2-Functoriality of Initial Semantics, and Applications

Ahrens, Benedikt; Lafont, Ambroise; Lamiaux, Thomas

DOI
10.1145/3747527

Publication date
2025

Document Version
Final published version

Published in
Proceedings of the ACM on Programming Languages

Citation (APA)
Ahrens, B., Lafont, A., & Lamiaux, T. (2025). 2-Functoriality of Initial Semantics, and Applications. *Proceedings of the ACM on Programming Languages*, *9*(ICFP), Article 258. https://doi.org/10.1145/3747527

2-Functoriality of Initial Semantics, and Applications

BENEDIKT AHRENS, Delft University of Technology, Netherlands
AMBROISE LAFONT, LIX, Ecole Polytechnique, France and Inria, France
THOMAS LAMIAUX, Nantes University, France and Inria, France

Initial semantics aims to model inductive structures and their properties, and to provide them with recursion principles respecting these properties. An ubiquitous example is the `fold` operator for lists. We are concerned with initial semantics that model languages with variable binding and their substitution structure, and that provide substitution-safe recursion principles.

There are different approaches to implementing languages with variable binding depending on the choice of representation for contexts and free variables, such as unscoped syntax, or well-scoped syntax with finite or infinite contexts. Abstractly, each approach corresponds to choosing a different monoidal category to model contexts and binding, each choice yielding a different notion of "model" for the same abstract specification (or "signature").

In this work, we provide tools to compare and relate the models obtained from a signature for different choices of monoidal category. We do so by showing that initial semantics naturally has a 2-categorical structure when parametrized by the monoidal category modeling contexts. We thus can relate models obtained from different choices of monoidal categories provided the monoidal categories themselves are related. In particular, we use our results to relate the models of the different implementation — de Bruijn vs locally nameless, finite vs infinite contexts —, and to provide a generalized recursion principle for simply-typed syntax.

CCS Concepts: • **Theory of computation → Categorical semantics**.

Additional Key Words and Phrases: initial semantics, variable binding, substitution, monoidal categories

ACM Reference Format:
Benedikt Ahrens, Ambroise Lafont, and Thomas Lamiaux. 2025. 2-Functoriality of Initial Semantics, and Applications. *Proc. ACM Program. Lang.* 9, ICFP, Article 258 (August 2025), 32 pages. https://doi.org/10.1145/3747527

1 Introduction

When studying abstract syntax and initial semantics for it, a mathematical modelling of contexts and variable binding needs to be chosen. This choice, while seemingly insignificant, determines the remainder of the theory, and shapes the recursion principle obtained from the initiality property. In this work, we provide an abstract framework formalizing the dependency of the relating different ways of representing contexts and variable binding.

In the remainder of the introduction, we provide a gentle introduction to initial semantics (in Section 1.1) and to the particular challenge of modelling substitution (in Section 1.2).

Authors' Contact Information: Benedikt Ahrens, Delft University of Technology, Delft, Netherlands, B.P.Ahrens@tudelft.nl; Ambroise Lafont, LIX, Ecole Polytechnique, Palaiseau, France and Inria, Palaiseau, France, ambroise.lafont@lix.polytechnique.fr; Thomas Lamiaux, Nantes University, Nantes, France and Inria, Nantes, France, thomas.lamiaux@etu.univ-nantes.fr.

ACM 2475-1421/2025/8-ART258
https://doi.org/10.1145/3747527

1.1 Initial Semantics for Variable Binding

Initial semantics [18, 19, 32] aims to characterize an inductive structure — e.g., the abstract syntax of a language — as an initial object in a suitable category. The property of being an initial object amounts to a recursion principle that can be used to specify maps — e.g., translations of the abstract syntax — into objects of the same category. An ubiquitous instance of this principle is Haskell's `fold` operator, which exploits the fact that the type `[a]` of lists over base type `a`, together with the constructors `[] :: [a]` and `(:) :: a -> [a] -> [a]`, is the initial object in a suitable category of "list-algebras". Specifically, a list-algebra consists of a type `b` (the "carrier") and two functions, `nil :: b` and `cons :: a -> b -> b`, and a morphism between two list-algebras is a function on the underlying carriers that preserves `nil` and `cons` in a suitable way.

In this work, we are concerned with more complicated languages featuring *variable binding*, such as the lambda calculus. When implementing and formalizing languages such languages, the question of how to model variable binding and substitution arises. The chosen answer to this question leads to different notions of renaming and substitution, and to different categories of "algebras" where, in particular, the types of the "carriers" differ.

Consider the implementation[1] of the untyped lambda calculus using natural De Bruijn variables to represent variables as given in Listing 1. This definition is also known as *unscoped* syntax, since free and bound variables are not directly specified by the definition of the syntax, but require a separate definition on top of it.

```
Inductive LC : Set :=
| Var : ℕ → LC
| App : LC → LC → LC
| Abs : LC → LC
```

Listing 1. Unscoped Lambda Calculus

Taking an approach analogous to the list-algebras as before, we could define a LC-algebra to be a carrier set X : **Set** together with suitable functions $var : \mathbb{N} \to X$, $app : X \to X \to X$, and $abs : X \to X$ compatible with substitution. With a suitable notion of morphism of LC-algebras, the abstract syntax of the lambda calculus constitutes the initial LC-algebra.

Another possible formalization of the lambda calculus is to use *well-scoped* syntax, also known as *intrinsic* syntax, with finite or infinite contexts as given in Listings 2 and 3. In this implementation, the context of variables is explicit and directly part of the typing of the meta-language. The advantage of this approach is that all terms are well-scoped by construction.

```
Inductive LC : ℕ → Set :=
| Var : ∀n : ℕ, Fin n → LC n
| App : ∀n : ℕ, LC n → LC n → LC n
| Abs : ∀n : ℕ, LC (n + 1) → LC n
```

Listing 2. Well-scoped Lambda Calculus with Finite Contexts

```
Inductive LC : Set → Set :=
| Var : ∀X : Set, X → LC X
| App : ∀X : Set, LC X → LC X → LC X
| Abs : ∀X : Set, LC (X + 1) → LC X
```

Listing 3. Well-scoped Lambda Calculus with Infinite Contexts

A LC-algebra for well-scoped syntax with infinite contexts consists of a functor $F : \mathbf{Set} \to \mathbf{Set}$ associating well-scoped terms to contexts, together with natural transformations $var : 1 \to F$, $app : F \times F \to F$, and $abs : F \circ \text{option} \to F$ modeling the constructors. Here option : $\mathbf{Set} \to \mathbf{Set}$ maps a set X to the set $X + 1$. With a suitable notion of morphism of LC-algebras, the abstract syntax

[1] These examples are given in Rocq syntax, but that is irrelevant in the remainder of the paper; in particular, no knowledge of Rocq is required.

of the lambda calculus constitutes the initial LC-algebra. A LC-algebra for well-scoped syntax with finite contexts is similar except that it uses a functor $F : \mathbb{F} \to \mathbf{Set}$, where $\mathbb{F}$ is the category of finite cardinals (whose objects are natural numbers).

All the mentioned approaches have been used to prove initial semantics results for syntax with variable binding. Specifically, Fiore et al. [16] used the category $[\mathbb{F}, \mathbf{Set}]$ of functors from the category of finite cardinals and any map between them to sets; Hirschowitz and Maggesi [25] used the category $[\mathbf{Set}, \mathbf{Set}]$, thus modelling languages as monads on sets; Hirschowitz et al. [23] studied abstract syntax using De Bruijn variables.

All these semantic frameworks account for substitution from the start using suitable monoidal structures, as we will see in the next subsection. This is in contrast with the approach to variable binding based on nominal sets [17], which we do not investigate in this paper.

1.2 Initial Semantics for Substitution

Modeling the untyped lambda calculus as an initial algebra only captures the abstract syntax. It does not model substitution and how (variable binding) constructors interact with it, even though that is the core characteristic of languages with variable binding.

Linton [33] discovered that simultaneous substitution equips the lambda calculus as given in Listing 3 with the structure of a monad. This insight was rediscovered later in different contexts, see, for instance, the work by Bellegarde and Hook [9], Bird and Paterson [10], and Altenkirch and Reus [8]. Monads are equivalently monoids for composition: in this respect, the inclusion of variables into terms provides the unit of the monad, and the flattening of terms with terms as variables constitutes the monad multiplication. This is also equivalent to monads in terms of "extension systems" whose `bind` operation directly corresponds to simultaneous substitution; see, for instance, Section 3, Exercise 12 of the book by Manes [35].

We can extend the definition of LC-algebra, in this approach, as follows: the carrier is a monad (not just a functor) on the category of sets, and the operations *app* and *abs* should preserve the substitution structure of the monad in a suitable way. Expressed concisely, the carrier of a LC-algebra is a monoid in the monoidal category $([\mathbf{Set}, \mathbf{Set}], \circ, I)$, where the monoidal product is given by composition $G \circ F$ of endofunctors, with the unit I being the identity functor.

The other approaches can be upgraded, in a similar way, to incorporate substitution for different choices of the monoidal category. In the finite-context approach, a carrier can be defined to be a monoid in the monoidal category $([\mathbb{F}, \mathbf{Set}], \otimes, J)$, for a suitable monoidal product $\otimes$ (involving a left Kan extension). In the De Bruijn approach, a carrier can be defined to be a relative monad on the inclusion $\mathbf{B}\mathbb{N} \to \mathbf{Set}$ where $\mathbf{B}\mathbb{N}$ is the full subcategory of sets consisting of $\mathbb{N}$ as its single object [23].

Abstractly, to all of these different ways of modelling syntax corresponds a suitable *monoidal category* modelling contexts, variable binding, and substitution. Given a notion of signature that specifies the constructors of a language, this choice of a monoidal category leads to a corresponding notion of "algebra" or "model" of a signature. These different notions of model differ in the carriers, modelling the language-independent details on how contexts are modelled, but contain the same language-specific information, in particular, the types of the language constructors.

1.3 Contributions

Most lines of work on initial semantics of which we are aware start by fixing a choice of base monoidal category, e.g. to study a particular class of languages or semantics like reduction rules, and keep it fixed throughout the paper. In this paper, we do the opposite and study how to *change* the monoidal category modelling contexts, and its applications. Specifically, given two base monoidal categories satisfying a relationship like an adjunction or an equivalence, how do the resulting categories of models relate?

To do so, we identify that module signatures, once parametrized by the choice of monoidal category, form a 2-category, and that the category of models can then be computed by a 2-functor. More specifically:

(1) We identify that, once parametrized by monoidal categories, module signatures form a 2-category **ModSig**.
(2) There is a 2-functor **ModSig** $\to$ Cat that computes the category of models of a signature (Theorem 4.13).

As a consequence, functors, adjunctions, or equivalences, between different kinds of contexts, lift to functors, adjunctions, or equivalences between the resulting categories of models. We then use the abstract machinery to establish old and new concrete results about the relationship between models of syntax using different monoidal categories.

We first recover, in a general and systematic way, links between the models of the different implementations of abstract syntax like the lambda calculus mentioned above.

