level (the prefix M stands for model and serves to disambiguate syntactic and semantic operations).

To get to the final type signature of check-proof, recall from Section 5.1 that a disadvantage of the extrinsically typed approach is the fact that we cannot use Agda's interactive development features to write partial proofs and inspect goals. This makes it difficult to write even simple proofs.

```
record Goal : Set where
  constructor goal
  field
    gl-identifier : String
    gl-mode : Mode
    gl-ctx : ProofCtx gl-mode
    gl-prop : bProp (to-ctx gl-ctx)
```

(a) Agda definition of goals.

```
SemGoals : List Goal → Set
SemGoals [] = ⊤
SemGoals (goal _ _ Ξ φ :: gls) =
  SemTm ⟦ Ξ ⟧pctx (⟦ φ ⟧bprop M.[ to-ctx-subst Ξ ]) × SemGoals gls

record PCResult (Ξ : ProofCtx m) (φ : bProp (to-ctx Ξ)) : Set where
  field
    goals : List Goal
    denotation : SemGoals goals
      → SemTm ⟦ Ξ ⟧pctx (⟦ φ ⟧bprop M.[ to-ctx-subst Ξ ])
```

(b) The result type of a successful run of the proof checker.[18]

Fig. 11. BiSikkel's support for holes in proofs.

For this reason, we allow holes in BiSikkel proofs via a special constructor hole : String → Proof Γ that receives an identifier for the hole as argument. When reaching such a hole during proof checking, we do not immediately raise an error but record the proof state (i.e. the proof context and proposition to prove) and at the end we provide a list of all the goals that were encountered. Figure 11a shows the definition of a record type Goal that describes all the information that should be recorded when reaching a hole. Upon success, the proof checker will return an Agda value of the record type PCResult implemented in Fig. 11b. As already mentioned, the result will contain a list of goals. Furthermore, there should still be semantic evidence of the proof checker's decision, but this time it will be conditional: we can give a semantic term of the proposition we are trying to prove provided that we are given semantic terms corresponding to all holes that were encountered in the proof. This last part is taken care of by the dependent Agda type SemGoals goals, which just collects all semantic evidence for the goals. Finally, the actual type signature of BiSikkel's proof checker is as follows.

```
check-proof : (Ξ : ProofCtx m) → Proof (to-ctx Ξ) → (φ : bProp (to-ctx Ξ))
              → PCM (PCResult Ξ φ)
```

5.3 Proof Extraction

As we saw in the previous section, BiSikkel's proof checker provides semantic evidence whenever it declares a proof to be valid. This evidence consists of a semantic term in the internally constructed presheaf model. In order to interact well with Sikkel's extraction mechanism, we would like to be able to extract not just terms but also proofs at mode ★.

To this end, we extend Ceulemans et al.'s extraction mechanism [2022] (Section 2.3) as outlined in Fig. 12. In order to extract propositions, which may contain program variables bound by ∀[_|_∈_]_, we need to be able to extract terms in a non-empty context (this was not needed in Sikkel). Therefore, we introduce a type class on syntactic contexts Γ at the trivial mode ★, an instance of which should provide an Agda type extract-ctx Γ and an Agda isomorphism between this type and Γ's denotation in the presheaf model. The story for types is similar, and a term can then be extracted as a function between its context's and type's extractions. Propositions are extracted as Agda type families over their context's extraction, so effectively as (potentially proof-relevant) predicates. Again, an instance should also provide a dependent isomorphism between this type

[18]Here ⊤ is Agda's unit type, containing no information, and × is the Agda type constructor for (non-dependent) product types. Contrary to Section 3, :: here is the Agda constructor for lists.

family and the proposition's denotation. Finally, there is a function that extracts a proof of an extractable proposition in an extractable proof context to the correct dependent Agda function. In Fig. 12, we only show the version for the empty proof context.

