Sikkel: Multimode Simple Type Theory as an Agda Library

Joris Ceulemans Andreas Nuyts Dominique Devriese
Department of Computer Science, KU Leuven

Many variants of type theory extend a basic theory with additional primitives or properties like univalence, guarded recursion or parametricity, to enable constructions or proofs that would be harder or impossible to do in the original theory. However, implementing such extended type theories (either from scratch or by modifying an existing implementation) is a big hurdle for their wider adoption. In this paper we present Sikkel, a library in the dependently typed programming language Agda that allows users to program in extended type theories. It uses a deeply embedded language that can be easily extended with additional type and term constructors, thus supporting a wide variety of type theories. Moreover, Sikkel has a type checker that is sound by construction in the sense that all well-typed programs are automatically translated to their semantics in a shallow embedding based on presheaf models. Additionally, our model supports combining different base categories by using modalities to transport definitions between them. This enables in particular a general approach for extracting definitions to the meta-level, so that we can use the extended type theories to define regular Agda functions and prove properties of them. In this paper, we demonstrate Sikkel theories with guarded recursion and parametricity, but other extensions can be easily plugged in. For now, Sikkel supports only simple type theories but its model already anticipates the future addition of dependent types and a universe.

1 Introduction

Dependently typed programming languages like Agda or Coq are based on mathematical formal systems such as Martin-Löf Type Theory (MLTT [27]) or the Calculus of Inductive Constructions (CIC [17, 18]). Over the past years many authors have proposed new type systems that extend a theory like MLTT or CIC with new primitives which allow users to prove more theorems or write more programs. Examples of such extensions include guarded recursion [10, 20], parametricity [1, 2, 4, 33, 30, 11], univalence [6, 12, 15], directed type theory [36, 41] and nominal reasoning [34].

However, the question remains how these extended formal systems can be used in existing languages like Agda or Coq. In this paper we present Sikkel[1], an Agda library that allows users to work in a class of extended type theories called multimode or multimodal type theories [26, 20]. Such type theories are parametrized by a mode theory which specifies new primitive type constructors called modalities.

The Sikkel library consists of 3 layers, depicted in Figure 1. The first is an extrinsically typed syntax of multimode simple type theory (MSTT) defined in Agda. Once a mode theory has been fixed, a user can write programs in a modal setting at this layer. Moreover, the syntax can be easily extended with additional (non-modal) type and term formers.

Programs written at the first layer cannot be directly interpreted as Agda programs because that would require Agda interpretations for the newly added primitives. Since our goal is to work within off-the-shelf Agda, this is impossible for many of the features we want to support. However, many extensions of type theory (including all examples mentioned above) can be interpreted in a class of mathematical models called presheaf models. Sikkel's second, semantic layer therefore consists of a formalization of

[1] Available at `https://github.com/JorisCeulemans/sikkel/releases/tag/v1.0`.

Jeremy Gibbons and Max S. New (Eds.): Ninth Workshop on
Mathematically Structured Functional Programming (MSFP 2022)
EPTCS 360, 2022, pp. 93–112, doi:10.4204/EPTCS.360.5

presheaf models in Agda, essentially a shallow embedding of MSTT [42]. The bridge between the first and the second layer is formed by a type checker, implemented in Agda, for the deeply embedded syntax. It is sound by construction in the sense that it will either refute a judgment or accept it *and* interpret it in the presheaf model.

However, this is insufficient if we want to incorporate programs defined in Sikkel into an "ordinary" Agda project. For that reason, Sikkel considers a third layer: plain Agda, and provides an extraction mechanism that translates terms at the semantic layer to ordinary Agda terms. We only support this for terms in the trivial mode (modeled in the category of sets), but modalities allow us to carry over terms from other modes (modeled in other presheaf categories) to the trivial mode, whence they can be extracted to Agda.

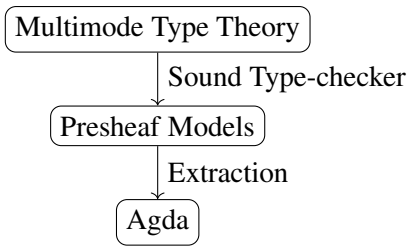


Figure 1: Sikkel's architecture.

We demonstrate the use of Sikkel in two applications: guarded recursive type theory and a restricted form of parametricity. With guarded recursion, we can write definitions of infinite streams that would not be accepted by Agda's termination checker. The extraction mechanism allows us to interpret these definitions as ordinary Agda streams. For parametricity, we demonstrate how a function definition can be interpreted with two different representations for an abstract type, and how parametricity allows us to relate the two resulting definitions.

Sikkel is implemented in a standard off-the-shelf version of Agda (in which uniqueness of identity proofs is by default enabled). We postulate function extensionality but do not use other postulates or unsafe Agda features like pragmas or sized types [16], which could (potentially) break soundness of the meta-theory. Consequently, our library should also be implementable in other dependently typed languages than Agda.

Contributions Our central contribution is Sikkel, an Agda library for multimode simple type theory (MSTT) based on presheaf models. It combines some novel and some existing ideas to obtain a system with novel characteristics:

- Sikkel's semantic layer is fully general w.r.t. the base category. As such, it can in principle represent many different type theory extensions like guarded recursion [10, 20], parametricity [1, 2, 4, 33, 30], univalence [6, 12, 15], nominal [34] and directed [36] type theory, etc., as well as combinations of these [11, 41].
- Sikkel is built on multimode type theory, which allows programs to move between modes (presheaf models over different base categories), along modalities (adjunctions between those presheaf models). Thanks to this, the extraction mechanism does not expose the user to extension-specific models (like the topos of trees or reflexive graphs). Essentially, conversion from the richer object theory to the meta-theory is done within Sikkel (using a modality to the trivial mode) rather than outside it.
- Sikkel is easy to extend. The layer 2 model has an open mode theory and modes and modalities can be added simply by implementing an additional base category or a dependent right adjoint on presheaf categories, respectively. In the syntactic layer, the mode theory as well as primitive types and their typing rules can be changed as long as they can be checked in a way that is sound with respect to the Sikkel model.
- To accomodate future Sikkel support for dependent types, the presheaf model already implements the

structural rules of an internal category with families (CwF) [19]: contexts, context-dependent types and terms, equality judgments, and type and term substitution.

- We demonstrate two concrete examples of multimode simple type theories in Sikkel: one with support for guarded recursion, where we are able to implement all examples involving streams that were presented on paper by Clouston et al. [13, pp. 8–12], and one with support for proving representation independence results using parametricity.

Overview of the paper We explain Sikkel's syntactic layer in Section 2, followed by a discussion of guarded recursion in Section 3. This application will be used as a motivating example throughout the rest of the paper. Section 4 presents the semantic layer of Sikkel. The type checker which translates between the first two layers is presented in Section 5 and the extraction from the semantic layer to Agda can be found in Section 6. To demonstrate Sikkel's generality, Section 7 discusses a second application: the use of parametricity to obtain representation independence results. We conclude by discussing the current, related and future work in Section 8.

2 Multimode Simple Type Theory (MSTT)

We first present Sikkel's syntactic layer: Multimode Simple Type Theory (MSTT), which is essentially Multimode Type Theory (MTT) by Gratzer et al. [20] restricted to simple (non-dependent) types. Just like MTT, MSTT is parametrized by a mode theory specifying the available modalities, discussed in Section 2.1. In Section 2.2, we present MSTT's type formers and contexts. Finally, the typing rules are explained in Section 2.3. We note that MSTT does not have an equational theory or evaluation function for terms. This is unnecessary because MSTT is not dependently typed and all computational aspects will be handled by the model after interpretation of the syntax.

```
record ModeTheory : Set₁ where
  field
    ModeExpr : Set
    ModalityExpr : ModeExpr →
      ModeExpr → Set
    _ⓜ_ : ModalityExpr n o →
          ModalityExpr m n →
          ModalityExpr m o
    𝟙 : ModalityExpr m m
```

(a) Fields introduced in Section 2.1.

```
    TwoCellExpr : Set
    id-cell : TwoCellExpr
```

(b) Fields introduced in Section 2.3.

Figure 2: Type signature of the mode theory parameter to Sikkel. More fields are introduced in Figs. 7a, 8b and 8d.

2.1 Mode Theory

In a multimode type theory, every context, type or term lives at a particular mode and the available modes are specified by the mode theory. Moreover, a mode theory fixes for every two modes m and n a collection of modalities going from m to n together with a composition operation for modalities and a unit modality for every mode. As such, a mode theory can be any small category with the objects functioning as modes and the morphisms as modalities.