(1) We define a 2-category **BindMonCat** of monoidal categories with enough structure to interpret any binding signature. We then show that given binding signature S, there is a 2-functor Sem_S : **BindMonCat** $\to$ **ModSig** (Proposition 5.15).
(2) Using this result, we construct a coreflection between the models of well-scoped syntax with finite contexts (using $[\mathbb{F}, \mathbf{Set}]$) and the one using infinite contexts (using $[\mathbf{Set}, \mathbf{Set}]$), cf Theorem 6.5. Hence, we recover a result which previously only had a long proof, fully constructed by hand by Zsidó [43].
(3) Similarly, we prove that models of unscoped syntax (using $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$) and of well-scoped syntax with finite contexts (using $[\mathbb{F}, \mathbf{Set}]$) are equivalent once restricted to well-behaved ones (Theorem 7.15), which only had been proven by hand by Hirschowitz et al. [22].

We then study how changing the base monoidal categories enables us to build more general recursion principles for simply-typed languages, as the one generated by initiality is by default limited to a fixed type system.

(1) We define a category STSig of simply-typed binding signatures, and show there is a 1-functor **STSigModel**: $\mathrm{STSig}^{\mathrm{op}} \to$ **ModSig** $\to$ Cat computing the category of models (Theorem 8.7). As a consequence, a morphism between simply-typed signatures over different type systems lifts to a functor between their categories of models.
(2) Building upon **STSigModel**, we recover Ahrens' category of models [4] in Proposition 8.15 using the *Grothendieck construction* [27, Definition 1.10.1], hence providing a new insight on this framework specifically tailored for translating across type systems. This category gathers models over different object types in one "large" category; translations between languages over different types can thus be viewed as morphisms in this category.

1.4 Synopsis

In Section 2, we start by giving a brief introduction to initial semantics, on which we rely in the remainder of the paper. We also give a brief introduction to the very few notions of 2-category theory needed to understand this paper in Section 3. Note that we only use *strict* 2-category theory, where axioms about 1-cells are expressed modulo equality, not modulo invertible 2-cells. We show in Section 4 that initial semantics has a 2-categorical structure, and that models can be computed by a 2-functor. We then define *binding-friendly* monoidal categories in Section 5 to leverage the 2-functoriality of models to relate the models of the different implementations of the untyped lambda calculus in Sections 6 and 7. We also leverage 2-functoriality of models to prove a generalized recursion principle for simply-typed syntax in Section 8. We discuss related work in Section 9, and conclude in Section 10.

2 A Short Introduction to Initial Semantics

In this section, we give a short introduction to initial semantics. It is necessarily terse and incomplete, and we refer to the work by Lamiaux and Ahrens [31] for details and additional examples.

In this section, we discuss initial semantics in a fixed monoidal category. Later, in Section 4, we give these definitions a 2-categorical structure depending on a 2-category of monoidal categories.

2.1 Abstracting Syntax and Substitution

Initial semantics requires a base category to model context and variable binding. To model syntax, it requires more than the structure of a mere category, it requires a *monoidal* category. In this section, we briefly recall the basic categorical definitions and sketch how they model syntax and substitution. For details on monoids, we refer to Chapter VII of the book by Mac Lane [34].

Recall that a monoidal category [34, VII.1] is a tuple $(\mathsf{C}, \otimes, I, \alpha, \lambda, \rho)$, where C is a category, $\_ \otimes \_ : \mathsf{C} \times \mathsf{C} \to \mathsf{C}$ is a bifunctor called the monoidal product, and $I : \mathsf{C}$ an object called the unit. Furthermore, we have natural isormorphisms α, λ, ρ as below – called the associator, and the left and right unitor – that satisfy the unit axiom and the pentagon axiom.

$$\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \cong X \otimes (Y \otimes Z) \qquad \lambda_X : I \otimes X \cong X \qquad \rho_X : X \otimes I \cong X$$

In the following, we will simply write $(\mathsf{C}, \otimes, I)$ or even C for a monoidal category, leaving the other components implicit. The main interests of monoidal categories is that syntax and simultaneous substitution can be modeled as *monoids* in particular monoidal categories.

Given a monoidal category $(\mathsf{C}, \otimes, I)$, a monoid on C [34, VII.3] is a tuple (R, μ, η) where R is an object of C and the *multiplication* $\mu : R \otimes R \to R$ and the *unit* $\eta : I \to R$ are morphisms of C satisfying the monoid laws. This forms a category $\mathsf{Mon}(\mathsf{C})$ together with morphism of monoids.

Example 2.1 ([8–10]). The lambda calculus $\mathsf{LC} : \mathbf{Set} \to \mathbf{Set}$ as defined in Listing 3, together with a suitable substitution operation, is a monad on $\mathbf{Set}$, that is, a monoid on $([\mathbf{Set}, \mathbf{Set}], \circ, I)$.

Monoids are not enough on their own to fully model languages with variable binding, as monoids do not capture constructors and their substitution structures. To do so, on top of monoids, we model constructors as morphisms of modules over monoids.

Definition 2.2 (Category of Modules over a Monoid). Given a monoid $R : \mathsf{Mon}(\mathsf{C})$, a (left) R-*module* is a tuple (M, p^M) where M is an object of C and $p^M : M \otimes R \to M$ is a morphism of C called *module substitution* that is compatible with the multiplication and the unit of the monoid:

$$\begin{array}{ccccc} (M \otimes R) \otimes R & \xrightarrow{\alpha_{M,R,R}} & M \otimes (R \otimes R) & \xrightarrow{M \otimes \mu} & M \otimes R \\ \downarrow{\scriptstyle p^M \otimes R} & & & & \downarrow{\scriptstyle p^M} \\ M \otimes R & & \xrightarrow[p^M]{\hspace{6em}} & & M \end{array} \qquad \begin{array}{ccc} M \otimes I & \xrightarrow{M \otimes \eta} & M \otimes R \\ & \searrow^{\rho_M} & \downarrow{\scriptstyle p^M} \\ & & M \end{array}$$

Given two modules (M, p) and (M', p') over R, a *module morphism* from (M, p) and (M', p') is a morphism $r : M \to M'$ of C commuting with the respective module substitutions:

$$\begin{array}{ccc} M \otimes R & \xrightarrow{r \otimes R} & M' \otimes R \\ \downarrow{\scriptstyle p} & & \downarrow{\scriptstyle p'} \\ M & \xrightarrow[r]{} & M' \end{array} \tag{1}$$

With composition and identity induced by that of the monoidal category C, modules over R and their morphisms form a category, which we call $\mathsf{Mod}(R)$.

Definition 2.3. Any monoid R induces a module over itself, also called R.

Example 2.4. The functor $X \mapsto \mathsf{LC}(X) \times \mathsf{LC}(X)$ yields a module over the monad LC of Example 2.1. The functor $X \mapsto \mathsf{LC}(X + 1)$ yields a module over the monad LC of Example 2.1. For both, the module substitution is given by parallel substitution.

The constructors $\mathsf{app} : \mathsf{LC} \times \mathsf{LC} \to \mathsf{LC}$ and $\mathsf{abs} : \mathsf{LC} \circ \mathsf{option} \to \mathsf{LC}$ are morphisms of LC-modules. For both, Diagram 1 spells out the commutation of app and abs with substitution, that is, $(\mathsf{app}(M, N))[f] = \mathsf{app}(M[f], N[f])$ and $(\mathsf{abs}(M))[f] = \mathsf{abs}(M[f \uparrow])$, where $f \uparrow$ is a suitable lift of the substitution function f, necessary when descending under a variable binder.

2.2 Signatures and Models

Initial semantics aims to provide a generic framework for studying syntax, therefore, it requires a generic notion of signatures and of models. As constructors are represented by morphisms of modules over monoids, we would like to take this as our specification. However, modules are defined over specific monoids, and we need signatures to specify languages for any choice of monoids. To express this, we use the total category of modules over monoids.

Definition 2.5 (Total category of modules). There is a category of modules over monoids, denoted $\mathsf{Mod}(\mathsf{C})$. Its objects are tuples (R, M) where $R : \mathsf{Mon}(C)$ is a monoid, and $M : \mathsf{Mod}(R)$ a module over it. Its morphisms $(R, M) \to (R', M')$ are tuples (f, r) where $f : R \to R'$ is a morphism of monoids and $r : M \to f^*M'$ a morphism of R-monoids. Here, $f^*(M') : \mathsf{Mod}(R)$ denotes the R-module with underlying object M' and module multiplication $M' \otimes R \xrightarrow{M' \otimes f} M' \otimes R' \xrightarrow{p^{M'}} M'$.

Remark 2.6. The category $\mathsf{Mod}(C)$ is usually denoted by $\int_{X:\mathsf{Mon}(C)} \mathsf{Mod}(X)$ as it is the total category for the 1-functor $\mathsf{Mod} : \mathsf{Mon}(C)^{\mathsf{op}} \to \mathsf{Cat}$. In this work, we denote it instead by $\mathsf{Mod}(C)$ for brevity.

It is then possible to specify languages by functors $\mathsf{Mon}(C) \longrightarrow \mathsf{Mod}(C)$ that return a module over its input.

Definition 2.7 (Signature). A *module signature* is a functor $\Sigma : \mathsf{Mon}(C) \longrightarrow \mathsf{Mod}(C)$ such that $\pi_1 \circ \Sigma = \mathrm{id}$, that is Σ is a section of π_1. In other words, such that the following diagram commute:

$$\begin{array}{ccc} \mathsf{Mon}(\mathsf{C}) & \xrightarrow{\Sigma} & \mathsf{Mod}(\mathsf{C}) \\ & \searrow\!\!= \quad \swarrow_{\pi_1} & \\ & \mathsf{Mon}(\mathsf{C}) & \end{array}$$

Example 2.8 (Signature of LC). The signature of the lambda calculus, considered in the monoidal category $[\mathbf{Set}, \mathbf{Set}]$, maps a monad R to the R-module $R \times R + R \circ \mathsf{option}$, thus specifying the source of the two domain-specific constructors app and abs.

Definition 2.9. There is a trivial signature $\Theta : \mathsf{Mon}(C) \to \mathsf{Mod}(C)$ that maps a monoid (R, η, μ) to the module (R, μ).

Example 2.10 (Signature of LC, continued). Using the trivial signature Definition 2.9 to represent a unary operation, and writing $\Theta^{(1)} : R \mapsto R \circ \mathsf{option}$, we can write the signature of the lambda calculus from Example 2.8 as $\Theta \times \Theta + \Theta^{(1)}$.

To specify constructors, we need two signatures: one for the input and one for the output. Since we always return regular terms without additional free variables, and to simplify the framework, we fix the output signature to be the trivial signature Θ. Thus, we only need one signature Σ to specify the constructors. We are now ready to define the models of a module signature.

Definition 2.11 (Category of models of a signature). A *model* of a module signature Σ consists of a pair (R, m) of a monoid R together with a morphism $m : \Sigma(R) \to R$ of R-modules.

A morphism $f : (R, m) \to (R', m')$ of models is a morphism of monoids $f : R \to R'$ that commutes with the module morphisms:

$$\begin{array}{ccc} \Sigma(R) & \xrightarrow{\ r\ } & R \\ {\scriptstyle \Sigma(f)}\downarrow & & \downarrow{\scriptstyle f} \\ f^*\Sigma(R') & \xrightarrow[f^*r']{} & f^*R' \end{array}$$

Models of Σ, and their morphisms, form the category **Model**(Σ) of models of Σ.