Returning to the example from Section 4.3, we note that MSTT streams of type Stream' Nat' are interpreted in the presheaf model as dependent Agda functions of type $\forall\ n \to \mathsf{Vec}\ \mathbb{N}\ (\mathsf{suc}\ n)$ (together with a coherence condition). Consequently, the evidence produced by the proof checker for a valid equality proof of such streams consists of an equality proof of the corresponding vector-producing Agda functions (the implementation of the proof checker makes use of function extensionality). We can provide an extractability instance for Stream' Nat', extracting this type as the type Stream $\mathbb{N}$ of standard Agda streams. In order to prove one of the required isomorphism equalities, we need an axiom conflating bisimilarity and equality of such Agda streams (this axiom is not used anywhere else in the BiSikkel code).

$$
\begin{array}{rcl}
\Gamma : \mathsf{Ctx}\ \star & \rightsquigarrow & \mathsf{extract\text{-}ctx}\ \Gamma : \mathsf{Set} \\
T : \mathsf{Ty}\ \star & \rightsquigarrow & \mathsf{extract\text{-}ty}\ T : \mathsf{Set} \\
t : \mathsf{Tm}\ \Gamma\ T & \rightsquigarrow & \mathsf{extract\text{-}tm}\ t : \mathsf{extract\text{-}ctx}\ \Gamma \to \mathsf{extract\text{-}ty}\ T \\
\varphi : \mathsf{bProp}\ \Gamma & \rightsquigarrow & \mathsf{extract\text{-}bprop}\ \varphi : \mathsf{extract\text{-}ctx}\ \Gamma \to \mathsf{Set} \\
\Xi : \mathsf{ProofCtx}\ \star & \rightsquigarrow & \mathsf{extract\text{-}pfctx}\ \Xi : \mathsf{Set} \\
p : \mathsf{Proof}\ \diamond & \rightsquigarrow & \mathsf{extract\text{-}pf\text{-}\diamond}\ p\ \varphi : \mathsf{extract\text{-}bprop}\ \varphi\ \_
\end{array}
$$

Fig. 12. BiSikkel extraction signatures (only applicable to contexts, types, ... which are indeed extractable). Note that proofs can also be extracted in other proof contexts than $\diamond$, but this is not shown here.

It is now theoretically possible to extract the equivalence proof of iterate and iterate' from Section 4.3 and obtain an equivalence proof of the extracted Agda functions. However, in practice this does not work due to performance of Agda type checking (it is possible to extract the proof, but not to verify that the Agda type of the extracted proof matches the type we expect). Complete extraction of proofs is possible for smaller examples; we extracted a proof of commutativity of natural number addition (implemented in BiSikkel) as an example.

5.4 Substitution and (Fueled) Normalization for MSTT

As we have seen in Section 4, our proof system critically depends on substitution and normalization for MSTT. These operations are not needed in Sikkel [Ceulemans et al. 2022], which does not have a computational theory at the syntactic level. Since the implementation of both procedures is quite intricate, we spend some time in this section to look at the details.

Substitution. In a non-modal type theory, a substitution from a context Γ to a context Δ basically assigns to every variable of type T in Δ a term of type T in Γ. When modalities are added to the story, this situation gets considerably more complicated. Moreover, we want substitution for terms to be an operation on the syntax that computes, unlike MTT [Gratzer 2022] where there is a constructor for applying a substitution to a term (i.e. MTT uses explicit substitutions). In other words, we want to construct a substitution algorithm for MSTT. Such an algorithmic description of substitution is described for MTT in [Ceulemans et al. 2024c]. In BiSikkel, we provide the first implementation of this algorithm for a the simply typed fragment MSTT of MTT. This means that just like Ceulemans et al., we first define the action of renamings on terms (which in a modal-setting includes more than just renaming, e.g. also applying a two-cell to a term) and then the action of substitutions. Moreover, we use the same representation of renamings and substitutions, including the distinction between atomic and regular renamings and substitutions.

Furthermore, we also prove soundness of our substitution algorithm with respect to the presheaf model of MSTT. More concretely, we show that $[\![\ t\ [\ \sigma\ ]\mathsf{tm}\ ]\!]\mathsf{tm}$ and $[\![\ t\ ]\!]\mathsf{tm}\ [\ [\![\ \sigma\ ]\!]\mathsf{subst}\ ]$ are equal

in the model, where $[\![\ \sigma\ ]\!]$subst is the denotation of σ as a presheaf morphism. This proof is used multiple times in the soundness proof for BiSikkel's proof checker.