A Sikkel mode theory consists of an Agda type of modes and a type family of modalities together with the composition operation and unit modality (Fig. 2a).[2] The ModeTheory record actually contains a few other fields that we will introduce in Sections 2.3 and 5. In particular, a mode theory also specifies

[2]Here Set and Set_1 are the first two types in Agda's universe hierarchy. Note that the seemingly unbound variables m, n and o are implicitly universally quantified. _ⓜ_ is Agda syntax for defining a mixfix operator ⓜ. The underscore characters indicate where arguments are expected.

$$\frac{\alpha \in \mu \Rightarrow \mathsf{locks}(\Delta) \qquad x \notin \Delta}{\Gamma\ ,\ \mu \mid x \in T\ ,\Delta \vdash \mathsf{var}\ x\ \alpha : T}\ \textsc{Tm-Var} \qquad \frac{\Gamma \vdash t : T \qquad T \simeq^{\mathsf{ty}} S}{\Gamma \vdash \mathsf{ann}\ t \in S : S}\ \textsc{Tm-Ann}$$

$$\frac{\Gamma\ ,\ \mathbb{1} \mid x \in T \vdash s : S}{\Gamma \vdash \mathsf{lam}[\, x \in T \,]\ s : T \Rrightarrow S}\ \textsc{Tm-Lam} \qquad \frac{\Gamma \vdash f : R \Rrightarrow S \qquad \Gamma \vdash t : T \qquad R \simeq^{\mathsf{ty}} T}{\Gamma \vdash f \cdot t : S}\ \textsc{Tm-App}$$

$$\frac{\Gamma\ ,\mathsf{lock}\langle\ \mu\ \rangle \vdash t : T}{\Gamma \vdash \mathsf{mod}\langle\ \mu\ \rangle\ t : \langle\ \mu \mid T\ \rangle}\ \textsc{Tm-ModIntro} \qquad \frac{\Gamma \vdash t : \langle\ \rho \mid T\ \rangle \qquad \rho \simeq^{\mathsf{m}} \mu \qquad \Gamma\ ,\ \mu \mid x \in T \vdash s : S}{\Gamma \vdash \mathsf{let'}\ \mathsf{mod}\langle\ \mu\ \rangle\ x \leftarrow t\ \mathsf{in'}\ s : S}\ \textsc{Tm-ModElim}$$

Figure 4: Some typing rules of MSTT.

an equivalence relation $\simeq^{\mathsf{m}}$ on ModalityExpr $m\ n$ for every two modes m and n. Associativity and unit laws for ⓜ are expected to hold up to $\simeq^{\mathsf{m}}$ in order for the resulting type theory to be well-behaved, but there are no corresponding proof obligations when constructing a ModeTheory record. In the remainder of this section, we assume a mode theory and field names of the record type will refer to this theory.

2.2 Types and Contexts

We define a type TyExpr of MSTT type expressions as an inductive family indexed by a ModeExpr (Fig. 3). MSTT has built-in types for natural numbers, booleans, functions (⇛) and products (⊠) at any mode. The most interesting type former takes a modality μ from mode m to n and a type T at mode m to produce the modal type $\langle\ \mu \mid T\ \rangle$ at mode n.

The typing rules for MSTT will use an equivalence relation $\simeq^{\mathsf{ty}}$ on types as a form of judgmental equality. This relation $\simeq^{\mathsf{ty}}$ is the smallest congruence on types such that the modal type former respects the relation $\simeq^{\mathsf{m}}$ on modalities.

MSTT contexts also live at a particular mode (Fig. 3). Every mode has an empty context ⋄. Variables in MSTT are represented by strings[3] and are annotated with a modality. As such, we can extend any context Γ at mode m with a variable x of type T (at mode n) under a modality μ from n to m, to obtain a new context $\Gamma\ ,\ \mu \mid x \in T$. Finally, a context Γ at mode n can be locked with a modality μ from mode m to n to produce a new context $\Gamma\ ,\mathsf{lock}\langle\ \mu\ \rangle$ at mode m. This operation will play an important role in the introduction rule for modal types and in the variable rule.

```
data TyExpr : ModeExpr → Set where
  Nat Bool : TyExpr m
  _⇛_ _⊠_ : TyExpr m → TyExpr m → TyExpr m
  ⟨_|_⟩ : ModalityExpr m n → TyExpr m → TyExpr n

data CtxExpr : ModeExpr → Set where
  ⋄ : CtxExpr m
  _,_|_∈_ : CtxExpr m → ModalityExpr n m →
    String → TyExpr n → CtxExpr m
  _,lock⟨_⟩ : CtxExpr n → ModalityExpr m n →
    CtxExpr m
```

Figure 3: Syntax for MSTT types and contexts.

2.3 The Typing Rules

The most interesting typing rules of MSTT can be found in Figure 4. The rules and term formers for constructing and destructing product values, booleans, and natural numbers have been omitted. Note that we are formulating MSTT as an *algorithmic* system for type *inference*, i.e. the typing judgment can be

[3] A de Bruijn approach is inherent to the CwF-structure of the semantic layer. As such, the strings will be resolved to de Bruijn indices during type-checking/interpretation and hence before any equational or computational reasoning can occur (recall that MSTT does not have an equational theory or evaluation function for terms).

seen as taking the context and term as inputs and returning – if the term is well-typed – an inferred type as output. This explains why in some places we reuse a type or modality symbol whereas in other places we use different symbols and check for equivalence. In particular, we never have control over the type inferred by a premise (although of course we can raise a type error if it does not match a certain pattern) so we can only assign it a new symbol and check for equivalence if necessary.

The variable rule Tm-Var mentions a new concept of the mode theory that we have not yet introduced. Apart from modes and modalities, a ModeTheory also specifies an Agda type TwoCellExpr of 2-cell expressions together with a particular so-called trivial 2-cell id-cell (Fig. 2b). Such 2-cells can be seen as morphisms between modalities and they will control when a variable in the context can be used in the construction of a term. We write $\alpha \in \mu \Rightarrow \rho$ to mean that the 2-cell α can have the modality μ as its domain and the modality ρ as its codomain and a ModeTheory will additionally consist of a function to check such statements (Fig. 8b). For the trivial 2-cell, it is required that id-cell $\in \mu \Rightarrow \rho$ for any μ and ρ as long as $\mu \simeq^{\mathsf{m}} \rho$.[4] The MSTT variable rule expresses that we can use a variable x which is in the context under a modality μ, provided that there is a 2-cell going from μ to the composite of all modalities that appear in the locks to the right of x. In many cases, these two modalities are equivalent so Sikkel has the abbreviation svar x = var x id-cell.

The rule Tm-Ann allows to cast a term to an equivalent type. There are standard typing rules for lambda abstraction (Tm-Lam, introducing a variable under the trivial modality $\mathbb{1}$, see further below for details about modal functions) and function application (Tm-App). The modal constructor mod⟨ μ ⟩ allows us to construct a term of the modal type ⟨ μ | T ⟩ if we can produce a term of type T after locking the context with modality μ (Tm-ModIntro). This lock ensures that in both the premise and the conclusion of Tm-ModIntro, context, term and type live at the same mode. Indeed, if μ goes from m to n, then t and T will live at mode m but Γ will live at mode n. The modal elimination rule Tm-ModElim allows us to pattern match in some sense on a term of a modal type.[5] More concretely, it allows to view any term t of type ⟨ μ | T ⟩ as a term of the form mod⟨ μ ⟩ x for some variable x of type T that appears in the context under modality μ. Just like in MTT, the modal eliminator is actually more expressive and may take an additional modality as a parameter, but this will not be needed in the paper.

MSTT has no *primitive* term formers for modal functions, but they are provided as syntactic sugar and implemented using the primitives introduced in Figure 4. More concretely, one can write lam[μ | $x \in T$] s to construct a function of type ⟨ μ | T ⟩ ⇛ S. In this case, when type-checking the term s, the variable x will appear in the context under the modality μ instead of under $\mathbb{1}$ as in Tm-Lam. We can also use modal function application $f \cdot$⟨ μ ⟩ t to apply a modal function f of type ⟨ μ | T ⟩ ⇛ S to a term t of type T. To check such a term in a context Γ, the function f will also be checked in Γ but the argument t will be checked in the locked context Γ ,lock⟨ μ ⟩.[6]

[4] In MTT [20], a mode theory additionally provides 2 composition operations for 2-cells: vertical composition lets us combine $\alpha \in \mu \Rightarrow \rho$ and $\beta \in \kappa \Rightarrow \mu$ to obtain α ⓣ-vert $\beta \in \kappa \Rightarrow \rho$ and horizontal composition allows to combine $\alpha_1 \in \mu_1 \Rightarrow \rho_1$ and $\alpha_2 \in \mu_2 \Rightarrow \rho_2$ to obtain α_1 ⓣ-hor $\alpha_2 \in \mu_1$ ⓜ $\mu_2 \Rightarrow \rho_1$ ⓜ ρ_2 (the latter is of course only possible when the modalities involved can be composed). These operations make an MTT mode theory a strict 2-category instead of just a 1-category, but they are not required when constructing a record of type ModeTheory because they are not necessary in the formulation of the MSTT syntax or type checker. It is however recommendable to provide them to the programmer from a user-friendliness perspective.