This provides a generic and abstract framework to model syntax and its properties. Initial semantics is then concerned with proving initiality theorems, that is, theorems asserting the existence of an initial model, under hopefully simple conditions on the monoidal category and the module signature specifying a language. The existence of this model then ensures the existence of an implementation, and the initiality provides it with a recursion principle respecting the substitution structure of the language. In this work, we are not concerned with initiality theorems, but such a theorem and a discussion about them can be found, e.g., in a paper by Lamiaux and Ahrens [31].

3 Overview of 2-Category Theory

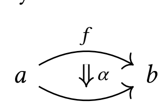
In this section we recall the few notions of 2-categories needed to understand the paper, and state some results used later. Lack [30] provides an extensive introduction to 2-categories, as does Mac Lane [34, XII.3-4].

Note that our 2-categories and related structures are all "*strict*". This means that they satisfy laws about 1-cells up to equality, not only up to (invertible) 2-cell; see also Remark 3.4.

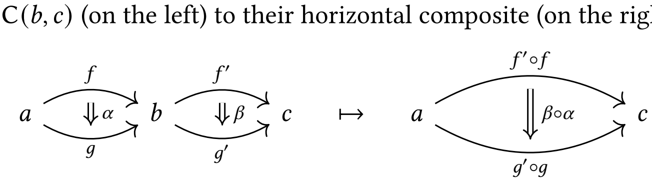
A 2-category is a category *enriched* in the cartesian category Cat of categories. Equivalently, a 2-category is a 1-category with a "system of 2-cells or 'maps' which can be composed in two different but commuting categorical ways", to quote Mac Lane [34, XII.3].

Definition 3.1. A 2-category C consists of

(1) an underlying 1-category;
(2) for each pair of objects (a, b), a *hom-category* $\underline{\mathrm{C}(a, b)}$ whose underlying set of objects is $\mathrm{C}(a, b)$; we denote its composition by $\bullet$ and draw a morphism α from f to g as follows;



(3) an associative *horizontal composition* operation mapping morphisms α and β in hom-categories $\mathrm{C}(a, b)$ and $\mathrm{C}(b, c)$ (on the left) to their horizontal composite (on the right):



such that the horizontal composition of two identities is the identity, and such that the following equations hold: $1_{1_b} \circ f = f = f \circ 1_{1_a}$, and $(\beta' \circ \beta) \bullet (\alpha' \circ \alpha) = (\beta' \bullet \alpha') \circ (\beta \bullet \alpha)$. Here, 1_{1_x} denotes the identity morphism in $\mathrm{C}(x, x)$ on the identity 1_x on x.

We refer to the morphisms of the underlying 1-category C also as 1-cells or 1-morphisms, and to the composition of that 1-category as 1-composition. Similarly, for each pair of objects a, b, we refer to the morphisms of the category $\underline{\mathrm{C}(a, b)}$ as 2-cells or 2-morphisms, and to the composition of that category as vertical composition.

Example 3.2. Our prototypical example of 2-category is given by the 2-category of categories, functors, and natural transformations. This 2-category has, as its collections of objects, all (small) categories. For given categories C and D, their hom-category is given by the functor category $[\mathrm{C}, \mathrm{D}]$ of functors from C to D, and natural transformations between them.

Given a 2-category C, we write C_0 also for its underlying collection of objects; we write $\mathrm{C}(a, b)$ for the collection of 1-cells or 1-morphisms from a to b, and $\mathrm{C}(f, g)$ for the collection of 2-cells from f to g. We write 1-composition of $f : \mathrm{C}(a, b)$ and $g : \mathrm{C}(b, c)$ as $g \circ f$ or gf. We also write αf for the horizontal composition $\alpha \circ 1_f$ — this is known as "whiskering" — and similar for whiskering in the other order. A 2-cell $\alpha : f \Rightarrow g$ is an isomorphism if there is a 2-cell $\alpha' : g \Rightarrow f$ such that $\alpha \bullet \alpha' = 1_g$ and $\alpha' \bullet \alpha' = 1_f$.

Example 3.3. Rephrasing Example 3.2 in terms of Definition 3.1, the 2-category of categories has, as underlying 1-category, the 1-category of (small) categories. The 2-cells from functor $F : \mathrm{C} \longrightarrow \mathrm{D}$ to functor $G : \mathrm{C} \longrightarrow \mathrm{D}$ are given by the natural transformations $\alpha : F \Rightarrow G$.

Remark 3.4. The 1-composition of a 2-category satisfies the usual categorical laws *strictly*, that is, up to equality. This is in contrast to the weaker notion of a *bi*category [34, XII.6], where the 1-categorical laws only hold up to an invertible 2-cell. The notion of bicategory is thus more general than that of a 2-category; we do not have any use for that generality and work exclusively with 2-categories.

We continue the axiomatization of our main example of 2-categories, the 2-category of categories (Example 3.2). In particular, the different notions of maps between categories, in particular, adjunction, coreflection, and equivalence, directly generalize to 2-categories:

Definition 3.5 (Adjunction in a 2-category). Let C be a 2-category. An adjunction in C consists of two 1-morphisms $l\colon a \longrightarrow b$ and $r\colon b \longrightarrow c$ with two 2-cells $\eta\colon \mathrm{id}_c \longrightarrow r \circ l$ and $\varepsilon\colon l \circ r \longrightarrow \mathrm{id}_a$, called the unit and the counit, subject to the usual triangle equalities $\varepsilon f \bullet f\eta = 1_f$ and $g\varepsilon \bullet \eta g = 1_g$. We write $l : a \dashv b : r$ or $l \dashv r$ to denote this situation.

Definition 3.6 (Coreflection in a 2-category). A *coreflection* is an adjunction $l \dashv r$ such that the unit η is an isomorphism.

Definition 3.7 (Equivalence in a 2-category). An equivalence is an adjunction $l \dashv r$ such that both η and ε are isomorphisms.

Definition 3.8. Given a 2-category C and $f : \mathrm{C}(a, b)$, we define the functor

$$f_* : \underline{\mathrm{C}(x, a)} \to \underline{\mathrm{C}(x, b)}$$

given on objects as $g \mapsto fg$ and on morphisms as $\alpha \mapsto f\alpha$.

We call f *fully faithful* if f_* is a fully faithful functor.

Lemma 3.9. *An adjunction $l \dashv r$ in* Cat *is a coreflection if and only if l is full and faithful.*

We now review *morphisms of 2-categories*, to which we refer to as *2-functors*. Intuitively, a 2-functor from C to D is a 1-functor between the underlying 1-categories that also acts on the 2-cells, preserving their source and target.

Definition 3.10 (2-functor). Given 2-categories C and D, a *2-functor* $F : \mathsf{C} \to \mathsf{D}$ from C to D consists of functions on the objects, morphisms, and 2-cells from C to D that preserve source and target; in detail,

(1) $F_0 : \mathsf{C}_0 \to \mathsf{D}_0$;
(2) $F_1 : \mathsf{C}(a, b) \to \mathsf{D}(F_0 a, F_0 b)$;
(3) $F_2 : \mathsf{C}(f, g) \to \mathsf{D}(F_1 f, F_1 g)$.

In practice, we omit the indices, since they will be clear from the context. Furthermore, identities and compositions are preserved by the functions on morphisms and 2-cells: $F(1_a) = 1_{Fa}$, $F(1_f) = 1_{Ff}$, $F(gf) = (Fg)(Ff)$, and $F(\beta\alpha) = (F\beta)(F\alpha)$.

The important result for the rest of the paper is the following:

Proposition 3.11. *2-functors preserve adjunctions, coreflections, and equivalences.*

4 A 2-Categorical Perspective on Initial Semantics

In this section, we show how the category of models of a signature can be computed by a 2-functor from the 2-category **ModSig** of module signatures (Theorem 4.13). Leveraging the 2-functoriality of this construction, we can then study changes in the base monoidal category (Sections 5 to 7) and develop a generalized recursion principle for simply typed syntax (Section 8).

Signatures can be formulated as particular "(vertical) inserter diagrams" (Definition 4.1) In Section 4.1, we construct a 2-functor from the 2-category of inserter diagrams to the 2-category Cat; this 2-functor computes the category of models of an inserter diagram.

In Section 4.2 we construct another 2-functor, from module signatures to vertical inserters. By composing these two 2-functors, we can then compute the category of models of a module signature, in a 2-functorial way.

4.1 Categories of Models as Vertical Inserters

Here, we first show in Section 4.1.1 how the module signatures and models reviewed in Section 2 can be formulated as inserter diagrams and vertical inserters, respectively. We study the 2-categorical structure of inserter diagrams in Section 4.1.2. We then use these intermediate notions and their 2-categorical structure, in Section 4.1.3, to construct a 2-functor that computes the category of models from an inserter diagram.

4.1.1 Abstracting Signatures and Models. We show how module signatures are subsumed by "vertical inserter diagrams", and their categories of models by inserters of inserter diagrams.

A module signature Σ induces the following diagram with $\pi_1 \circ \Sigma = \mathrm{id} = \pi_1 \circ \Theta$:

$$\mathsf{Mon}(C) \overset{\Sigma}{\underset{\Theta}{\rightrightarrows}} \mathsf{Mod}(C) \xrightarrow{\pi_1} \mathsf{Mon}(C)$$

This structure naturally generalizes as inserter diagrams (Σ, Θ, π_1):

Definition 4.1. A vertical *inserter diagram* consists of functors $F, G : A \to B$ and a functor $p : B \to C$ such that $p \circ F = p \circ G$ as below:

$$A \overset{F}{\underset{G}{\rightrightarrows}} B \xrightarrow{p} C$$

In what follows, we may omit the adjective "vertical" and simply speak of "inserter diagrams".

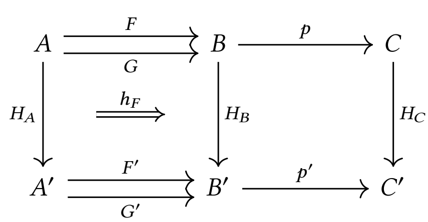
Proposition 4.2. *Inserter diagrams are exactly functors from the walking co-fork* $\rightrightarrows\!\longrightarrow$ *to* Cat.

Given a signature Σ, its category of models then corresponds to the *vertical inserter* of the inserter diagram (Σ, Θ, π_1) associated to Σ:

Definition 4.3. The *vertical inserter* of an inserter diagram (F, G, p) is the category whose objects are pairs (a, f) where a is an object of A and f is a morphism $f : Fa \to Ga$ such that $p(f)$ is the identity morphism.

4.1.2 2-Categorical Structure of Inserter Diagrams. The interest of abstracting module signatures by inserter diagrams is that inserter diagrams have a 2-categorical structure as described below. This will then enable us to define a 2-functor computing the category of models, which we have abstracted as vertical inserters.

Definition 4.4. A *morphism* between two inserter diagrams (F, G, p) and (F', G', p') is given by three functors $(A \xrightarrow{H_A} A', B \xrightarrow{H_B} B', C \xrightarrow{H_C} C')$ such that $H_C \circ p = p' \circ H_B$ and $H_B \circ G = G' \circ H_A$, together with a natural transformation $F' \circ H_A \xrightarrow{h_F} H_B \circ F$ as summarised in the following diagram. Moreover, $p' \circ h_F$ must the identity 2-cell on $p' \circ H_B \circ F = p' \circ F' \circ H_A$.



Remark 4.5. This notion of morphism may seem ad-hoc, but it naturally appears when building the 2-functor computing the category of models. It also enables us to understand vertical inserters in terms of *marked limits*, as explained in Remark 4.8 below.