Fueled Normalization. As mentioned in Section 4.2, the proof checker makes use of a decision procedure for β-equivalence of terms. This amounts to the implementation of a β-normalization function for MSTT in Agda. However, convincing Agda that such a function terminates is highly non-trivial. Although normalization for MTT has been proven on paper in [Gratzer 2022] and implemented for certain mode theories [Stassen et al. 2023], in BiSikkel we opt for a fueled normalization function that takes an extra natural number as argument which will decrease in non-structurally recursive calls. The normalizer will then fail to produce a result when it has no fuel left.

Recall that the proof checker is sound by construction, providing semantic evidence whenever it validates a proof. Since we want to integrate the normalizer into the proof checker, it too should produce some evidence that its result is semantically equal to its argument. We pack the normalized term and the semantic preservation proof in a record.

```
record NormalizeResult (t : Tm Γ T) : Set where
  field
    nt : Tm Γ T
    pres-sem : [[ t ]]tm ≅tm [[ nt ]]tm
```

The normalizer then has the following Agda type.

```
normalize : ℕ → (t : Tm Γ T) → Maybe (NormalizeResult t)
```

Normalization for e.g. function redexes is implemented in terms of the substitution operations defined above. This is in contrast to for example mitten [Stassen et al. 2023], which employs normalization by evaluation.

6 Conclusion, Related Work and Future Work

6.1 Universal Algebra and Formalizations of Type Theory

Sikkel and BiSikkel are based on a deep embedding: their syntax is defined inductively and interpreted in the presheaf model. Such deep embeddings can be understood categorically via algebraic theories. The simplest form of algebraic theories are un(i)sorted ones, examples of which are the theories of groups, monoids, real vector spaces, ... These are specified by a set of operations, each with an arity, and a number of axioms that the operations need to satisfy. A record containing the above data is also called a container [Abbott et al. 2005]. An algebra or model of the theory (e.g. a group, monoid, or real vector space) is then a carrier (a set) on which the operations are defined and the axioms are satisfied. The syntax of the theory is the initial such algebra and its carrier is the set of expressions freely generated by the operations and quotiented out by the axioms.[19] This carrier can be regarded as a quotient inductive type (QIT) and initiality exactly expresses there is a unique, recursively defined interpretation function from the syntax to any model of the theory. Richer notions of algebraic theories include multisorted algebraic theories corresponding to indexed containers [Altenkirch and Morris 2009] and whose syntax is an indexed QIT; second-order algebraic theories which support abstraction [Allais et al. 2021; Fiore and Szamozvancev 2022]; and generalized algebraic theories (GATs) [Cartmell 1986] whose syntax is a quotient-inductive-inductive type (QIIT) [Altenkirch et al. 2018].

Type theory has been formalized within type theory using QIITs [Altenkirch and Kaposi 2016]. Such QIIT formalization is necessarily intrinsically typed, and as mentioned before (Section 3.3),

[19] In the above examples, one would need to extend the theory with constants (nullary operations) or metavariables in order to obtain anything that is not the neutral element.

practical programming is not feasible in such a system. Nevertheless, it would be interesting to maximally categorify (Bi)Sikkel. To this end, we would formalize either MSTT with μLF or MTT as an algebraic theory, obtaining in this way an intrinsically typed syntax as well as an abstract notion of model. This intrinsically typed syntax could then be inserted as an additional layer in the (Bi)Sikkel architecture, between the extrinsic syntax and the semantic layer. The type-checker (which would still rely on fueled normalization) would output intrinsically typed terms instead of semantic ones, and an interpretation of the intrinsically typed syntax in any model – including but not limited to the presheaf model – would be obtained by initiality.

6.2 Integration with Existing Category Libraries

Currently, (Bi)Sikkel relies on its own ad hoc implementation of the relevant categorical concepts, as the original Sikkel code predates the agda-categories library [Hu and Carette 2021]. There is an alternative formalization for category theory in the Agda cubical library [Agda Community 2024; Vezzosi et al. 2021], but the falsehood of uniqueness of identity proofs (UIP) would significantly complicate the construction of the presheaf model. Integration with a mature library for category theory would ease the use of categorical ideas and thus simplify further development of (Bi)Sikkel.