[5] The primes in let' and in' avoid conflicts with Agda's keywords let and in.

[6] This is *almost* as good as a native modal function type like in MTT, except that 1) our modal function type of modality $\mathbb{1}$ is not equivalent to the non-modal function type and 2) since η-equality cannot in general be expressed for modal types in MTT, if we were to add an equational theory to MSTT, then we cannot have η-equality for these 'sugary' functions. These equalities however do hold in the presheaf model.

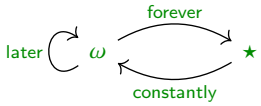
```
record Stream (A : Set) : Set where
  coinductive
  field head : A
        tail : Stream A

zeros : Stream ℕ
head zeros = 0
tail zeros = zeros

map : (A → B) → Stream A → Stream B
head (map f as) = f (head as)
tail (map f as) = map f (tail as)

nats : Stream ℕ
head nats = 0
tail nats = map suc nats
```

(a) Defining and working with coinductive streams in Agda.[7]




```
forever ⓜ later ≃ᵐ forever
forever ⓜ constantly ≃ᵐ 𝟙
𝟙-to-later ∈ 𝟙 ⇒ later
const∘forev-to-𝟙 ∈ (constantly ⓜ forever) ⇒ 𝟙
```

(b) Mode theory for guarded recursion.

```
g-map : TyExpr ★ → TyExpr ★ → TmExpr ω
g-map A B = ann (
  lam[ constantly | "f" ∈ A ⇛ B ]
    löb[later| "m" ∈ GStream A ⇛ GStream B ]
      lam[ "s" ∈ GStream A ]
        let' mod⟨ constantly ⟩ "head-s" ← g-head · svar "s" in'
        let' mod⟨ later ⟩ "tail-s" ← g-tail · svar "s" in'
        g-cons ·⟨ constantly ⟩ (svar "f" · svar "head-s")
               ·⟨ later ⟩ (svar "m" · svar "tail-s")
  ) ∈ (⟨ constantly | A ⇛ B ⟩ ⇛ GStream A ⇛ GStream B)

g-nats : TmExpr ω
g-nats = löb[later| "s" ∈ GStream Nat ]
  g-cons ·⟨ constantly ⟩ lit 0
         ·⟨ later ⟩ (g-map ·⟨ constantly ⟩ suc · svar "s")

Stream' : TyExpr ★ → TyExpr ★
Stream' A = ⟨ forever | GStream A ⟩

cons' : TyExpr ★ → TmExpr ★
cons' A = lam[ "a" ∈ A ] lam[ "as" ∈ Stream' A ]
  let' mod⟨ forever ⟩ "g-as" ← svar "as" in'
  (mod⟨ forever ⟩ (g-cons A ·⟨ constantly ⟩ svar "a"
                            ·⟨ later ⟩ svar "g-as"))
nats : TmExpr ★
nats = mod⟨ forever ⟩ g-nats
```

(c) Sikkel code for guarded type theory.

Figure 5: Streams in Agda and guarded recursive streams in Sikkel (Section 3).

3 Application 1: Guarded Recursive Type Theory

To illustrate the use of Sikkel in practice, we demonstrate the use of guarded recursion. To motivate guarded recursion, consider the definitions in Figure 5a which illustrate Agda's support for infinite structures using coinduction and copatterns. The code defines infinite streams as a coinductive record, essentially by specifying how they can be destructed: by taking their head of type A and their tail which is again of type Stream A. Values of such a coinductive type can be created using copattern matching and corecursion. This is used to define a stream of zeros by specifying its head and its tail. Note how the tail is defined corecursively by referring to the stream we are defining. Slightly more complicated is the definition of map, which consumes one stream while producing another one.

For Agda to be sound as a logic and for type checking to be decidable, corecursive definitions must be productive, meaning that any element of the stream must be computable in a finite amount of time. To enforce productivity, Agda's productivity checker only allows a corecursive call if the operations applied to its result form a strict suffix of the coinductive destructors applied to the left-hand side of the clause. This check is conservative and sometimes quite restrictive, as illustrated by the nats example.[7] This example is rejected by Agda's productivity checker, because the right-hand side of the last clause corecursively refers to nats in an application of map suc, which is not a strict suffix of tail.

[7] The salmon-colored background is applied by Agda to inform that there are termination or productivity issues.

The definition of nats is actually productive: computing the $(n+1)$th element only requires the n-th element of the stream for any n. Agda does not see this because it depends on a property of the function map suc: the fact that it only needs the n-th element of the input stream to produce the n-th element of its output stream. If we replaced map suc with a function flipFst (of the same type Stream $\mathbb{N} \to$ Stream $\mathbb{N}$) that flips the first two elements of a stream, we would be defining head (tail nats) as head (tail nats) itself, which would not be productive.

Guarded recursive type theory, originally proposed by Nakano [29], allows to express the difference between map suc and flipFst in the type system and can therefore accept more productive definitions. In order to work with guarded recursion in Sikkel, we use the same mode theory as Gratzer et al. [20] which is specified in Figure 5b. There is a time-independent or trivial mode $\star$ and a time-dependent mode ω. Intuitively, one can imagine terms at mode ω to unfold over time, revealing at every time step a new piece of information as specified by their type, whereas terms at mode $\star$ can best be thought of as ordinary Agda values. There is a modality later from ω to ω, whose action on types we will denote by $\triangleright\, T$ instead of $\langle$ later $\mid T \rangle$. Intuitively, a term of type $\triangleright\, T$ is just a term of type T whose unfolding process is delayed by one step: the information that would be available today is actually only available tomorrow. The intuition behind the constantly modality is that it takes a static, time-independent value and embeds it as a term in mode ω which does not unfold: it is constantly the same and all information is present from the beginning. The forever modality, on the other hand, converts a time-dependent value to a time-independent one by applying the unfolding *forever*, obtaining the fully unfolded value with all information present.

Following Veltri and van der Weide [39] and Gratzer et al. [20], we then add a new type constructor GStream of *guarded* streams to our language. It takes a type in mode $\star$ and produces a type in mode ω, whose values can be thought of as streams that unfold over time by making available one additional element at every time step. As a result, the tail of such a guarded stream will have type $\triangleright$ (GStream A) as its first element will only be available tomorrow since it is the second element of the original stream. The other built-in operations for guarded streams can be found at the top of Figure 6. Note that the types of g-head and g-cons use the constantly modality to embed a time-independent type into mode ω.

$$\Gamma \vdash \text{g-head} : \text{GStream}\ A \Rrightarrow \langle\ \text{constantly} \mid A\ \rangle \qquad \text{(Tm-GHead)}$$
$$\Gamma \vdash \text{g-tail} : \text{GStream}\ A \Rrightarrow \triangleright\ (\text{GStream}\ A) \qquad \text{(Tm-GTail)}$$
$$\Gamma \vdash \text{g-cons} : \langle\ \text{constantly} \mid A\ \rangle \Rrightarrow \triangleright\ (\text{GStream}\ A) \Rrightarrow \text{GStream}\ A \qquad \text{(Tm-GCons)}$$

$$\frac{\Gamma\ ,\ \text{later} \mid x \in T \vdash t : S \qquad T \simeq^{\text{ty}} S}{\Gamma \vdash \text{löb}[\text{later}|\ x \in T\ ]\ t : T}\ \text{Tm-Löb}$$

Figure 6: Additional typing rules for the Sikkel implementation of guarded recursive type theory.

Guarded recursive type theory offers a built-in induction primitive called Löb induction (Tm-Löb, see Figure 6). When constructing a value of any type T at mode ω, it allows us to use a variable of type T that appears under the later modality. This variable will represent a (co)recursive call, and the fact that it appears under the later modality prevents us from writing unproductive definitions.

We can now implement a map operation g-map for guarded streams in Sikkel (Fig. 5c). We use modal elimination twice to make sure that there are variables representing the head and the tail of `"s"` and that they appear in the context under the right modalities to be used in the application of the modal function g-cons. The variable `"m"` bound by Löb induction is used to corecursively map `"f"` over the tail of `"s"`. It is now also possible to write a valid definition of the guarded stream of natural numbers

g-nats (Fig. 5c). Here lit introduces natural number literals and suc is the successor function.

Note that it is possible to implement a function g-flipFst that flips the first two elements of a guarded stream. However, it will have type GStream A ⇛ ▷ (GStream A) because the head of the resulting stream is the second element of the argument stream, which is only available tomorrow. As a result, replacing g-map ·⟨ constantly ⟩ suc with g-flipFst in the definition of g-nats results in an ill-typed term.