Let us now complete the definition of the 2-category of vertical inserter diagrams.

Definition 4.6. We define the 2-category **VInsDiag** of vertical inserter diagrams, where objects and morphisms are as above, and a 2-cell between (H_A, H_B, H_C, h_F) and (H'_A, H'_B, H'_C, h'_F) is given by three natural transformations $(H_A \xrightarrow{h_A} H'_A, H_B \xrightarrow{h_B} H'_B, H_C \xrightarrow{h_C} H'_C)$ satisfying coherence conditions, making them modifications [28, 4.4] between the lax transformations induced by Proposition 4.2.

4.1.3 Models from Inserter Diagrams, 2-Functorially. The main interest of the 2-categorical structure of inserter diagrams is that vertical inserters, abstracting models, can be computed by a 2-functor. Consequently, adjunctions or equivalences in the 2-category **VInsDiag** lift to adjunctions or equivalences between vertical inserters, that is, between categories of models.

Theorem 4.7. *The 2-functor* **CstDiag**: Cat $\to$ **VInsDiag** *mapping a category* C *to the trivial diagram* C $\rightrightarrows$ C $\to$ C *has a right adjoint* **VIns**, *computing the vertical inserter — that is, the category of models — of a given diagram.*

$$\mathbf{VInsDiag} \xrightarrow{\mathbf{VIns}} \mathsf{Cat}$$

Remark 4.8. Theorem 4.7 exploits the characterisation of vertical inserters as 2-dimensional limits, and is analogous to the fact that a category C has limits of shape D precisely when the functor C $\to$ [D, C] mapping c to the constant functor equal to c has a right adjoint, which computes the limit. Indeed, if we take $\mathsf{C} = \mathsf{Cat}$ and D the walking cofork $\rightrightarrows\rightarrow$, we almost get the same situation, except that we need to replace the strict morphisms of $[\rightrightarrows\rightarrow, \mathsf{Cat}]$ by *lax natural transformations* [28, Chapter 4.2] such that the only non-strict component of the natural transformation is the one associated with top left arrow. Any other choice of "laxity" in the notion of morphism of inserter diagrams would not yield a right adjoint.

This adjunction holds more generally for *marked limits*, or *cartesian quasi-limits* in the terminology of Gray [20, I.7.9.1.(iii)].

Given a module signature Σ, there is a projection $\pi_1 : \mathbf{Model}(\Sigma) \to \mathsf{C}$ mapping models (R, m) to R. This extends to a 2-natural transformation by requiring it to be compatible with the projection for inserter diagrams as defined below. Naturality is important, as for instance, it ensures that the functor induced between the categories of models is compatible with the projection to the category of monoids.

Theorem 4.9. *There is a 2-functor $\pi_1 : \mathbf{VInsDiag} \to \mathsf{Cat}$ mapping an inserter diagram of the form* $A \overset{F}{\underset{G}{\rightrightarrows}} B \xrightarrow{p} C$ *to the category A.*

4.2 Models from Module Signatures, 2-Functorially

We have shown how the categories of models of a module signatures abstract as vertical inserters of inserter diagrams, and that computing models of such signatures is 2-functorial. However, in practice, we want to lift adjunctions or equivalences directly from monoidal categories and module signatures, not from inserter diagrams. To make this possible, we now precompose the 2-functor $\mathbf{VIns} : \mathbf{VInsDiag} \to \mathsf{Cat}$ by a 2-functor $\mathbf{PModDiag} : \mathbf{ModSig} \to \mathbf{VInsDiag}$ mapping a module signature to the associated inserter diagram.

To define a 2-category **ModSig** whose objects are pairs (C, Σ) of a monoidal category C and a module signature Σ over it, we first need to show that lax monoidal functors lift to monoids and modules.

Proposition 4.10 ([43, §§ 4.5-4.6]). *A lax monoidal functor $F : \mathsf{C} \to \mathsf{D}$ lifts to a functor between categories of monoids $F : \mathsf{Mon}(\mathsf{C}) \to \mathsf{Mon}(\mathsf{D})$ and modules $F : \mathsf{Mod}(\mathsf{C}) \to \mathsf{Mod}(\mathsf{D})$. Those liftings are compatible with composition of lax monoidal functors, identity functors, projections to monoids and the tautological module signature. Similarly, a monoidal transformation between lax monoidal functors induces a compatible natural transformation between the induced functors between monoids and modules.*

We can now define the 2-category of module signatures.

Definition 4.11. We define the 2-category **ModSig** of module signatures as follows:

- an object consists of a monoidal category C and a module signature $\Sigma \colon \mathsf{Mon}(\mathsf{C}) \to \mathsf{Mod}(\mathsf{C})$;
- a 1-cell between (C, Σ) and (D, Σ') consists of a monoidal functor $F \colon \mathsf{C} \to \mathsf{D}$ and a natural transformation $\alpha \colon \Sigma' \circ F \to F \circ \Sigma$, which is the identity when composed with the projection to monoids $\pi_1 : \mathsf{Mod}(\mathsf{C}) \to \mathsf{Mon}(\mathsf{C})$. Here, F corresponds to the lifting of F to monoids and modules as in Proposition 4.10.
- a 2-cell between (F, α) and (G, β) consists of a monoidal natural transformation $\gamma \colon F \to G$ such that the following diagram commutes.

$$\begin{array}{ccc} \Sigma' \circ F & \xrightarrow{\alpha} & F \circ \Sigma \\ {\scriptstyle \Sigma' \circ \gamma}\downarrow & & \downarrow{\scriptstyle \gamma \circ \Sigma} \\ \Sigma' \circ G & \xrightarrow[\beta]{} & G \circ \Sigma \end{array}$$

With this definition, we can now show that the assignment of the inserter diagram to a module signature is 2-functorial, and hence that the category of models is 2-functorial in module signatures.

THEOREM 4.12. *There is a 2-functor* **PModDiag**: **ModSig** → **VInsDiag** *mapping* (C, Σ) *to the following inserter diagram:*

$$\mathrm{Mon}(\mathrm{C}) \underset{\Theta}{\overset{\Sigma}{\rightrightarrows}} \mathrm{Mod}(\mathrm{C}) \xrightarrow{p} C$$

THEOREM 4.13. *By composing the 2-functors introduced above, we get a 2-functor* **Model** : **ModSig** → Cat *that computes the category of models of any module signature:*

$$\mathbf{ModSig} \xrightarrow{\mathbf{PModDiag}} \mathbf{VInsDiag} \xrightarrow{\mathbf{VIns}} \mathrm{Cat}$$

Though not directly related to the main topic of this paper, this functor can be used to give a 2-categorical perspective on the initiality theorem [31, Theorem 4.23].

Remark 4.14. The 2-functor of Theorem 4.13 can be restricted to the full sub-2-category PSigStr of module signatures induced by *signatures with strength* satisfying the hypotheses of the initiality theorem [31, Theorem 4.23]. Such a signature is given by an ω-cocontinuous functor specifying the operations, and a strength used to build the monoid structure. The image of any such signature by this functor is a category which has an initial object.

$$\mathrm{PSigStr} \longrightarrow \mathbf{ModSig} \xrightarrow{\mathbf{Model}} \mathrm{Cat}$$

5 Binding-Friendly Monoidal Categories

Binding signatures can be used to specify untyped languages involving variable bindings. We are interested in relating models of binding signatures over *different* base categories, such as $[\mathbb{F}, \mathbf{Set}]$ and $[\mathbf{Set}, \mathbf{Set}]$. We could do so by first relating the module signatures generated by a binding signature for each pair of base categories of our interest (e.g., exhibiting an adjunction in **ModSig**), and then applying 2-functoriality to get the same relation on the categories of models, by Proposition 3.11.

Instead, we follow another path that abstracts and factors a large amount of this work: we show that binding signatures can be "interpreted" in any monoidal category with enough structure, which we call "binding-friendly" monoidal categories, and that this interpretation is 2-functorial. Consequently, it suffices to prove that two binding-friendly monoidal categories are related in order to conclude that their respective models of binding signatures are also related.

In this way we can recover, as simple consequences and generically, results that were previously proven in an ad-hoc manner. Specifically, in Section 6, we show that there is a coreflection between models in $[\mathbb{F}, \mathbf{Set}]$ and models in $[\mathbf{Set}, \mathbf{Set}]$, and in Section 7 that there is an equivalence between suitable full subcategories of models in the De Bruijn setting [23] (see Definition 7.6 below about *De Bruijn monads*) and $[\mathbb{F}, \mathbf{Set}]$.

5.1 Binding Arities in a Generic Monoidal Category

A binding signature specifies a syntax generated by operations, potentially binding variables in each of their arguments.

Definition 5.1 ([1]). A *binding arity* is a list of natural numbers. A *binding signature* is a family $([n_1, \ldots, n_{p_o}])_{o \in O}$ of binding arities.

Each binding arity specifies a language constructor. The length of the list is the number of arguments, and the i-th element is the number of variables bound by the i-th argument.

Example 5.2. The untyped lambda calculus with constructors app and abs can be specified by the binding signature $([0, 0], [1])$. The corresponding module signature is the signature $\Theta \times \Theta + \Theta^{(1)}$ of Example 2.10.

A binding signature $((n_1, \ldots, n_{p_o}))_{o \in O}$ can be interpreted in $[\mathbf{Set}, \mathbf{Set}]$ as the module signature below [24]. Each summand of the coproduct represents a constructor, and the elements of the product representing its arguments, the module signature $\Theta^{(n_i)}$ representing the binding of n_i variables. This uses the trivial signature (Definition 2.9), and the n-th *derivative* of the module signature Σ, adding n fresh variables to contexts as $\Sigma^{(n)}(X)(\Gamma) := \Sigma(X)(\Gamma + n)$.

$$\coprod_{o \in O} \prod_{i=1}^{p_o} \Theta^{(n_i)}$$

In general, to interpret a binding signature as a module signature in a monoidal category, one has to be able to build coproducts, finite products, the trivial signature Θ and the derivative of a module signature. The module signature Θ already exists for all monoidal categories. Furthermore, module signatures inherit their limits and colimits from the base category C provided that for all Z precomposition by $\_ \otimes Z$ preserves them. This is a consequence of the following result.

Theorem 5.3. *Given a monoidal category* C*, the category of module signatures is monadic over* $[\mathsf{Mon}(\mathsf{C}), \mathsf{C}]$*, the monad mapping* Σ *to* $\Sigma(-) \otimes -$*.*

Proof. First, we check that $\Sigma(-) \otimes -$ is indeed a monad. The unit at Σ is the natural family of morphisms $\Sigma(X) \xrightarrow{\rho} \Sigma(X) \otimes I \xrightarrow{\Sigma(X) \otimes e} \Sigma(X) \otimes X$, where $e \colon I \to X$ is the unit of the monoid X.

The multiplication at Σ is the natural transformation $(\Sigma(X) \otimes X) \otimes X \xrightarrow{\alpha} \Sigma(X) \otimes (X \otimes X) \xrightarrow{\Sigma(X) \otimes m} \Sigma(X) \otimes X$, where $m \colon X \otimes X \to X$ is the multiplication of the monoid X. It is easy to check that the monad laws hold.