6.3 Equality in the Metatheory

In this section, we first discuss a number of possible metatheories and formalization paradigms in which one could implement a project such as (Bi)Sikkel, with their advantages and drawbacks, all related to equality. A variant of XTT that could be reasonably implemented in Agda, would suit our needs entirely. Afterward, we explain how (Bi)Sikkel strikes a balance between the different options, avoiding as many drawbacks as we were able to, but resulting in the near-impossibility to implement a semantic universe.[20]

Extensional Type Theory (ETT). Extensional type theory features the reflection rule, which promotes propositional to judgmental equality. Consequences are function extensionality and uniqueness of identity proofs. In this system, we would have no problems whatsoever; however since extensional equality is undecidable [Hofmann 1995], ETT cannot be supported by a proof assistant such as Agda.

Intensional Type Theory (ITT). In intensional type theory, function extensionality is not provable but can be soundly postulated as an axiom. A naïve presheaf model of type theory in which equality in the object language is modeled as propositional equality, would rely on function extensionality. This is especially problematic in the case of type equality, where transport along equality models the conversion rule, while its computation may be blocked by function extensionality. This may ultimately block extracted (Bi)Sikkel programs. If we enable axiom K [Cockx et al. 2014], then we have computational UIP. It is worth noting that Lean has UIP and quotient types (implying function extensionality) by default, and while these can block computation at type-checking time, transport along them would be compiled away [Avigad et al. 2017, ch. 11].

Setoid Hell in ITT. Setoid hell is the ubiquitous usage of setoids and is, as the term suggests, cumbersome: one needs to provide, for every type concerned, an equivalence relation and prove, for every function one implements, that it respects equivalence. Moreover, if we want UIP for our custom equivalence, this leads to an additional axiom in the definition of a setoid. On the bright side, function extensionality is for free, since we can take it as the definition of function equivalence. Modeling a universe in setoid hell is difficult, as it is unclear how to define equivalence of types (or

[20]Note that, despite the limited dependency available in μLF, we could imagine a universe of propositions Prop' at the program level and a predicate El' over it.

even setoids) depending on elements of other setoids. Altenkirch et al. [2021] build an inductive universe instead, which is laborious especially as part of a presheaf model.

Cubical Agda. Cubical Agda [Vezzosi et al. 2021] is an implementation of cubical type theory [Cohen et al. 2018], itself a computational variant of homotopy type theory (HoTT) [Univalent Foundations Program 2013], in which function extensionality is provable and UIP is false. In this setting, it would be sensible to model (Bi)Sikkel in h-set-valued presheaves. However, the type of h-set-valued presheaves is not an h-set but an h-groupoid, which precludes the construction of a universe of types based on cubical Agda's universe. Again, we could build an inductive universe instead.

XTT. XTT [Sterling et al. 2019] uses cubical techniques to strike a balance between ITT and ETT. It is a decidable type system featuring definitional UIP and provable function extensionality. We believe that XTT would suit our needs, but to our knowledge it currently has no mature implementation. There are open feature requests to provide a version of Cubical Agda without univalence and with computational non-definitional UIP[21], which would suit our needs equally well.

Observational Type Theory (OTT). Observational type theory [Altenkirch et al. 2007] is a system where the equality type of any concrete type T reduces to its characterization, e.g. equality at $A \times B$ reduces to the product of equality at A and at B. In this system, function extensionality is proven by the identity function and UIP could reduce in a similar fashion as the identity type itself. This system, too, would suit our needs, but to our knowledge it currently has no mature implementation.

BiSikkel. BiSikkel is currently implemented in ITT: partially in plain ITT, partially in setoid hell and partially in groupoid hell. Specifically, our presheaves are set-valued (not setoid or groupoid-valued). Context and type equality are modeled as isomorphism of (dependent) presheaves (a groupoid structure), because this yields better computational behaviour than the inductive identity type which only computes in trivial cases. Term equality is modeled as pointwise equality (a setoid structure) rather than propositional equality, which saves an invocation of function extensionality. Term equality could not be modeled as pointwise equivalence in some sense, as the presheaf that models the type is set-valued and does not specify a notion of equivalence. By consequence, congruence of the abstraction rule does rely on function extensionality. The fact that the collection of types constitutes a groupoid while the cells of a type form a set, rules out the most sensible implementations of the universe.[22]