Guarded streams do not behave in the same way as Agda's coinductive streams. An important difference is the occurrence of the modalities in the type of g-cons. We can however define a type constructor Stream' of standard streams (as fully unfolded guarded streams) at mode ⋆ by applying the forever modality to the type of guarded streams [39, 20] (Fig. 5c). The cons' function then has the expected type A ⇛ Stream' A ⇛ Stream' A. It is well-typed because of the two modality equivalences from Figure 5b, which let us use the variables "a" and "g-as" via the trivial 2-cell. We can construct the standard stream nats of natural numbers using g-nats (Fig. 5c). Many more examples, including all the examples involving streams by Clouston et al. [13, pp. 8–12], can be found in the Agda implementation.

4 Presheaf Models

For now, the definition of nats in Fig. 5c gives us just the syntax of an MSTT term. However, Sikkel is intended to make modal primitives available for defining regular Agda values, not just terms in a deeply embedded syntax. A standard approach to do this would be to write an interpreter that translates MSTT types and terms to Agda types and terms. However, modal types or term formers like löb would be hard to translate in an off-the-shelf version of Agda. Instead, we will first interpret MSTT types and terms in Sikkel's second, semantic layer: a presheaf model that supports the new primitives of the extended type theory. All of the intended applications of Sikkel listed in the introduction have such a presheaf model.

The intuition behind presheaf models is covered in Section 4.1 and we continue with details about the actual formalization of presheaf models in Section 4.2. We refer to Appendix A for details about the implementation of the semantics of guarded recursion.

4.1 Some Intuition

We can explain the intuition behind presheaf models using the example of guarded recursion from Section 3. Recall that a value at mode ω can be seen as if it unfolds over time, making available some new information at every time step. Consequently, we can model a type at mode ω as a sequence of Agda types, the n-th type representing the unfolding after n steps. For example, the type GStream A will be interpreted as the following diagram of Agda types and functions.

$$\mathsf{Vec}_1\, A \xleftarrow{\mathsf{init}} \mathsf{Vec}_2\, A \xleftarrow{\mathsf{init}} \dots \xleftarrow{\mathsf{init}} \mathsf{Vec}_n\, A \xleftarrow{\mathsf{init}} \dots \tag{1}$$

Here Vec_n is the Agda type constructor for lists of length n and init drops the last element of such a list. A term of type GStream A will then be modeled as a sequence of Agda values of these types compatible with the init functions, i.e. effectively as a vector gaining more elements over time where every element, once present, must remain unchanged.[8]

[8] It might be surprising that a guarded stream is represented as a sequence of vectors (lists), rather than as a sequence of values of type A. However, in the n-th type of the sequence, we want to keep track of *all* information present after n steps, not just the newly added data. The functions in diagram (1) will then tell how to forget the new information when going from the $(n+1)$th to the n-th type. This is the standard way to interpret guarded streams in a presheaf model [9]. Moreover, by changing the types and functions in diagram (1), one can model structures different from streams (e.g. replacing Vec_n with the type constructor for binary trees of height n allows to model infinite binary trees).

In general, a presheaf model is parametrized by a base category. Types at a mode m are modeled as diagrams such as the one above, called *presheaves*, whose shape is determined by the base category that corresponds to m. Depending on the shapes of these diagrams, one can implement operations on semantic types which will serve as the interpretation of modalities or other primitives added to an instance of MSTT. For example, since every type at mode ω is represented as a diagram with a shape as in (1), we can implement the later modality by shifting the sequence one step to the right and adding Agda's unit type $\top$, containing no information, to the front (i.e. delaying the unfolding process by one step).

4.2 Presheaf Models in Agda

Our formalization of presheaf models in Agda follows the general construction by Hofmann [21] and is structured as an internal Category with Families (CwF) [19].

A presheaf model is parametrized by a base category C : BaseCategory, which can be any small category. Its object and morphism types will be denoted Ob and Hom x y. A semantic context Γ : Ctx C is now defined as a presheaf (i.e. a Set-valued contravariant functor) over the base category. In the following record type, as well as in the rest of this section, we omit fields expressing equality laws.

```
record Ctx (C : BaseCategory) : Set₁ where
  field ctx-cell : Ob → Set
        ctx-hom : Hom x y → ctx-cell y → ctx-cell x
```

We use the notation $\Gamma\ \langle\ x\ \rangle$ for the type of cells over x : Ob and $\Gamma\ \langle\langle\ f\ \rangle\rangle\ \gamma$ for the restriction of $\gamma : \Gamma\ \langle\ y\ \rangle$ by f : Hom x y. A semantic context Γ : Ctx C , is a diagram as in (1), with a shape determined by the base category C. Indeed, for every object x of C there is a node in the diagram containing the Agda type $\Gamma\ \langle\ x\ \rangle$ and for every morphism f from x to y in C there is an arrow in the diagram representing an Agda function $\Gamma\ \langle\langle\ f\ \rangle\rangle\_ : \Gamma\ \langle\ y\ \rangle \to \Gamma\ \langle\ x\ \rangle$. The base category that determines the shape of diagram (1) has the natural numbers as objects, giving rise to a sequence of Agda types. Furthermore, it has exactly one morphism from m to n if and only if $m \leq n$. This means that diagram (1) actually has more arrows than shown (e.g. from $\mathsf{Vec}_{n+2}\ A$ to $\mathsf{Vec}_n\ A$), but the omitted equality laws in the definition of Ctx above make sure that all of these can be obtained by composing the init functions.

There is a semantic empty context $\diamond$ that is defined as $\diamond\ \langle\ x\ \rangle = \top$ for every x. Furthermore, a CwF has the notion of semantic substitutions between two contexts. These are implemented in Sikkel, but not needed in the rest of the paper.

Although MSTT is currently not dependently typed, in the future we do plan to support theories where types may depend on variables. Our formalization of presheaf models already anticipates this and every semantic type lives in a certain context. As a result, the representation of semantic types is more complicated than the diagram from (1). We define the type of semantic types in a given context as the following record type (omitting equality laws).[9]

```
record Ty (Γ : Ctx C) : Set₁ where
  field ty-cell : (x : Ob) (γ : Γ ⟨ x ⟩) → Set
        ty-hom : (f : Hom x y) {γy : Γ ⟨ y ⟩} {γx : Γ ⟨ x ⟩} → Γ ⟨⟨ f ⟩⟩ γy ≡ γx → ty-cell y γy → ty-cell x γx
```

Again, we introduce the notation $T\ \langle\ x\ ,\ \gamma\ \rangle$ = ty-cell T x γ and $T\ \langle\langle\ f\ ,\ e\ \rangle\rangle\ t$ = ty-hom T f e t. If T is a type in context Γ, we get an Agda type $T\ \langle\ x\ ,\ \gamma\ \rangle$ for every object x in the base category C and every

[9] Here $\equiv$ is Agda's identity type with reflexivity constructor refl. Arguments between curly brackets are implicit and will be inferred by Agda.

cell $\gamma : \Gamma\ \langle\ x\ \rangle$. (In practice, all types in this paper will be closed and will hence not depend on the cell γ.) Furthermore, there is a restriction map $T\ \langle\langle\ f\ ,\ e\ \rangle\rangle\_ : T\ \langle\ y\ ,\ \gamma_y\ \rangle \to T\ \langle\ x\ ,\ \gamma_x\ \rangle$ for every morphism f from x to y in C and for all cells γ_x and γ_y that satisfy $\Gamma\ \langle\langle\ f\ \rangle\rangle\ \gamma_y \equiv \gamma_x$. This makes T a presheaf over the category of elements of Γ. In standard mathematical presentations of presheaf models, ty-hom has the type $(f : \mathsf{Hom}\ x\ y) \to \text{ty-cell}\ y\ \gamma_y \to \text{ty-cell}\ x\ (\Gamma\ \langle\langle\ f\ \rangle\rangle\ \gamma_y)$. However, in our formalization this turns out to make the implementation of type substitution considerably more complicated. In the context of indexed inductive type families, the technique to introduce a variable γ_x together with a propositional equality constraint is known as fording [28] (named after a quote by Henry Ford: "Any customer can have a car painted any color that he wants so long as it is black.").

A semantic term of type T in context Γ then specifies for every object x in the base category and every cell $\gamma : \Gamma\ \langle\ x\ \rangle$ an Agda value of type $T\ \langle\ x\ ,\ \gamma\ \rangle$.

```
record Tm (Γ : Ctx C) (T : Ty Γ) : Set where field term : (x : Ob) (γ : Γ ⟨ x ⟩) → T ⟨ x , γ ⟩
```

We omit a naturality condition assuring that these Agda values are stable under the restriction maps of T. We use the notation $t\ \langle\ x\ ,\ \gamma\ \rangle' = \text{term}\ t\ x\ \gamma$.

Types in MSTT do not depend on variables in the context. As a result, they can be interpreted as closed types which can live in any semantic context.

```
ClosedTy : BaseCategory → Set₁
ClosedTy C = {Γ : Ctx C} → Ty Γ
```

(We omit a condition that expresses naturality w.r.t. substitutions; this condition implies that a closed type is entirely determined by its manifestation in the empty context.)