An algebra consists of a functor $\Sigma \colon \mathsf{Mon}(\mathsf{C}) \to \mathsf{C}$ and a natural family of morphims $x \colon \Sigma(X) \otimes X \to X$ and the algebra laws show that this provides a X-module structure on $\Sigma(X)$. We then get a functor from algebras to $[\mathsf{Mon}(\mathsf{C}), \mathsf{Mod}(\mathsf{C})]$; this functor is an isomorphism. □

Proposition 5.4. *Limits in the category of module signatures are computed pointwise in C, and if* $- \otimes X$ *preserves colimits of shape* D *for all* X*, then colimits of shape* D *are also computed pointwise.*

Proof. By Theorem 5.3, the functor from module signatures to the category $[\mathsf{Mon}(\mathsf{C}), \mathsf{C}]$ is monadic, and thus creates limits by [40, Theorem 5.6.5]. Because limits are computed pointwise in a functor category, we get the desired result for limits.

As for colimits of shape D, the involved monad $\Sigma \mapsto \Sigma(-) \otimes -$ preserves them because colimits are computed pointwise in a functor category. We conclude by [40, Theorem 5.6.5]: a monadic functor creates any shape of colimit that is preserved by the monad. □

It remains to interpret the derivative of a module signature in a generic monoidal category. In the case of $[\mathbf{Set}, \mathbf{Set}]$, this interpretation is given by precomposition with $- + 1$, which has the following characterisation.

Lemma 5.5. *Denoting the endofunctor* $- + 1$ *by* $O \colon \mathbf{Set} \to \mathbf{Set}$*, the endofunctor* $- \circ O$ *on* $[\mathbf{Set}, \mathbf{Set}]$ *is right adjoint to* $- \times \mathrm{Id}$*.*

Proof. By [7, Proposition 3], since the identity endofunctor is, using their notations, $[\![1 \triangleleft 1]\!]$. □

Indeed, as exponentiation by $(-)^{\mathrm{Id}}$ is a right adjoint to $- \times \mathrm{Id}$, and adjoints are unique up to isomorphism, we could define instead derivation by $\Sigma^{(1)} := \Sigma(-)^{\mathrm{Id}}$. This definition generalises to any monoidal category with exponentiable unit, that is such that $- \times I$ has a right adjoint.

Theorem 5.6. *Let* C *be a monoidal category with binary products which are preserved on the left by the tensor, such that* I *is exponentiable. If* Σ *is a module signature, then so is* $\Sigma(-)^I$*, with module structure* $\Sigma(X)^I \otimes X \to \Sigma(X)^I$ *defined as the transpose of the following morphism.*

$$(\Sigma(X)^I \otimes X) \times I \to (\Sigma(X)^I \otimes X) \times (I \otimes X) \cong (\Sigma(X)^I \times I) \otimes X \to \Sigma(X) \otimes X \to \Sigma(X)$$

Moreover, $\Sigma(-)^I$ *is actually the exponential* Σ^Θ *in the category of module signatures.*

Proof. See Appendix A. □

Notation 5.7. In the situation above, given a module signature Σ, we denote Σ^Θ by $\Sigma^{(1)}$, $(\Sigma^\Theta)^\Theta$ by $\Sigma^{(2)}$, and so on.

5.2 Binding-Friendly Monoidal Categories

The discussion in Section 5.1 leads us to introduce the notion of binding-friendly monoidal categories, which contains sufficient structure to interpret binding signatures. Such categories, and suitable morphisms and 2-cells between them, form a 2-category (Proposition 5.13).

Definition 5.8. A monoidal category C is said ***binding-friendly*** if it has

- finite products left-preserved by the tensor;
- non-empty coproducts left-preserved by the tensor;
- an exponentiable unit I.

Remark 5.9. It would be more convenient to ensure that empty coproducts exist, so that the empty binding signature induces a module signature like any other. However, we will encounter a situation in Section 7 where this is not the case (see Proposition 7.14 below).

Example 5.10. The category [**Set**, **Set**] is binding-friendly, by Lemma 5.5

Binding-friendly monoidal categories naturally inherit a 2-categorical structure from monoidal categories by requiring the extra structure to be preserved by the morphisms.

Definition 5.11. A binding-friendly functor F between two binding-friendly monoidal categories consists of a lax monoidal functor preserving finite products, small coproducts, and exponentiation by the unit, in the sense that for every object X of the domain, the canonical morphism $F(X^I) \to F(X)^I$, obtained as the transpose of the below morphism, is an isomorphism.

$$F(X^I) \times I \longrightarrow F(X^I) \times FI \cong F(X^I \times I) \longrightarrow F(X)$$

Definition 5.12. A transformation between binding-friendly functors is a monoidal transformation between the underlying monoidal functors.

Proposition 5.13. *This defines a 2-category* **BindMonCat** *of binding-friendly monoidal categories, which comes equipped with a forgetful 2-functor to the 2-category of monoidal categories.*

5.3 Interpreting Binding Signatures

The point of binding-friendly monoidal categories is that there is a canonical 2-functorial interpretation of binding signatures into them. Here, the existence of products and coproducts in the binding-friendly monoidal category ensures the existence of products and coproducts of modules, and the existence of an exponentiable unit ensures the existence of the derivative module $\Theta^{(n)}$ representing the binding of n-variables, by Theorem 5.6.

Definition 5.14. Given any non-empty binding signature $S = ((n_1, \ldots, n_{p_o}))_{o \in O}$ and a binding-friendly monoidal category C, we define the module signature S_C as

$$\coprod_{o \in O} \prod_{i=1}^{p_o} \Theta^{(n_i)}$$

Moreover, for any non-empty binding signature, this mapping from binding-friendly monoidal categories to module signatures extends to a 2-functor. By post-composing it with the 2-functor **Model**, we then get the category of models in any given binding-friendly monoidal category, 2-functorially.

Proposition 5.15. *Any non-empty binding signature S induces a 2-functor* Sem_S : **BindMonCat** $\longrightarrow$ **ModSig** *mapping a binding-friendly monoidal category C to the module signature S_C.*

Proof. See Appendix B. □

Theorem 5.16. *Any binding signature S induces a 2-functor* $\mathbf{BSigModel}_S$: **BindMonCat** $\longrightarrow$ Cat *mapping a binding-friendly monoidal category to its category of models, defined in the non empty case by the composition:*

$$\mathbf{BindMonCat} \xrightarrow{\mathrm{Sem}_S} \mathbf{ModSig} \xrightarrow{\mathbf{Model}} \mathrm{Cat}$$

Proof. 2-functoriality for the empty signature is trivial as its category of model is just the category of monoids. For the non-empty case, it follows by composition. □

6 Application: Relating Models in [Set, Set] and [𝔽, Set]

In her PhD dissertation [43, Chapter 4.11], Zsidó related initial semantics of binding signatures in [**Set**, **Set**] and in the monoidal category [𝔽, **Set**] introduced by Fiore et al. [16]. More specifically, she lifted the monoidal adjunction between [𝔽, **Set**] and [**Set**, **Set**] to monoids and uses it directly to build the initial model of [𝔽, **Set**] out of the one on [**Set**, **Set**], and vice versa.

We generalize this result, and provide a direct proof of it, using 2-functoriality. We lift the monoidal adjunction $[\mathbb{F}, \mathbf{Set}] \simeq [\mathbf{Set}, \mathbf{Set}]_f \leftrightarrows [\mathbf{Set}, \mathbf{Set}]$ to the 2-category **BindMonCat**, and hence obtain an adjunction between the respective categories of models by 2-functoriality (Proposition 3.11). We then recover Zsidó results as a corollary, since this adjunction is actually a coreflection, and therefore preserves initial objects. In summary, we construct the following functors:

$$\mathbf{BSigModel}_S([\mathbb{F}, \mathbf{Set}]) \simeq \mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]_f) \leftrightarrows \mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]).$$

6.1 The Coreflection Lifts to BindMonCat

Let us first review the adjunction between [𝔽, **Set**] and [**Set**, **Set**] in the 2-category of categories.

Proposition 6.1. *Let* $J : \mathbb{F} \longrightarrow \mathbf{Set}$ *be the canonical embedding. There is a coreflection* $[\mathbb{F}, \mathbf{Set}] \overset{-\circ J}{\leftrightarrows} [\mathbf{Set}, \mathbf{Set}]$ *which factorises as an equivalence of categories followed by a coreflective embedding*

$$[\mathbb{F}, \mathbf{Set}] \simeq [\mathbf{Set}, \mathbf{Set}]_f \leftrightarrows [\mathbf{Set}, \mathbf{Set}], \tag{2}$$

where the left adjoint to precomposition is the left Kan extension, and $[\mathbf{Set}, \mathbf{Set}]_f$ *denotes the full subcategory of* finitary endofunctors *of* **Set**, *that is, endofunctors that preserve* filtered colimits *[2, Definition 1.4].*

Proof. Since J is full and faithful, the left Kan extension Lan_J is also full and faithful by [40, Corollary 6.3.9 and Lemma 4.5.13]. Thus Lan_J factors as an equivalence with its image, which is $[\mathbf{Set}, \mathbf{Set}]_f$, and its coreflection. For instance, see [23, Proposition 4.2]. □

To lift the adjunction to **BindMonCat**, we must first show that the different categories involved are binding-friendly monoidal categories.

Proposition 6.2. *$[\mathbf{Set}, \mathbf{Set}]_f$ is a binding-friendly monoidal subcategory of $[\mathbf{Set}, \mathbf{Set}]$.*

Proof. Finitary endofunctors are closed under composition, colimits, finite products, and exponentials by Id (which is precomposition by $- + 1$). □

By the equivalence between $[\mathbb{F}, \mathbf{Set}]$ and $[\mathbf{Set}, \mathbf{Set}]_f$, we get a binding-friendly monoidal structure on $[\mathbb{F}, \mathbf{Set}]$, which is used by Fiore et al. [16] to define their category of models of a binding signature.

Corollary 6.3. *The category $[\mathbb{F}, \mathbf{Set}]$ is binding-friendly monoidal with $-^I$ given by precomposition by $- + 1$.*

Proof. To check what $-^I$ does, we consider $X : \mathbb{F} \to \mathbf{Set}$, apply the left Kan extension, precompose with $- + 1$, which is $-^I$ in $[\mathbf{Set}, \mathbf{Set}]_f$, and investigate the restriction along J: we get what is claimed. □

We are now ready to lift the coreflection to **BindMonCat**.

Proposition 6.4. *The factorisation $[\mathbb{F}, \mathbf{Set}] \simeq [\mathbf{Set}, \mathbf{Set}]_f \leftrightarrows [\mathbf{Set}, \mathbf{Set}]$ as an equivalence followed by a coreflection lifts in the 2-category **BindMonCat**.*

Proof. The equivalence in **BindMonCat** between $[\mathbb{F}, \mathbf{Set}]$ and $[\mathbf{Set}, \mathbf{Set}]_f$ comes from the fact that the binding-friendly monoidal structure on $[\mathbb{F}, \mathbf{Set}]$ is defined by transporting the one of $[\mathbf{Set}, \mathbf{Set}]_f$ along the equivalence.