6.4 Reflection

It would be practical if the syntax of a subset of a future version of Agda with better modal support, could double as an intrinsically typed syntax of BiSikkel. One could use reflection techniques to quote the Agda syntax and analyze the abstract syntax tree. The proof assistant Coq has a rich tradition of Coq-to-Coq translations. In particular, Jaber et al. [2012] interpret non-modal Coq syntax in an internal presheaf model different from Hofmann's [1997], and Jaber et al. [2016] even manage to preserve conversion. This alternative presheaf model has not, to our knowledge, been studied as a model of multimode type theory.

Data Availability Statement

We provide a virtual machine on which the BiSikkel library, as well as compatible versions of Agda and its standard library, have been installed [Ceulemans et al. 2024b]. All code related to the

[21] https://github.com/agda/agda/issues/3750 and https://github.com/agda/agda/issues/6696

[22] More precisely, we cannot support a type encoding rule (for converting types to universe terms), nor can we allow the decoding rule to reflect judgmental equality. Nothing prevents us, of course, from extending BiSikkel with an inductive universe, just as we can extend BiSikkel with other inductive types.

examples in this paper is also included. Furthermore, the BiSikkel library and code for the examples is also available on GitHub [Ceulemans et al. 2024a].

Acknowledgments

Joris Ceulemans held a PhD fellowship (1184122N) of the Research Foundation – Flanders (FWO) while working on this research. Andreas Nuyts holds a Postdoctoral fellowship (1247922N and 12AB225N) of the Research Foundation – Flanders (FWO). This research is partially funded by the Research Fund and Internal Funds KU Leuven, and by the Research Foundation – Flanders (FWO, G0G0519N and G030320N).

References

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Andreas Abel. 2006. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* Ph. D. Dissertation. Ludwig-Maximilians-Universität München.

Andreas Abel. 2008. Polarised Subtyping for Sized Types. *Mathematical Structures in Computer Science* 18, 5 (2008), 797–822. https://doi.org/10.1017/S0960129508006853

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017a. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (dec 2017), 29 pages. https://doi.org/10.1145/3158111

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* Volume 8, Issue 1 (March 2012), 1–36. https://doi.org/10.2168/LMCS-8(1:29)2012

Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017b. Normalization by Evaluation for Sized Dependent Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 33 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110277

The Agda Community. 2024. A Standard Library for Cubical Agda. https://github.com/agda/cubical

The Agda Development Team. 2024. *Agda.* https://wiki.portal.chalmers.se/agda

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A Type- and Scope-safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *J. Funct. Program.* 31 (2021), e22. https://doi.org/10.1017/S0956796820000076

Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, Christian Sattler, and Filippo Sestini. 2021. Constructing a Universe for the Setoid Model. In *Foundations of Software Science and Computation Structures*, Stefan Kiefer and Christine Tasson (Eds.). Springer International Publishing, Cham, 1–21. https://doi.org/10.1007/978-3-030-71995-1_1

Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. 2018. Quotient Inductive-Inductive Types. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 293–310. https://doi.org/10.1007/978-3-319-89366-2_16

Thorsten Altenkirch and Ambrus Kaposi. 2016. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 18–29. https://doi.org/10.1145/2837614.2837638

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational Equality, Now!. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification* (Freiburg, Germany) *(PLPV '07)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/1292597.1292608

Thorsten Altenkirch and Peter Morris. 2009. Indexed Containers. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, Washington D.C., 277–285. https://doi.org/10.1109/LICS.2009.33

Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/2500365.2500597

Jeremy Avigad, Leonardo de Moura, and Soonho Kong. 2017. *Theorem Proving in Lean.* https://leanprover.github.io/theorem_proving_in_lean/index.html

Bruno Barras and Bruno Bernardo. 2008. *The Implicit Calculus of Constructions as a Programming Language with Dependent Types.* Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379. https://doi.org/10.1007/978-3-540-78499-9_26

Viktor Bense, Ambrus Kaposi, and Szumi Xie. 2024. Strict Syntax of Type Theory via Alpha-normalisation. In *30th International Conference on Types for Proofs and Programs (TYPES)*. 65–67. https://types2024.itu.dk/abstracts.pdf#page=75