Sikkel provides an equivalence relation $\cong^{\mathsf{ty}}$ on semantic types in the same context. It is defined as natural isomorphism, so a proof of $T \cong^{\mathsf{ty}} S$ amounts to a collection of Agda isomorphisms between $T\ \langle\ x\ ,\ \gamma\ \rangle$ and $S\ \langle\ x\ ,\ \gamma\ \rangle$ that is compatible with the restriction maps. Such a proof allows to convert terms of type S to type T via the operation $\iota[\_]\_ : T \cong^{\mathsf{ty}} S \to \mathsf{Tm}\ \Gamma\ S \to \mathsf{Tm}\ \Gamma\ T$ whose implementation we omit.

Regardless of the base category, every presheaf model supports the standard type and term formers for booleans, natural numbers, function types, product types, etc. We will discuss some details about the implementation of function types and refer to the Agda code for the other type formers. If T and S are seen as diagrams, a term of type $T \Rrightarrow S$ consists of a function from every Agda type of T to the corresponding type of S such that all squares arising in this way commute. Hence, one could naively try to define the Agda type $(T \Rrightarrow S)\ \langle\ x\ ,\ \gamma\ \rangle$ as $T\ \langle\ x\ ,\ \gamma\ \rangle \to S\ \langle\ x\ ,\ \gamma\ \rangle$, but that definition does not allow to implement the restriction maps for $T \Rrightarrow S$. The solution is to require such an Agda function not only for x, but for every object y and every morphism from y to x. Consequently, $(T \Rrightarrow S)\ \langle\ x\ ,\ \gamma\ \rangle$ is defined as $\mathsf{PshFun}\ T\ S\ x\ \gamma$, where PshFun is the following record type:

```
record PshFun (T S : Ty Γ) (x : Ob) (γx : Γ ⟨ x ⟩) : Set where
  field fun : ∀ {y} (ρ : Hom y x) {γy : Γ ⟨ y ⟩} → Γ ⟨⟨ ρ ⟩⟩ γx ≡ γy → T ⟨ y , γy ⟩ → S ⟨ y , γy ⟩
```

Again a naturality condition has been omitted. We use the notation $f\ \$\langle\ \rho\ ,\ e\ \rangle\ t = \text{fun}\ f\ \rho\ e\ t$. The restriction maps can then be implemented by using composition in the base category. Just like for ty-hom in the definition of Ty, we use fording for the field fun to make the implementation of function types considerably easier.

A modality will be interpreted as a dependent right adjunction (DRA) defined by Birkedal et al. [8].[10]

[10] In fact a DRA requires the lock operation to be a functor between context categories that also acts on semantic substitutions. We follow this in the Sikkel implementation, but simplified the presentation in the paper.

```
record Modality (C D : BaseCategory) : Set₁ where
  field lock : Ctx D → Ctx C
        mod : Ty (lock Γ) → Ty Γ
        mod-intro : Tm (lock Γ) T → Tm Γ (mod T)
        mod-elim : Tm Γ (mod T) → Tm (lock Γ) T
```

For readability we use the notation $\langle\ \mu \mid T\ \rangle = \mathsf{mod}\ \mu\ T$. As we can see, a DRA specifies the semantic counterparts of the context and type formers associated with modalities in MSTT, as well as the interpretation of the modal term constructor. The eliminator mod-elim is the inverse of mod-intro and is hence different from the rule Tm-ModElim in Figure 4. This difference will be handled by the type checker when interpreting MSTT syntax in the model. There is a unit DRA $\mathbb{1}$ for every base category (whose constituent operations are just the identity) as well as composition of DRAs (by composing the constituent operations). Furthermore, Sikkel has an equivalence relation $\cong^{\mathsf{m}}$ for DRAs according to which composition is associative and the unit DRA is a unit. We will not discuss the definition of $\cong^{\mathsf{m}}$, but it is sufficient to know that if $\mu \cong^{\mathsf{m}} \rho$, then it follows that $\langle\ \mu \mid A\ \rangle \cong^{\mathsf{ty}} \langle\ \rho \mid A\ \rangle$ for every A : ClosedTy C. Finally, there is a type family TwoCell, indexed by two DRAs, whose values are essentially natural transformations between the lock context functors of the DRAs. We refer to the Agda code for details.

5 A Sound Type Checker

The typing relation in Figure 4 is not formalized as a relation in Agda, but as an algorithm that checks whether a term is well-typed. In fact, as pointed out before, the terms of MSTT provide enough information for types to be inferred rather than checked so there is a function infer-type that returns a term's type, provided that it is well-typed.

Moreover, Sikkel's type checker is sound by construction, so on top of returning a well-typed term's type, it also returns its denotation in the presheaf model. As such, the type checker bridges Sikkel's syntactic and semantic layers. We will use the prefix "M." (for model) to disambiguate names that are defined in both the semantic and syntactic layer.

```
⟦_⟧mode : ModeExpr → BaseCategory
⟦_⟧modality : ModalityExpr m n →
              Modality ⟦ m ⟧mode ⟦ n ⟧mode
```

(a) Modes and modalities (these are additional fields to ModeTheory, Fig. 2).

```
⟦_⟧ty : TyExpr m → ClosedTy ⟦ m ⟧mode
⟦ T ⇛ S ⟧ty = ⟦ T ⟧ty M.⇛ ⟦ S ⟧ty
⟦ ⟨ μ | T ⟩ ⟧ty = M.⟨ ⟦ μ ⟧modality | ⟦ T ⟧ty ⟩
```

(b) Types (similar clauses for other types are omitted).

```
⟦_⟧ctx : CtxExpr m → Ctx ⟦ m ⟧mode
⟦ ◇ ⟧ctx = M.◇
⟦ Γ , μ | x ∈ T ⟧ctx =
  ⟦ Γ ⟧ctx ,, M.⟨ ⟦ μ ⟧modality | ⟦ T ⟧ty ⟩
⟦ Γ ,lock⟨ μ ⟩ ⟧ctx = lock ⟦ μ ⟧modality ⟦ Γ ⟧ctx
```

(c) Contexts. Here, _,,_ is semantic context extension (the semantic layer uses a de Bruijn representation of variables).

Figure 7: Interpretation functions for necessarily well-typed expressions.

Before we can interpret terms, we must know how to interpret types, modes and modalities. As already mentioned in Section 4, modes in MSTT will be interpreted as base categories and modalities as DRAs. Correspondingly, in Fig. 7a we add two fields to the definition of ModeTheory from Fig. 2.

Furthermore, ModeTheory additionally requires the interpretation of the modality $\mathbb{1}$ to be equivalent to the DRA $M.\mathbb{1}$ according to $\cong^{\mathsf{m}}$, and similarly for composition of modalities resp. DRAs. MSTT types are interpreted as closed semantic types and MSTT contexts as semantic contexts (Figs. 7b and 7c).

In order to handle type errors, the type checker makes use of a type checking monad (Fig. 8a).[11] A value of type TCM A represents either a type error (with a string as message) or a value of type A. TCM has the structure of a monad, and in fact it is just a reformulation of Haskell's error monad.

An additional field ⟦_∈_⇒_⟧two-cell is included in the definition of ModeTheory (Fig. 8b). It will result in a type error if the given modalities cannot be the domain and codomain of the given 2-cell, and otherwise it will return the interpretation of the 2-cell in the presheaf model.

The type checker will also need to test whether two modes or two modalities are equivalent. Therefore, we add two more fields to the definition of ModeTheory (Fig. 8d). The first function takes two syntactic modes and either provides a proof that these modes are syntactically equal or results in a type error if they are not.[12] The second function is the long awaited specification of the modality equivalence relation $\simeq^{\mathsf{m}}$, not as a predicate but as a sound decision procedure. As such, it tests whether two syntactic modalities of the same domain and codomain are equivalent, and if so, it produces a proof that the modalities' interpretations as DRAs are equivalent. As mentioned in 2.1, it is expected that associativity and unit laws of ⓜ hold up to $\simeq^{\mathsf{m}}$, but it is not necessary to prove this explicitly.

```
data TCM (A : Set ℓ) : Set ℓ where
  type-error : String → TCM A
  ok : A → TCM A
```

(a) Type-checking monad.

```
⟦_∈_⇒_⟧two-cell : TwoCellExpr →
  (μ ρ : ModalityExpr m n) →
  TCM (TwoCell ⟦ μ ⟧modality ⟦ ρ ⟧modality)
```

(b) Checking and interpretation of 2-cells (an additional field to ModeTheory, Fig. 2).

```
record InferInterpretResult
         (Γ : CtxExpr m) : Set where
  constructor _,_
  field type : TyExpr m
        interpretation : Tm ⟦ Γ ⟧ctx ⟦ type ⟧ty
```

(c) Output type of successful type inference.

```
_=mode?_ : (m n : ModeExpr) → TCM (m ≡ n)
_≃ᵐ?_ : (μ ρ : ModalityExpr m n) →
        TCM (⟦ μ ⟧modality ≅ᵐ ⟦ ρ ⟧modality)
```

(d) Equivalence checking of modes and modalities (these are additional fields to ModeTheory, Fig. 2).

```
infer-interpret : TmExpr m → (Γ : CtxExpr m) →
                  TCM (InferInterpretResult Γ)
infer-interpret (mod⟨ μ ⟩ t) Γ = do
  T , sem-t ← infer-interpret t (Γ ,lock⟨ μ ⟩)
  return (⟨ μ | T ⟩ , mod-intro ⟦ μ ⟧modality sem-t)
infer-interpret (ann t ∈ S) Γ = do
  T , sem-t ← infer-interpret t Γ
  e ← S ≃ᵗʸ? T
  return (S , ι[ e ] sem-t)
```

(e) Type inference (most clauses are omitted).