Proposition 6.2 shows that the embedding $[\mathbf{Set}, \mathbf{Set}]_f \hookrightarrow [\mathbf{Set}, \mathbf{Set}]$ lifts in **BindMonCat**, so what remains to show is that the right adjoint $[\mathbf{Set}, \mathbf{Set}] \xrightarrow{-\circ J} [\mathbb{F}, \mathbf{Set}] \simeq [\mathbf{Set}, \mathbf{Set}]_f$ is binding-friendly. Preservations of finite products and coproducts follows from precomposition being continuous and cocontinuous; lax monoidality follows from being right adjoint to an oplax monoidal functor. Finally, exponentiation by the unit is preserved by the right adjoint. This is easy to check since in all the involved categories, this exponentiation is given by precomposition with $- + 1$. □

6.2 The Coreflection between the Categories of Models

Using 2-functoriality of $\mathbf{BSigModel}_S$ (Theorem 5.16), we can now directly lift the adjunction to the categories of models.

Theorem 6.5. *The factorisation of Eq. (2) lifts to the category of models, so that for any binding signature S, there is an equivalence and a coreflection as follows:*

$$\mathbf{BSigModel}_S([\mathbb{F}, \mathbf{Set}]) \simeq \mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]_f) \leftrightarrows \mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]). \tag{3}$$

Proof. As a 2-functor, $\mathbf{BSigModel}_S$ preserves equivalences and coreflections. □

We can now recover Zsidó's result as coreflections preserve initial objects.

Corollary 6.6. *All the back-and-forth functors in Eq. (3) preserve initial objects.*

Proof. Left adjoints and equivalences preserve colimits: this proves the case of all the above functors except $\mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]) \to \mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]_f)$. But the right adjoint R of a coreflection L also preserves the initial object because then the unit component $0 \to RL0$ is an isomorphism [40, Lemma 4.5.13]. □

7 Application: Relating Models in $[\mathbb{F}, \mathbf{Set}]$ and the De Bruijn Setting

In this section, we recover a new proof of the restricted equivalence between *De Bruijn models* [23] and models in $[\mathbb{F}, \mathbf{Set}]$. The two notions of models are exemplified in Listings 1 and 2 for the lambda calculus. The crucial difference is that a De Bruijn model does not provide explicit information about the support of terms, while in $[\mathbb{F}, \mathbf{Set}]$, terms come with their scope of available free variables.

Nonetheless, given any binding signature, De Bruijn models are equivalent to models in $[\mathbb{F}, \mathbf{Set}]$, provided that both are restricted to an appropriate notion of well-behavedness. Intuitively, these conditions amount, on the De Bruijn side, to restricting De Bruijn models to using only finitely many variables. On the finite-context side, we restrict to models that map the empty context to the set of closed terms (see Example 7.12 below for a counter-example).

Once these definitions are in place, we can prove the following result:

Theorem ([23, Theorem 4.25]). *Given any binding signature, the full subcategory of well-behaved models in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ is equivalent to the full subcategory of well-behaved models in $[\mathbb{F}, \mathbf{Set}]$.*

We state and prove this result formally below, in Theorem 7.15, using the 2-functorial theory we have developed.

7.1 The De Bruijn Setting

The framework of De Bruijn monads as developed by Hirschowitz et al. [23] relies on a *skew-monoidal* category, yet our 2-categorical framework involve monoidal categories only. To bridge the gap, we show in this section how their skew-monoidal category can be replaced by a binding-friendly monoidal category.

7.1.1 The Binding-Friendly Monoidal Category $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$. To model De Bruijn monads, we define the monoidal category $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ using left Kan extension, similarly to $[\mathbb{F}, \mathbf{Set}]$, but using instead the full subcategory $\mathbf{B}\mathbb{N}$ of $\mathbf{Set}$ that has $\mathbb{N}$ as its single object.

Proposition 7.1. *Let $\mathbf{B}\mathbb{N}$ be the full subcategory of sets with $\mathbb{N}$ as its single object. Denoting the canonical embedding $\mathbf{B}\mathbb{N} \to \mathbf{Set}$ by J, there is a coreflection $[\mathbf{B}\mathbb{N}, \mathbf{Set}] \overset{-\circ J}{\leftrightarrows} [\mathbf{Set}, \mathbf{Set}]$ which factorises as an equivalence of categories followed by a coreflective embedding*

$$[\mathbf{B}\mathbb{N}, \mathbf{Set}] \simeq [\mathbf{Set}, \mathbf{Set}]_{\omega^+, cc_0} \leftrightarrows [\mathbf{Set}, \mathbf{Set}], \tag{4}$$

where

- *the left adjoint to precomposition is the left Kan extension;*
- *$[\mathbf{Set}, \mathbf{Set}]_{\omega^+, cc_0}$ denotes the full subcategory of endofunctors on $\mathbf{Set}$ preserving the empty set as well as colimits of ω^+-*filtered colimits *[2, 1.21], for ω^+ the successor cardinal of ω.*

Proof. The proof is similar to that of Proposition 6.1. The essential image of the left Kan extension was shown to be $[\mathbf{Set}, \mathbf{Set}]_{\omega^+, cc_0}$ in the proof by Hirschowitz et al. [23, Lemma 4.29]. □

Proposition 7.2. *$[\mathbf{Set}, \mathbf{Set}]_{\omega^+, cc_0}$ is a binding-friendly monoidal subcategory of $[\mathbf{Set}, \mathbf{Set}]$.*

Proof. This subcategory is closed under composition, colimits, finite products, and exponentials by Id (which is precomposition by $- + 1$). □

By the equivalence between $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and $[\mathbf{Set}, \mathbf{Set}]_{\omega^+, cc_0}$, we get a binding-friendly monoidal structure on $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$, which we can use to define the category of models of a binding signature.

Corollary 7.3. *The category $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ is binding-friendly monoidal with $-^I$ given by precomposition with $suc \colon \mathbf{B}\mathbb{N} \to \mathbf{B}\mathbb{N}$, where $suc(f)$ is $f(- + 1)$ for any $f \colon \mathbb{N} \to \mathbb{N}$.*

Proof. To check what $-^I$ does, we consider $X\colon \mathbf{B}\mathbb{N} \to \mathbf{Set}$, apply the left Kan extension, precompose with $- + 1$, which is $-^I$ in $[\mathbf{Set}, \mathbf{Set}]_{\omega^+,cc_0}$, and investigate the restriction at $\mathbb{N}$: we get what is claimed. □

Similarly to Theorem 6.5 for $[\mathbb{F}, \mathbf{Set}]$, we then obtain, by 2-functoriality of $\mathbf{BSigModel}_S$, that the equivalence and the coreflection lift to models.

Proposition 7.4. *The factorisation* $[\mathbf{B}\mathbb{N}, \mathbf{Set}] \simeq [\mathbf{Set}, \mathbf{Set}]_{\omega^+,cc_0} \leftrightarrows [\mathbf{Set}, \mathbf{Set}]$ *as an equivalence followed by a coreflection lifts in the 2-category* $\mathbf{BindMonCat}$.

Proof. C.f. Proposition 6.4. □

Theorem 7.5. *For any binding signature S, there is an equivalence and a coreflection as follows:*

$$\mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]) \leftrightarrows \mathbf{BSigModel}_S([\mathbf{Set}, \mathbf{Set}]_{\omega^+,cc_0}) \simeq \mathbf{BSigModel}_S([[\mathbf{B}\mathbb{N}, \mathbf{Set}]])$$

7.1.2 De Bruijn Monads, and $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$. The De Bruijn Monads used by Hirschowitz et al. [23] to model Listing 1 are by definition relative monads for the inclusion $\mathbf{B}\mathbb{N} \hookrightarrow \mathbf{Set}$. As a consequence (see Proposition 7.7 below), they are equivalent to monoids in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$.

Definition 7.6. A De Bruijn monad consists of a set X (the image of $\mathbb{N}$) equipped with a *simultaneous substitution operation* $\_[\_] : X \times X^{\mathbb{N}} \to X$ and a *variable function* $\eta\colon \mathbb{N} \to X$. It must satisfy three equations: associativity $x[f][g] = x[f[g]]$, left unitality $x[\eta] = x$, and right unitality $\eta(n)[f][\eta] = x[f]$. A morphism between De Bruijn monads X and Y is a function $X \to Y$ compatible with variables and substitution.

De Bruijn monads can model syntax with variable binding in the de Bruijn style, where variables are mere natural numbers.

Proposition 7.7. *The category of De Bruijn monads is isomorphic to the category of monoids in* $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$.

Proof. By definition, a De Bruijn monad is a *relative monad* [6, Definition 2.1] on the embedding $\mathbf{B}\mathbb{N} \hookrightarrow \mathbf{Set}$. The desired isomorphism is constructed by Altenkirch et al. [6, Theorem 3.5]. □

A similar isomorphism holds for De Bruijn modules and modules over the corresponding monoid in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$, see Appendix C. This justifies that our categories of models induced by the binding-friendly monoidal structure coincide with them (up to isomorphism).

7.2 The Restricted Equivalence

By Propositions 6.4 and 7.4, we have the coreflection in $[\mathbb{F}, \mathbf{Set}] \simeq ([\mathbf{Set}, \mathbf{Set}]_f) \leftrightarrows [\mathbf{Set}, \mathbf{Set}]$ and $[\mathbf{B}\mathbb{N}, \mathbf{Set}] \simeq ([\mathbf{Set}, \mathbf{Set}]_{\omega^+,cc_0}) \leftrightarrows [\mathbf{B}\mathbb{N}, \mathbf{Set}]$ in $\mathbf{BindMonCat}$. Consequently, by 2-functoriality, the difference between using $[\mathbb{F}, \mathbf{Set}]$ and $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ to model syntax boils down to the difference between $[\mathbf{Set}, \mathbf{Set}]_f$ and $[\mathbf{Set}, \mathbf{Set}]_{\omega^+,cc_0}$ as binding-friendly monoidal categories. These categories are barely different, and therefore, models in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and $[\mathbb{F}, \mathbf{Set}]$ only differ in ill-behaved models, which we need to exclude to establish an equivalence.

7.2.1 Restricting $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ *and* $[\mathbb{F}, \mathbf{Set}]$. Since objects of $[\mathbb{F}, \mathbf{Set}]$ correspond to finitary endofunctors on sets, it is natural to consider a finitary restriction on objects $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$.

Definition 7.8. A functor $X\colon \mathbf{B}\mathbb{N} \to \mathbf{Set}$ is said *finitary* if every element $t \in X(\mathbb{N})$ has a *finite support*, that is, there exists $n \in \mathbb{N}$ such that for any $f\colon \mathbb{N} \to \mathbb{N}$, if $f(i) = i$ for all $i < n$, then $X(f)(t) = t$.

Example 7.9. Let us give examples of non-finitary functors. Consider the De Bruijn encoding of infinitary λ-terms, that is, we allow infinitely deep syntax trees. Let us denote the embedding of variables $\mathbb{N} \to X$ by var. This is not finitary since the term $app(var(1), app(var(2), \ldots))$ does not have a finite support.

If we consider a syntax with an infinitary operation $op\colon X^{\mathbb{N}} \to X$ and variables $var\colon \mathbb{N} \to X$, then the corresponding functor $\mathbf{B}\mathbb{N} \to \mathbf{Set}$ is not finitary, since $op(var(1), var(2), \ldots) \in X$ does not have a finite support.

To restrict $[\mathbb{F}, \mathbf{Set}]$, we need to exclude functors that are ill-behaved on the empty set.

Definition 7.10 ([23, Definition 4.15]). A functor $F : \mathbb{F} \to D$ or $F : \mathbf{Set} \to D$ is called *intersectional* if it preserves the following equaliser.

$$0 \longrightarrow 1 \underset{1}{\overset{0}{\rightrightarrows}} 2$$

Intersectional functors behave nicely with respect to evaluation at 0.