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2020. Modal Dependent Type Theory and Dependent Right Adjoints. *Mathematical Structures in Computer Science* 30, 2 (2020), 118–138. https://doi.org/10.1017/S0960129519000197

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First Steps in Synthetic Guarded Domain Theory: Step-indexing in the Topos of Trees. *Logical Methods in Computer Science* Volume 8, Issue 4 (Oct. 2012), 45 pages. https://doi.org/10.2168/LMCS-8(4:1)2012

John Cartmell. 1986. Generalised Algebraic Theories and Contextual Categories. *Ann. Pure Appl. Log.* 32 (1986), 209–243. https://doi.org/10.1016/0168-0072(86)90053-9

Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2022. Sikkel: Multimode Simple Type Theory as an Agda Library. In *Proceedings Ninth Workshop on Mathematically Structured Functional Programming (Electronic Proceedings in Theoretical Computer Science, Vol. 360)*, Jeremy Gibbons and Max S. New (Eds.). Open Publishing Association, Munich, Germany, 2nd April 2022, 93–112. https://doi.org/10.4204/EPTCS.360.5

Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2024a. BiSikkel. https://github.com/JorisCeulemans/bisikkel.

Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2024b. *BiSikkel (a multimode logical framework in Agda) VM.* https://doi.org/10.5281/zenodo.13939916

Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2024c. A Sound and Complete Substitution Algorithm for Multimode Type Theory. In *29th International Conference on Types for Proofs and Programs (TYPES 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 303)*, Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:23. https://doi.org/10.4230/LIPIcs.TYPES.2023.4

James Chapman. 2009. Type Theory Should Eat Itself. *Electronic Notes in Theoretical Computer Science* 228 (2009), 21–36. https://doi.org/10.1016/j.entcs.2008.12.114 Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2017. The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types. *Logical Methods in Computer Science* Volume 12, Issue 3 (April 2017), 39 pages. https://doi.org/10.2168/LMCS-12(3:7)2016

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern Matching without K. *SIGPLAN Not.* 49, 9 (aug 2014), 257–268. https://doi.org/10.1145/2692915.2628139

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:34. https://doi.org/10.4230/LIPIcs.TYPES.2015.5

Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal Metatheory of Second-order Abstract Syntax. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498715

Daniel Gratzer. 2022. Normalization for Multimodal Type Theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science* (Haifa, Israel) *(LICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. https://doi.org/10.1145/3531130.3532398

Daniel Gratzer and Lars Birkedal. 2022. A Stratified Approach to Löb Induction. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:22. https://doi.org/10.4230/LIPIcs.FSCD.2022.23

Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2021. Multimodal Dependent Type Theory. *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021), 67 pages. https://doi.org/10.46298/lmcs-17(3:11)2021

Adrien Guatto. 2018. A Generalized Modality for Recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 482–491. https://doi.org/10.1145/3209108.3209148

Martin Hofmann. 1995. *Extensional Concepts in Intensional Type Theory.* Ph. D. Dissertation. University of Edinburgh. College of Science and Engineering. https://era.ed.ac.uk/handle/1842/399

Martin Hofmann. 1997. *Syntax and Semantics of Dependent Types.* Cambridge University Press, Cambridge, Chapter 4, 79–130.

Jason Z. S. Hu and Jacques Carette. 2021. Formalizing Category Theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 327–342. https://doi.org/10.1145/3437992.3439922

Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. 2016. The Definitional Side of the Forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 367–376. https://doi.org/10.1145/2933575.2935320

Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. 2012. Extending Type Theory with Forcing. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)* (Dubrovnik, Croatia). IEEE Computer Society Press, Washington D.C., 395–404. https://doi.org/10.1109/LICS.2012.49

Daniel R. Licata and Robert Harper. 2011. 2-Dimensional Directed Type Theory. *Electr. Notes Theor. Comput. Sci.* 276 (2011), 263–289. https://doi.org/10.1016/j.entcs.2011.09.026

Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 108)*, Hélène Kirchner (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.22