Figure 8: Equivalence checking and type inference.

Analogously, the Sikkel type checker also provides a function $\_\simeq^{\mathsf{ty}}?\_ : (T\ S : \mathsf{TyExpr}\ m) \to \mathsf{TCM}\ (\llbracket T \rrbracket\mathsf{ty} \cong^{\mathsf{ty}} \llbracket S \rrbracket\mathsf{ty})$ to test whether two types are equivalent according to the relation $\simeq^{\mathsf{ty}}$.

If an MSTT term is well-typed, the type checker will return both its type and its denotation in the presheaf model. We pack this result in the record type InferInterpretResult (Fig. 8c). In Fig. 8e, we

[11] The ℓ in Set ℓ is a universe level. This means that TCM can act on any Agda type, not just types in one particular universe.

[12] We check syntactic equality, as the semantic layer does not account for equivalence of modes, i.e. base categories.

look at some of the cases in the implementation of the type checker. We make use of Agda's do-notation for the TCM monad. When inferring the type of mod⟨ μ ⟩ t in a context Γ, we first infer the type of t in the locked context Γ ,lock⟨ μ ⟩ and bind the result to T. Simultaneously, the denotation of t in the presheaf model is computed and bound to the variable *sem-t*, which will have the Agda type Tm (lock ⟦ μ ⟧modality ⟦ Γ ⟧ctx) ⟦ T ⟧ty. Then we infer that the type of the original term is ⟨ μ | T ⟩ and we construct the required semantic term of semantic type M.⟨ ⟦ μ ⟧modality | ⟦ T ⟧ty ⟩ by applying the introduction operation for DRAs to *sem-t*. In the case of an annotated term ann t ∈ S, we first infer the type and denotation of t. The variable *sem-t* will now have the Agda type Tm ⟦ Γ ⟧ctx ⟦ T ⟧ty. Next, we verify that S and T are equivalent types and as a result we get a proof e : ⟦ S ⟧ty $\cong^{\mathsf{ty}}$ ⟦ T ⟧ty that the denotations of S and T in the presheaf model are isomorphic. Finally, we say that the annotated term has type S and we produce a semantic term of type ⟦ S ⟧ty by transporting *sem-t* over the isomorphism e. We refer to the Agda implementation for all other clauses of infer-interpret.

Using infer-interpret, we can implement an operation ⟦_⟧tm-in_ that accepts an MSTT term and context and returns the term's denotation if it is well-typed or a value of the unit type ⊤ if it is not (hence this operation is a dependent function whose result type depends on the success of the Sikkel type checker). After extending Sikkel's generic type checker with cases for Löb induction and the operations for guarded streams, we can then interpret the examples from Section 3 in the presheaf model for guarded recursion. For example, the denotations of the streams g-nats and nats from Fig. 5c in the empty context are computed as follows.

```
g-nats-sem : Tm {M.ω} M.⋄
  (M.GStream M.Nat)
g-nats-sem = ⟦ g-nats ⟧tm-in ⋄
```

```
nats-sem : Tm {M.★} M.⋄
  M.⟨ M.forever | M.GStream M.Nat ⟩
nats-sem = ⟦ nats ⟧tm-in ⋄
```

Note that these have computational content in Agda, e.g. g-nats-sem ⟨ n , tt ⟩' is the Agda list consisting of the first $n+1$ natural numbers for any n.

6 Extraction to the Meta-level

The trivial base category ★ corresponding to the trivial mode for guarded recursion has one object (its type Ob of objects is Agda's unit type ⊤) and one trivial morphism. In the presheaf model over ★, a type T and a term t in the empty context are nothing more than respectively an Agda type T ⟨ tt , tt ⟩ and an Agda term t ⟨ tt , tt ⟩'. In fact, the presheaf model over ★ is the standard set model of type theory.

However, we cannot directly use this to extract a value out of a semantic term t (in the empty context with base category ★) to get to the third layer in Figure 1. The reason is that this value does not always have the desired type on the nose. For example, if t has type $T \Rrightarrow S$, then t ⟨ tt , tt ⟩' will have the Agda type PshFun T S tt tt and not the function type T ⟨ tt , tt ⟩ → S ⟨ tt , tt ⟩, although these two types are isomorphic. Similarly, the Agda value obtained from a semantic term of type ⟨ forever | GStream Nat ⟩ would have type ∀ n → Vec ℕ (suc n) (together with a naturality condition) and not Stream ℕ. Again, these two Agda types are isomorphic.

To bridge this gap, Sikkel has a type class Extractable for closed semantic types over ★.

```
record Extractable (T : ClosedTy ★) : Set₁ where
  field translated-type : Set
        extract-term : Tm ⋄ T → translated-type
        embed-term : translated-type → Tm ⋄ T
```

In order for a semantic type to be an instance of this type class, we must specify its intended translation as an Agda type. Furthermore, we must be able to produce an Agda value of this type given a semantic term in the empty context and vice versa. It is expected that these functions constitute an isomorphism, but this is not formally required and there are no corresponding proof obligations (for some types this will only be provable when assuming certain axioms, e.g. for function types, one should assume function extensionality). The field embed-term is essentially needed for the translation of function types.

Sikkel provides Extractable instances for natural numbers, booleans, functions and products. Moreover, we can also provide an instance for ⟨ forever | GStream A ⟩ whenever A is extractable, the extraction resulting in an Agda stream. In this way, we can finally construct the Agda stream of all natural numbers.

```
nats-agda : Stream ℕ
nats-agda = extract-term nats-sem
```

7 Application 2: Representation Independence through Parametricity

An important characteristic of Sikkel is that its different layers are parametrized by the mode theory and base category. As a result, Sikkel can be applied to extend type theory with other features than guarded recursion. In this section, we demonstrate a basic parametric type theory using a toy example where we define subtraction of integers in terms of addition and negation.

Consider the minimalistic interface for integers in Figure 9a. This interface can for instance be implemented by the types $\mathbb{N} \times \mathbb{N}$ (representing differences of natural numbers), or Sign $\times \mathbb{N}$ (representing natural numbers with a sign, with two representations for zero). We can use the operations in IntStructure to implement new functions such as subtract in Figure 9a.

```
record IntStructure (A : Set) : Set where
  field add : A → A → A
        negate : A → A
subtract : IntStructure A → A → A → A
subtract s x y = add s x (negate s y)
```

(a) For Agda types.

```
record IntStructure (A : TyExpr m) : Set where
  field add negate : TmExpr m
subtract : (IntStructure A) → TmExpr m
subtract s = lam[ "a" ∈ A ] lam[ "b" ∈ A ]
               add s · svar "a" · (negate s · svar "b")
```

(b) For Sikkel types.

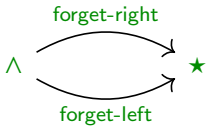
Figure 9: Minimalistic interface for integers and implementation of subtraction.

The function subtract operates on an abstract type A, with only the operations exposed in the IntStructure interface. Because of this, it will satisfy a form of *representation independence*: its behavior will not depend on the underlying representation of the abstract type A, but only on the behavior exposed through the interface. This is traditionally formalized as a form of parametricity: for any relation R between two types implementing IntStructure, R will be preserved by subtract if it is preserved by the operations add and negate [35].

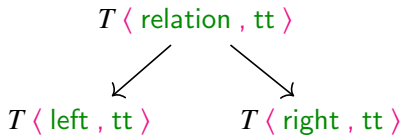
In Sikkel, we support such results in a parametric type theory with mode theory given in Figure 10a. Mode ⋆ is again the trivial mode, interpreted as the trivial base category. There is also a parametric mode, denoted by a wedge (∧) and interpreted as the walking cospan category M.∧. This category has 3 objects left, right and relation and 2 non-trivial morphisms: one from both left and right to relation. Consequently, a presheaf over M.∧ is a diagram as depicted in Figure 10b, i.e. a span of types. We can

think of such a presheaf as consisting of two Agda types, a left one and a right one, together with a relation between them which is represented as the type of related pairs. The 2 restriction maps will map every related pair to its left and right components.