Proposition 7.11 ([23, Lemma 4.14]). *A functor $F\colon \mathbb{F} \to \mathbf{Set}$ is intersectional if it preserves empty intersections.*

Example 7.12. Let us give an example of a non-intersectional functor. Consider the functor $F\colon \mathbb{F} \to \mathbf{Set}$ mapping 0 to $\emptyset$ and n to the set of λ-terms taking free variables in $\{1, \ldots, n\}$. This induces a relative monad, but it is not a model of the binding signature of λ-calculus since it no longer has an abstraction operation $F(-+1) \to F -$ indeed, what would be its 0-component? However, it still has an application operation $F \times F \to F$ and thus is a model of the binding signature specifying a binary operation.

Remark 7.13. Any model in $[\mathbb{F}, \mathbf{Set}]$ is intersectional as soon as it has at least one closed term [23, Remark 4.19]. This is the case in particular if there is a binding operation (1), or a constant ().

7.2.2 Recovering The Equivalence The Between Categories of Models. Using 2-functoriality, the key resut to prove the equivalence between the (restricted) categories of models is that the restricted subcategories of $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and $[\mathbb{F}, \mathbf{Set}]$ are monoidally equivalent. Of course, this first requires to show that they are binding-friendly, in particular, that they are stable by monoidal product.

Proposition 7.14. *The full subcategory of non-empty finitary functors $\mathbf{B}\mathbb{N} \to \mathbf{Set}$ is equivalent in* **BindMonCat** *to the full subcategory of non-empty intersectional functors from $\mathbb{F} \to \mathbf{Set}$.*

Proof. C.f. Section 7.3. □

We then recover a simple proof of the main result:

Theorem 7.15 ([23, Theorem 4.25]). *Given any binding signature, the full subcategory of finitary models in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ is equivalent to the full subcategory of intersectional models in $[\mathbb{F}, \mathbf{Set}]$.*

Proof. By application of the 2-functor from Theorem 5.16 to the equivalence of Proposition 7.14, we get an equivalence between non-empty finitary models in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and non-empty intersectional models in $[\mathbb{F}, \mathbf{Set}]$. But since the initial functor is not a relative monad anyway, the restriction to non-empty models is actually unnecessary. □

This equivalence maps a De Bruijn model X to the functor $\mathbb{F} \to \mathbf{Set}$ mapping n to the subset of elements with support n; in the other direction it intuitively maps a functor $F\colon \mathbb{F} \to \mathbf{Set}$ to the union of all the sets $F(n)$ (see Lemma 7.16 below for a more formal statement).

7.3 Proof of Proposition 7.14

First, we have an adjunction between $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and $[\mathbb{F}, \mathbf{Set}]$ defined as follows, where $*$ denotes precomposition by inclusion; and Lan and Ran denote Kan extensions along inclusion.

$$[\mathbf{B}\mathbb{N}, \mathbf{Set}] \underset{*}{\overset{\mathrm{Ran}}{\rightleftarrows}} [\mathbf{Set}, \mathbf{Set}] \underset{\mathrm{Lan}}{\overset{*}{\rightleftarrows}} [\mathbb{F}, \mathbf{Set}] \tag{5}$$

As always, this adjunction restricts to an equivalence between the full subcategories of fixed points, i.e., of objects for which the (co)unit component is an isomorphism. It turns out that the fixed points of this adjunction are precisely the finitary functors in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and the intersectional functors in $[\mathbb{F}, \mathbf{Set}]$, so that we get an equivalence $[\mathbf{B}\mathbb{N}, \mathbf{Set}]_f \simeq [\mathbb{F}, \mathbf{Set}]_{int_0}$ between the two full subcategories. Moreover, the binding-friendly monoidal structures restrict to them, and it is easy to check that models correspond to finitary models in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and intersectional models in $[\mathbb{F}, \mathbf{Set}]$. Unfortunately, the above equivalence is not monoidal: $H\colon [\mathbb{F}, \mathbf{Set}] \to [\mathbf{B}\mathbb{N}, \mathbf{Set}]$ is not strong monoidal, although it is lax. In fact, H is almost strong monoidal: $H(I) \to I$ is an isomorphism, and $H(A \otimes B) \to H(A) \otimes H(B)$ is always an isomorphism, except for $B = 0$. This observation motivates the exclusion of the initial objects mentioned in the statement of Proposition 7.14. Doing so, we get binding-friendly monoidal subcategories $[\mathbf{B}\mathbb{N}, \mathbf{Set}]^*_f$ and $[\mathbb{F}, \mathbf{Set}]^*_{int_0}$, and the above equivalence restricts to a monoidal equivalence as desired.

The rest of this section is devoted to detailing the above arguments.

Lemma 7.16. *The left adjoint of Eq. (5) maps $F\colon \mathbb{F} \to \mathbf{Set}$ to the functor $\mathbf{B}\mathbb{N} \to \mathbf{Set}$ mapping $\mathbb{N}$ to the colimit of the following chain.*

$$F0 \to F1 \to F2 \to \dots$$

Lemma 7.17. *Consider the assignment $X\colon \mathbf{B}\mathbb{N} \to \mathbf{Set}$ to $X'\colon \mathbb{F} \to \mathbf{Set}$, defined as mapping n to the subset of elements t of $X(\mathbb{N})$ with support n, that is, such that for any $f\colon \mathbb{N} \to \mathbb{N}$, if $f(i) = i$ for all $i < n$, then $X(f)(t) = t$. This extends to a functor which is right adjoint to $[\mathbb{F}, \mathbf{Set}] \to [\mathbf{B}\mathbb{N}, \mathbf{Set}]$.*

The counit evaluated at $X\colon \mathbf{B}\mathbb{N} \to \mathbf{Set}$ is the inclusion of the elements of $X(\mathbb{N})$ with finite support. The unit evaluated at $F\colon \mathbb{F} \to \mathbf{Set}$ is a natural transformation which is the identity at each natural number n, except for $n = 0$ where it is the inclusion of $F0$ into the equaliser of $F1 \rightrightarrows F2$.

Corollary 7.18. *The fixed points of the adjunction between $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and $[\mathbb{F}, \mathbf{Set}]$ are the finitary functors in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ and the intersectional functors in $[\mathbb{F}, \mathbf{Set}]$, which are thus equivalent subcategories.*

Notation 7.19. Given a category C, we denote the full subcategory of non-initial objects of C by C^*, and by $[C, C]_f$ the full subcategory of finitary endofunctors on C.

Proposition 7.20. *$[\mathbf{B}\mathbb{N}, \mathbf{Set}]^*$ (resp. $[\mathbb{F}, \mathbf{Set}]^*$) is a binding-friendly monoidal subcategory of $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ (resp. $[\mathbb{F}, \mathbf{Set}]$).*

The following result completes the last step in the proof of Proposition 7.14.

Lemma 7.21. *The following lax monoidal functor is strong monoidal.*

$$[\mathbb{F}, \mathbf{Set}]^* \hookrightarrow [\mathbb{F}, \mathbf{Set}] \xrightarrow{\mathrm{Lan}} [\mathbf{Set}, \mathbf{Set}] \xrightarrow{*} [\mathbf{B}\mathbb{N}, \mathbf{Set}]$$

Proof. The composition of the first two functors factors as the following composition of strong monoidal functors, where $[\mathbf{Set}, \mathbf{Set}]^*_{\omega^+}$ denotes the full subcategory of non-empty endofunctors preserving ω^+-filtered colimits.

$$[\mathbb{F}, \mathbf{Set}]^* \simeq [\mathbf{Set}, \mathbf{Set}]^*_f \hookrightarrow [\mathbf{Set}, \mathbf{Set}]^*_{\omega^+} \hookrightarrow [\mathbf{Set}, \mathbf{Set}]$$

We are left with showing that $[\mathbf{Set},\mathbf{Set}]^*_{\omega^+} \hookrightarrow [\mathbf{Set},\mathbf{Set}] \to [\mathbf{B}\mathbb{N},\mathbf{Set}]$ is strong monoidal. Let us denote by R the right adjoint $[\mathbf{Set},\mathbf{Set}] \to [\mathbf{B}\mathbb{N},\mathbf{Set}]$, and L its left adjoint (the Kan extension). The lax monoidal structure on R is $RA \otimes RB = LRA \circ RB \xrightarrow{\varepsilon_A \circ B} A \circ RB$. We want to show that when A and B are in $[\mathbf{Set},\mathbf{Set}]^*_{\omega^+}$, this is an isomorphism. It is enough to show that $\varepsilon_{A,X} : LRA(X) \to A(X)$ is an isomorphism for any non empty set X, for any A in $[\mathbf{Set},\mathbf{Set}]_{\omega^+}$. Let us decompose the adjunction $L \dashv R$ as follows.

$$[\mathbf{B}\mathbb{N},\mathbf{Set}] \underset{R_1}{\overset{L_1}{\rightleftarrows}} [\mathbf{Set}^*,\mathbf{Set}] \underset{R_2}{\overset{L_2}{\rightleftarrows}} [\mathbf{Set},\mathbf{Set}] \qquad (L_1 \dashv R_1,\ L_2 \dashv R_2)$$

Therefore, the counit $LRA \to A$ decomposes as

$$L_2L_1R_1R_2A \xrightarrow{L_2\varepsilon_{1,A}R_2A} L_2R_2A \xrightarrow{\varepsilon_{2,A}} A, \tag{6}$$

and the desired property is that the image by R_2 of this morphism is an isomorphism. But the image of R_2 of the second morphism in Eq. (6) is an isomorphism since $L_2 \dashv R_2$ is a coreflection. Therefore, it is enough to show that $L_1R_1R_2A \xrightarrow{L_2\varepsilon_{1,A}R_2A} R_2A$ is an isomorphism.

We check that the essential image of the fully faithful functor $[\mathbf{Set}^*,\mathbf{Set}]_{\omega^+} \hookrightarrow [\mathbf{Set}^*,\mathbf{Set}] \xrightarrow{\mathrm{Lan}} [\mathbf{Set},\mathbf{Set}]$ is $[\mathbf{Set},\mathbf{Set}]_{\omega^+,cc_0}$ so that the following square is a weak pullback.

$$\begin{array}{ccc} [\mathbf{Set}^*,\mathbf{Set}]_{\omega^+} & \longrightarrow & [\mathbf{Set},\mathbf{Set}]_{\omega^+,cc_0} \\ \downarrow & & \downarrow \\ [\mathbf{Set}^*,\mathbf{Set}] & \xrightarrow[\mathrm{Lan}]{} & [\mathbf{Set},\mathbf{Set}] \end{array}$$

Therefore, we get a functor $[\mathbf{B}\mathbb{N},\mathbf{Set}] \to [\mathbf{Set}^*,\mathbf{Set}]_{\omega^+}$, and it is actually an equivalence because the top functor above is an equivalence, and $[\mathbf{B}\mathbb{N},\mathbf{Set}] \to [\mathbf{Set},\mathbf{Set}]_{\omega^+,cc_0}$ is an equivalence, which factors L_1, which means that for any X in $[\mathbf{Set}^*,\mathbf{Set}]_{\omega^+}$, the counit component $L_1R_1X \to X$ is an isomorphism. Now, if A is in $[\mathbf{Set},\mathbf{Set}]_{\omega^+}$, then R_2A is in $[\mathbf{Set}^*,\mathbf{Set}]_{\omega^+}$, and therefore $L_1R_1R_2A \to R_2A$ is an isomorphism as desired. □

8 Application: A Generalized Recursion Principle for Simply-Typed Languages

Initial semantics represent languages as initial models over a base monoidal category C, which models contexts. Consequently, the base category depends on the type system of the language considered. For instance, for a simply-typed languages with type system T, one possible base category is $[\mathbf{Set}^T,\mathbf{Set}^T]$ as in [43, Chapter 6], see [15] or [23, Section 7] for alternative choices. There is an adequate notion of simply-typed binding signature and the initial model can be understood as the syntax, in the spirit of initial semantics. However, the recursion principle provided by initiality is limited to models over the same base monoidal category, and in particular over the same type system. This means we cannot directly translate between languages with different type systems by initiality. In this section, we explain how to deal with this issue, exploiting the 1-functoriality of the categories of models.