Daniel R. Licata and Michael Shulman. 2016. Adjoint Logic with a 2-Category of Modes. In *Logical Foundations of Computer Science - International Symposium, LFCS 2016, Deerfield Beach, FL, USA, January 4-7, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9537)*, Sergei N. Artëmov and Anil Nerode (Eds.). Springer, Cham, 219–235. https://doi.org/10.1007/978-3-319-27683-0_16

Alexandre Miquel. 2001. The Implicit Calculus of Constructions. In *Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer, Berlin, Heidelberg, 344–359. https://doi.org/10.1007/3-540-45413-6_27

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer, Berlin, Heidelberg, 350–364. https://doi.org/10.1007/978-3-540-78499-9_25

Hiroshi Nakano. 2000. A Modality for Recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science*. IEEE, Washington D.C., 255–266. https://doi.org/10.1109/LICS.2000.855774

nLab authors. 2024. Functoriality of Categories of Presheaves. https://ncatlab.org/nlab/show/functoriality+of+categories+of+presheaves. Revision 7.

Paige Randall North. 2018. Towards a Directed Homotopy Type Theory. *CoRR* abs/1807.10566 (2018), 17 pages. arXiv:1807.10566 http://arxiv.org/abs/1807.10566

Andreas Nuyts. 2023a. Higher Pro-arrows: Towards a Model for Naturality Pretype Theory. In *Workshop on Homotopy Type Theory / Univalent Foundations*. 4 pages. https://hott-uf.github.io/2023/HoTTUF_2023_paper_1410.pdf

Andreas Nuyts. 2023b. A Lock Calculus for Multimode Type Theory. In *29th International Conference on Types for Proofs and Programs (TYPES)*. 3 pages. https://lirias.kuleuven.be/retrieve/720873

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 779–788. https://doi.org/10.1145/3209108.3209119

Andreas Nuyts and Dominique Devriese. 2019. Menkar: Towards a Multimode Presheaf Proof Assistant. In *TYPES*. 4 pages. https://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_33

Andreas Nuyts and Dominique Devriese. 2024. Transpension: The Right Adjoint to the Pi-type. *Logical Methods in Computer Science* Volume 20, Issue 2 (June 2024), 54 pages. https://doi.org/10.46298/lmcs-20(2:16)2024

Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric Quantifiers for Dependent Type Theory. *Proc. ACM Program. Lang.* 1, ICFP, Article 32 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110276

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *LICS '01*. IEEE, Washington D.C., 221–230. https://doi.org/10.1109/LICS.2001.932499

Frank Pfenning and Rowan Davies. 2001. A Judgmental Reconstruction of Modal Logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. https://doi.org/10.1017/S0960129501003322

Josselin Poiret, Lucas Escot, Joris Ceulemans, Malin Altenmüller, and Andreas Nuyts. 2023. Read the Mode and Stay Positive. In *29th International Conference on Types for Proofs and Programs (TYPES)*. 3 pages. https://lirias.kuleuven.be/retrieve/720869

Jason Reed. 2003. Extending Higher-Order Unification to Support Proof Irrelevance. In *Theorem Proving in Higher Order Logics*, David Basin and Burkhart Wolff (Eds.). Springer, Berlin, Heidelberg, 238–252. https://doi.org/10.1007/10930755_16

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, Amsterdam, 513–523.

Michael Shulman. 2023. Semantics of Multimodal Adjoint Type Theory. *Electronic Notes in Theoretical Informatics and Computer Science* Volume 3 - Proceedings of MFPS XXXIX (Nov. 2023), 19 pages. https://doi.org/10.46298/entics.12300

Philipp Stassen, Daniel Gratzer, and Lars Birkedal. 2023. mitten: A Flexible Multimodal Proof Assistant. In *28th International Conference on Types for Proofs and Programs (TYPES 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 269)*, Delia Kesner and Pierre-Marie Pédrot (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:23. https://doi.org/10.4230/LIPIcs.TYPES.2022.6

Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2019. Cubical Syntax for Reflection-Free Extensional Equality. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:25. https://doi.org/10.4230/LIPIcs.FSCD.2019.31

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study.

Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing π-Calculus in Guarded Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 270–283. https://doi.org/10.1145/3372885.3373814

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *J. Funct. Program.* 31 (2021), e8. https://doi.org/10.1017/S0956796821000034

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