We can reformulate the interface for MSTT types and implement subtraction as shown in Figure 9b. Since MSTT is extrinsically typed, we cannot directly require add and negate to have a specific type. The actual definition of IntStructure therefore contains two more fields requiring that the types inferred for add and negate should be equal to respectively $A \Rrightarrow A \Rrightarrow A$ and $A \Rrightarrow A$.



(a) Mode theory.



(b) A semantic type T in the empty context over base category M.∧.

Figure 10: Mode theory for parametricity.

This application adds a new type former FromRel to MSTT, which allows a user to create new built-in types in mode ∧ that represent two user-specified Agda types with a relation between them. In our case, we add the built-in type ℤ that is interpreted as $\mathbb{N} \times \mathbb{N}$ and $\mathsf{Sign} \times \mathbb{N}$ with the relation ∼ expressing that a pair of natural numbers and a signed natural number represent the same integer. When interpreting this type ℤ in the presheaf model, the Agda type at the top of the diagram in Figure 10b will be Σ[*(i , j)* ∈ (ℕ × ℕ) × (Sign × ℕ)] *i* ∼ *j*, i.e. the type of related pairs (using Agda's Σ types). Furthermore, new term formers allow to construct MSTT functions between these built-in types given the corresponding Agda functions for the left and right types and a proof that these functions preserve the built-in relations. This means that we can create an implementation ℤ-int of IntStructure for ℤ from implementations of addition and negation for $\mathbb{N} \times \mathbb{N}$ and $\mathsf{Sign} \times \mathbb{N}$ and proofs that they respect the relation ∼.

Recalling the implementation of function types in a presheaf model, the interpretation of subtract ℤ-int in the semantic layer consists of the implementation of subtraction for $\mathbb{N} \times \mathbb{N}$ and $\mathsf{Sign} \times \mathbb{N}$, together with a proof that they respect the relation ∼. We can obtain the subtraction implementations at the Agda level by using the modalities from the mode theory. The forget-right modality turns a ∧-type into a ★-type by forgetting about the right type and the relation. Hence we can write an MSTT function

```
subtract★-left : TmExpr ★
subtract★-left = lam[ forget-right | "x" ∈ ℤ ] lam[ forget-right | "y" ∈ ℤ ]
  mod⟨ forget-right ⟩ (subtract ℤ-int · svar "x" · svar "y")
```

of type ⟨ forget-right | ℤ ⟩ ⇛ ⟨ forget-right | ℤ ⟩ ⇛ ⟨ forget-right | ℤ ⟩ and use the sound type-checker and extraction to obtain a function subtract-ℕ×ℕ for $\mathbb{N} \times \mathbb{N}$, and similarly subtract-Sign×ℕ.[13] Finally, we have a proof that these functions respect the relation ∼.

```
subtract-∼ : ∀ i1 j1 i2 j2 → i1 ∼ i2 → j1 ∼ j2 → subtract-ℕ×ℕ i1 j1 ∼ subtract-Sign×ℕ i2 j2
subtract-∼ i1 j1 i2 j2 ri rj = proj₂ ((([[ subtract ℤ-int ]]tm-in ⋄) ⟨ relation , tt ⟩'
  $⟨ relation-id , refl ⟩ ((i1 , i2) , ri)) $⟨ relation-id , refl ⟩ ((j1 , j2) , rj))
```

[13] It may seem laborious to define two analogues of subtract★-left for every parametric function we define. Instead we can define an applicative operator for arbitrary modalities μ and arbitrary functions f with k arguments. The instance for μ = forget-right and f = subtract ℤ-int and $k = 2$ yields subtract★-left.

8 Discussion, Related and Future Work

One of Sikkel's design choices is to make its syntactic layer extrinsically typed, i.e. the Agda type TmExpr is not indexed by a context or type of the object theory. A downside of this approach is that a user of the library cannot rely on Agda to indicate the expected types of subterms when writing programs in Sikkel. However, an intrinsically typed syntax would require explicit casting by the programmer whenever an equivalence of modalities is used, something which is now handled by Sikkel's type checker. Furthermore, an extrinsically typed syntax allows for easier implementation of named variables.

Sikkel's goal of extending type theories with primitives like guarded recursion or parametricity can be approached in many other ways as well.

One can simply modify an existing implementation of type theory, as was done in Agda-parametric [33], cubical Agda [40] or guarded cubical Agda [38]. Trading type-checking performance for flexibility, one can alternatively rely on postulates and rewrite rules [14]. Compared to Sikkel, such approaches have benefits and downsides: the result can be very user-friendly (since both syntax and typing rules can be controlled) but implementation errors may compromise soundness of the theory, it is not possible to restrict the effect of the modifications to a part of a larger codebase and it can be hard to deviate strongly from the structural rules of the system, as required for some type theory extensions, e.g. [5, 31].

Some other proposals are more closely related to Sikkel. Veltri and van der Weide [39] present an implementation of guarded recursion, which is closely related but uses a less general model and relies on Agda's experimental support for sized types. Jaber et al. [23] translate an extension of the calculus of constructions (CoC) to presheaves over an arbitrary preorder in regular CoC. Lacking modes (and the trivial mode), they do not support extracting code in the extension to regular CoC. Bach Poulsen et al. have implemented guarded type theory as ordered families of equivalences (OFEs) [3].[14] These are also used in the model of the Iris logic [24] which comes with support for interactive proofs in the Iris Proof Mode [25]. Finally, presheaves have been formalized for different purposes than Sikkel [7, 22, 37], but these formalizations do not provide all of Sikkel's features.

In future work, we plan to extend Sikkel with support for dependent types. As mentioned in Section 4, our formalization of presheaf models already anticipates this. However, with a dependently-typed syntax the interpretation functions for contexts, types and terms become mutually recursive and it is a significant challenge to satisfy Agda's termination checker. Furthermore, the addition of a Hofmann-Streicher universe to our presheaf model turns out to be non-trivial. Apart from the applications worked out in this paper, we intend to use Sikkel for other type theory extensions like nominal [34], directed [41] or univalent type theory [15], as well as for the implementation of additional primitives like the transpension type [32]

Acknowledgements Joris Ceulemans and Andreas Nuyts hold a PhD Fellowship and a Postdoctoral Fellowship, respectively, from the Research Foundation – Flanders (FWO). This work was partially supported by the research project G0G0519N of the Research Foundation – Flanders (FWO).

References

[1] Robert Atkey (2012): *Relational Parametricity for Higher Kinds*. In: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, *Leibniz International Proceedings in Informatics (LIPIcs)* 16, pp. 46–61, doi:10.4230/LIPIcs.CSL.2012.46.

[14] Available at `https://github.com/metaborg/mj.agda/tree/develop/src/Categorical`.

[2] Robert Atkey, Neil Ghani & Patricia Johann (2014): *A Relationally Parametric Model of Dependent Type Theory*. In: *Principles of Programming Languages*, doi:10.1145/2535838.2535852.

[3] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers & Eelco Visser (2017): *Intrinsically-Typed Definitional Interpreters for Imperative Languages*. *Proc. ACM Program. Lang.* 2(POPL), doi:10.1145/3158104. Available at https://doi.org/10.1145/3158104.

[4] Jean-Philippe Bernardy, Thierry Coquand & Guilhem Moulin (2015): *A Presheaf Model of Parametric Type Theory*. *Electron. Notes in Theor. Comput. Sci.* 319, pp. 67 – 82, doi:10.1016/j.entcs.2015.12.006.

[5] Jean-Philippe Bernardy, Andreas Nuyts & Andrea Vezzosi (2017): *Parametric application*. Private communication.

[6] Marc Bezem, Thierry Coquand & Simon Huber (2014): *A Model of Type Theory in Cubical Sets*. In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, 26, Dagstuhl, Germany, pp. 107–128, doi:10.4230/LIPIcs.TYPES.2013.107. Available at http://drops.dagstuhl.de/opus/volltexte/2014/4628.

[7] Mark Bickford (2018): *Formalizing Category Theory and Presheaf Models of Type Theory in Nuprl*. Available at https://arxiv.org/abs/1806.06114. ArXiv pre-print.

[8] Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts & Bas Spitters (2020): *Modal dependent type theory and dependent right adjoints*. *Mathematical Structures in Computer Science* 30(2), pp. 118–138, doi:10.1017/S0960129519000197.

[9] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer & 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, doi:10.2168/LMCS-8(4:1)2012. Available at https://lmcs.episciences.org/1041.

[10] Ales Bizjak & Rasmus Ejlers Møgelberg (2020): *Denotational semantics for guarded dependent type theory*. *Math. Struct. Comput. Sci.* 30(4), pp. 342–378, doi:10.1017/S0960129520000080. Available at https://doi.org/10.1017/S0960129520000080.

[11] Evan Cavallo & Robert Harper (2020): *Internal Parametricity for Cubical Type Theory*. In: *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, pp. 13:1–13:17, doi:10.4230/LIPIcs.CSL.2020.13. Available at https://doi.org/10.4230/LIPIcs.CSL.2020.13.

[12] Evan Cavallo, Anders Mörtberg & Andrew W Swan (2020): *Unifying Cubical Models of Univalent Type Theory*. In Maribel Fernández & Anca Muscholl, editors: *Computer Science Logic (CSL 2020)*, *LIPIcs* 152, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 14:1–14:17, doi:10.4230/LIPIcs.CSL.2020.14. Available at https://drops.dagstuhl.de/opus/volltexte/2020/11657.

[13] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl & 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, doi:10.2168/LMCS-12(3:7)2016. Available at https://lmcs.episciences.org/2019.

[14] Jesper Cockx, Nicolas Tabareau & Théo Winterhalter (2021): *The Taming of the Rew: A Type Theory with Computational Assumptions*. *Proceedings of the ACM on Programming Languages* 5(POPL), pp. 60:1–60:29, doi:10.1145/3434341.

[15] Cyril Cohen, Thierry Coquand, Simon Huber & Anders Mörtberg (2017): *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*. *FLAP* 4(10), pp. 3127–3170. Available at http://www.cse.chalmers.se/~simonhu/papers/cubicaltt.pdf.

[16] Agda community (2021): *Sized types are no longer safe*. Available at https://github.com/agda/agda/pull/5354. Agda issue 5354.

[17] Thierry Coquand & Gérard Huet (1988): *The calculus of constructions*. *Information and Computation* 76(2), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[18] Thierry Coquand & Christine Paulin (1988): *Inductively defined types*. In: *COLOG '88*, pp. 50–66, doi:10.1007/3-540-52335-9_47. Available at https://doi.org/10.1007/3-540-52335-9_47.

[19] Peter Dybjer (1996): *Internal type theory*. In Stefano Berardi & Mario Coppo, editors: *Types for Proofs and Programs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 120–134, doi:10.1007/3-540-61780-9_66.

[20] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts & Lars Birkedal (2021): *Multimodal Dependent Type Theory*. *Logical Methods in Computer Science* Volume 17, Issue 3, doi:10.46298/lmcs-17(3:11)2021. Available at https://lmcs.episciences.org/7713.

[21] Martin Hofmann (1997): *Syntax and Semantics of Dependent Types*, chapter 4, pp. 79–130. Cambridge University Press.

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

[23] Guilhem Jaber, Nicolas Tabareau & Matthieu Sozeau (2012): *Extending Type Theory with Forcing*. In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)*, IEEE Computer Society Press, pp. 395–404, doi:10.1109/LICS.2012.49.

[24] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal & Derek Dreyer (2018): *Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic*. *Journal of Functional Programming* 28, doi:10.1017/S0956796818000151.

[25] Robbert Krebbers, Amin Timany & Lars Birkedal (2017): *Interactive Proofs in Higher-Order Concurrent Separation Logic*. In: *POPL*, ACM, pp. 205–217, doi:10.1145/3009837.3009855.

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

[27] Per Martin-Löf (1984): *Intuitionistic type theory*. In: *Studies in proof theory*, Bibliopolis.

[28] Conor McBride (2000): *Dependently typed functional programs and their proofs*. Ph.D. thesis, University of Edinburgh, UK. Available at http://hdl.handle.net/1842/374.

[29] Hiroshi Nakano (2000): *A modality for recursion*. In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science*, pp. 255–266, doi:10.1109/LICS.2000.855774.

[30] Andreas Nuyts & 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*, LICS '18, Association for Computing Machinery, New York, NY, USA, pp. 779–788, doi:10.1145/3209108.3209119. Available at https://doi.org/10.1145/3209108.3209119.

[31] Andreas Nuyts & Dominique Devriese (2019): *Dependable Atomicity in Type Theory*. In: *TYPES*.

[32] Andreas Nuyts & Dominique Devriese (2020): *Transpension: The Right Adjoint to the Pi-Type*. *arXiv:2008.08533 [cs]*. arXiv:2008.08533.

[33] Andreas Nuyts, Andrea Vezzosi & Dominique Devriese (2017): *Parametric Quantifiers for Dependent Type Theory*. *Proc. ACM Program. Lang.* 1(ICFP), doi:10.1145/3110276. Available at https://doi.org/10.1145/3110276.

[34] Andrew M. Pitts, Justus Matthiesen & Jasper Derikx (2015): *A Dependent Type Theory with Abstractable Names*. *Electronic Notes in Theoretical Computer Science* 312, pp. 19 – 50, doi:10.1016/j.entcs.2015.04.003. Available at http://www.sciencedirect.com/science/article/pii/S1571066115000079. Ninth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014).

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

[36] E. Riehl & M. Shulman (2017): *A type theory for synthetic ∞-categories*. *ArXiv e-prints*. arXiv:1705.07442.

[37] Amin Timany & Bart Jacobs (2016): *Category Theory in Coq 8.5*. In Delia Kesner & Brigitte Pientka, editors: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 52, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 30:1–30:18, doi:10.4230/LIPIcs.FSCD.2016.30. Available at http://drops.dagstuhl.de/opus/volltexte/2016/6000.

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

[39] Niccolò Veltri & Niels van der Weide (2019): *Guarded Recursion in Agda via Sized Types*. In Herman Geuvers, editor: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 131, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 32:1–32:19, doi:10.4230/LIPIcs.FSCD.2019.32. Available at http://drops.dagstuhl.de/opus/volltexte/2019/10539.

[40] Andrea Vezzosi, Anders Mörtberg & Andreas Abel (2021/ed): *Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types*. *Journal of Functional Programming* 31, doi:10.1017/S0956796821000034.

[41] Matthew Z. Weaver & Daniel R. Licata (2020): *A Constructive Model of Directed Univalence in Bicubical Sets*. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi & Dale Miller, editors: *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, ACM, pp. 915–928, doi:10.1145/3373718.3394794. Available at https://doi.org/10.1145/3373718.3394794.

[42] Martin Wildmoser & Tobias Nipkow (2004): *Certifying Machine Code Safety: Shallow Versus Deep Embedding*. In: *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 305–320, doi:10.1007/978-3-540-30142-4_22.

A Presheaf Semantics of Guarded Recursion

In this appendix we take a closer look at the semantics of some of the primitives from Section 3.

The trivial mode corresponds to the trivial base category $\star$ with one object and one (identity) morphism from that object to itself. Presheaves over $\star$ are essentially sets. On the other hand, the base category ω corresponding to the time-dependent mode has the natural numbers as objects and the type of morphisms from numbers m to n is Agda's inequality type $m \leq n$. This inequality type has two constructors $\mathsf{z{\leq}n} : 0 \leq n$ and $\mathsf{s{\leq}s} : m \leq n \to \mathsf{suc}\ m \leq \mathsf{suc}\ n$. As such, a presheaf over ω is a diagram with a shape as in (1). The presheaf category over ω is called the topos of trees and is a well-known model for guarded recursion [9].

As mentioned in Section 4.1, we can implement the the later modality by shifting a diagram to the right and adding Agda's unit type to the front. The corresponding lock operation is called $\blacktriangleleft$ (earlier) and does the opposite: it shifts a diagram to the left.

$$\blacktriangleleft : \mathsf{Ctx}\ \omega \to \mathsf{Ctx}\ \omega$$
$$\blacktriangleleft\ \Gamma\ \langle\ n\ \rangle = \Gamma\ \langle\ \mathsf{suc}\ n\ \rangle$$

```
◂ Γ ⟨⟨ m≤n ⟩⟩ γ = Γ ⟨⟨ s≤s m≤n ⟩⟩ γ

▷ : Ty (◂ Γ) → Ty Γ
▷ T ⟨ zero , γ ⟩ = ⊤
▷ T ⟨ suc n , γ ⟩ = T ⟨ n , γ ⟩
▷ T ⟨⟨ z≤n , e ⟩⟩ t = tt
▷ T ⟨⟨ s≤s m≤n , e ⟩⟩ t = T ⟨⟨ m≤n , e ⟩⟩ t
```

Here we defined `▷ T ⟨ n , γ ⟩` by pattern matching on n and `▷ T ⟨⟨ m≤n , e ⟩⟩ t` by pattern matching on $m{\leq}n$.

Another intuition that we can make precise now is the representation of guarded streams as the diagram in (1).

```
GStream : Ty (lock constantly Γ) → Ty Γ
GStream A ⟨ n , γ ⟩ = Vec (⟨ constantly | A ⟩ ⟨ n , γ ⟩) (suc n)
```

`GStream`'s input and output types live over a different base category, but the `constantly` modality bridges this gap. The implementation of the restriction maps and the `constantly` modality can be found in the Agda code.