8.1 Models from Simply-Typed Signatures, 1-Functorially

A simply-typed binding signature is a standard notion of signature for simply-typed languages featuring variable binding, e.g., see [15]. Given a type system T, binding arities represent constructors by specifying the output type τ and the types of the arguments t_i together with the types $\vec{u}_i$ of the variables bound in each of them. A language is then represented by a family of arities.

Definition 8.1. A *simply-typed binding arity* over a set T is an element of $((\vec{u_1}, t_1), \ldots, (\vec{u_n}, t_n), \tau) \in (T^* \times T)^* \times T$ which we write

$$t_1^{(\vec{u_1})} \times \ldots \times t_n^{(\vec{u_n})} \to \tau$$

Definition 8.2. A *simply-typed binding signature* over a set T is a family $(\alpha_i)_{i\in I}$ of of simply-typed binding arities α_i over T.

Example 8.3. Let T be a set with a binary operation $(\Rightarrow): T \times T \to T$ and an interpretation $g: B \to T$ of a set B of *base types*. The simply-typed lambda calculus on T can be specified by the simply-typed binding signature consisting of the following binding arities for each pair $(s, t) \in T^2$.

$$\mathsf{app}_{s,t} : (t \Rightarrow s) \times t \to s \qquad\qquad \mathsf{abs}_{s,t} : s^{(t)} \to (t \Rightarrow s)$$

This definition naturally assembles into a category of simply-typed signatures, by requiring morphisms to translate the type system and labels while respecting arities. An important operation on arities and signatures is *retyping*:

Definition 8.4. Given a simply-typed binding arity $\alpha = \left(t_1^{(\vec{u_1})} \times \ldots \times t_n^{(\vec{u_n})} \to \tau\right)$ over T and a function $g: T \to T'$, the retyping $g^*\alpha$ of α along g is defined by applying g to all the involved types:

$$g^*\alpha := \left(g(t_1)^{(g(\vec{u_1}))} \times \ldots \times g(t_n)^{(g(\vec{u_n}))} \to g(\tau)\right)$$

The retyping of a simply-typed signature is then the retyping of each arity, i.e. $g^*(\alpha_i)_{i\in I} = (g^*\alpha_i)_{i\in I}$.

Definition 8.5. We define the category STSig of simply-typed binding signatures as follows. Objects are pairs of a set T and a simply-typed binding signature S over T. A morphism between $(T, (\alpha_i)_{i\in I})$ and $(T', (\alpha'_{i'})_{i'\in I'})$ consists of a pair of functions $(T \xrightarrow{g} T', I \xrightarrow{h} I')$ translating the types and the labels in a way that is compatible with the arities, that is, such that $\alpha'_{h(i)} = g^*\alpha_i$ for all $i \in I$.

The crucial results for our purpose are the following.

Theorem 8.6. *There is a 1-functor* $\mathrm{STSig}^{op} \to \mathbf{ModSig}$, *where the target 2-category is considered as a 1-category, mapping a simply-typed binding signature* $(\alpha_i)_{i\in I}$ *over* T *to the following module signature on* $[\mathbf{Set}^T, \mathbf{Set}^T]$.

$$R \mapsto \coprod_{\alpha_i = \left(t_1^{(\vec{u_1})} \times \ldots \times t_n^{(\vec{u_n})} \to \tau\right)_{i\in I}} (R_{t_1}^{(\vec{u_1})} \times \ldots \times R_{t_n}^{(\vec{u_n})}) \cdot \mathbf{y}_\tau$$

where

- $\mathbf{y}_\tau : T \to \mathbf{Set}$ *maps* t *to* 1 *if* $t = \tau$, *and to* $\emptyset$ *otherwise.*
- $R^{(u_1,\ldots,u_n)} : \mathbf{Set}^T \to \mathbf{Set}^T$ *maps* $X : T \to \mathbf{Set}$ *to* $R(X + \mathbf{y}_{u_1} + \ldots + \mathbf{y}_{u_n}) : T \to \mathbf{Set}$.

Theorem 8.7. *We define the 1-functor* **STSigModel** *as the composition*

$$\mathrm{STSig}^{op} \to \mathbf{ModSig} \xrightarrow{\mathbf{Model}} \mathrm{Cat}.$$

Given a simply-typed signature, it computes its category of models, which always has an initial object; indeed, Zsidó [43, Chapter 6] constructs an initial model for any simply-typed binding signature.

As a consequence, any morphism between simply-typed binding signatures $(g, h): (T, S) \to (T', S')$ induces a functor $\mathbf{STSigModel}(g, h): \mathbf{STSigModel}(T', S') \to \mathbf{STSigModel}(T, S)$ between their categories of models. We get a unique morphism from the initial model of $\mathbf{STSigModel}(T, S)$ to $\mathbf{STSigModel}(g, h)(M)$ for any model M of (T', S'): this enables us to translate a syntax into another one with a different type system.

Proposition 8.8. *Given a morphism $(g,h)\colon (T,S) \to (T',S')$ between simply-typed binding signatures, the functor* **STSigModel**(g,h) *maps a model M of (T',S') to the model $(g,h)^*M$ of (T,S) whose underlying monad maps $X\colon T \to \mathbf{Set}$ to*

$$\tau \mapsto M\left(\coprod_t X_t \cdot \mathbf{y}_{g(t)}\right)_{g(\tau)}$$

Proof. The functor STSig $\to$ **ModSig** maps (g,h) to a morphism of module signatures whose underlying lax monoidal functor is

$$[\mathbf{Set}^T, \mathbf{Set}^T] \xrightarrow{[\mathrm{Lan}_g, \mathbf{Set}^g]} [\mathbf{Set}^{T'}, \mathbf{Set}^{T'}].$$

Unfolding Lan_g yields the claimed result. □

Example 8.9. Consider the simply-typed binding signature S of simply-typed lambda-calculus for the set T generated by a set of base types B and a binary operation $f\colon T \times T \to T$. Now, consider T' which is the disjoint union of T with a singleton set $\{\tau\}$. The initial model of (T',S) is like the simply-typed lambda calculus, but extended with a new type τ which cannot occur in an arrow type. We can obviously still embed the lambda calculus into this extended language: this is actually given by the initial morphism from the initial model of (T,S) to $(g,h)^*M'$, where M' is the initial model of (T',S), and $g\colon T \to T'$ is the inclusion and h is the identity function. Note that in this particular case, $(g,h)^*M'$ is actually isomorphic to M.

8.2 Generalized Recursion Principles

8.2.1 Extended Models of Simply-Typed Binding Signatures. In the situation described above, we exploit a morphism $(g,h)\colon (T,S) \to (T',S')$ to turn a model M of (T',S') into a model $(g,h)^*M$ of (T,S), and then get an initial morphism from the initial model of (T,S) to this model. In short, (g,h) allows us to see models of (T',S') as models (in an extended sense) of (T,S). This suggests the following definition.

Definition 8.10. We define the category ExtModels(T,S) of *extended models* of (T,S) as follows. An object $((T,S) \xrightarrow{g,h} (T',S'), M)$ is a model M of a simply-typed binding signature S' over a set T' together with a morphism (g,h) between (T,S) and (T',S'). A morphism between extended models $((T,S) \xrightarrow{g_1,h_1} (T_1,S_1), M_1)$ and $((T,S) \xrightarrow{g_2,h_2} (T_2,S_2), M_2)$ consists of

- a morphism $(T_1,S_1) \xrightarrow{(g,h)} (T_2,S_2)$ such that $(g,h) \circ (g_1,h_1) = (g_2,h_2)$;
- a morphism $M_1 \to (g,h)^*(M_2)$ in **STSigModel**(T_1,S_1).

Remark 8.11. This is precisely the total category corresponding to the functor $((T,S)/\mathrm{STSig})^{\mathrm{op}} \to \mathrm{STSig}^{\mathrm{op}} \to \mathrm{Cat}$ by the *Grothendieck construction* [27, Definition 1.10.1], where $(T,S)/\mathrm{STSig}$ denotes the coslice category under (T,S): objects are pairs (T',S') with a morphism $(g,h)\colon (T,S) \to (T',S')$, and morphisms are commutative triangles, and the projection functor $(T,S)/\mathrm{STSig}$ maps an object as above to (T',S').

Note that the Grothendieck construction of a functor $F\colon C^{\mathrm{op}} \to \mathrm{Cat}$ yields a category whose objects are pairs of an object c of C and an object x of $F(c)$; a morphism between (c,x) and (c',x') consists of a morphism $c \xrightarrow{f} c'$ and a morphism $x \to F(f)(x')$.

It is then possible to prove that (T,S) is initial in this extended category of models, in which translations between languages over different type systems is possible. In summary, this provides simply-typed syntax with a generalized recursion principle, which we illustrate in Section 8.3.

Theorem 8.12. *Let M be the initial model M of (T, S). Then $((T, S) \xrightarrow{\text{id}} (T, S), M)$ is initial in* ExtModels(T, S).

This follows from the following general lemma, exploiting the fact that ExtModels(T, S) is a Grothendieck construction (Remark 8.11).

Lemma 8.13 ([5, Lemma 6.11]). *Let $F: S^{op} \to \mathsf{Cat}$ be a functor from a category with an initial object 0, such that $F0$ has an initial object $0'$. Then, the pair $(0, 0')$ is initial in the total category of the fibration on S generated by F by the Grothendieck construction.*

8.2.2 Recovering Ahrens's Framework as a Grothendieck Construction. We end this section by revisiting Ahrens' extended models with its initiality property in light of Theorem 8.12. Ahrens [4] defines a notion of *typed signature* with an associated category of models [4, Definition 3.30] that assembles monads on $\mathbf{Set}^T$ for different T. A typed signature [4, Definition 3.25] consists of a *signature for types* Σ and a *term-signature* S over Σ (see [4, 3.1] and [4, Definition 3.24]). First, any signature for types Σ induces a category of *algebras* $\mathbf{Set}_\Sigma$: they are sets equipped with an interpretation of the type system and one can show that there is an initial object (it is the category of algebras of a finitary endofunctor on sets). Then, a *term-signature* S over Σ induces a functor $\mathbf{Set}_\Sigma \to$ STSig mapping T to a binding signature S_T over T

Example 8.14. Given a set of *base types* B, the category of algebras of the type signature Σ of the simply-typed lambda calculus are sets T equipped with a binary operation $(\Rightarrow): T \times T \to T$ and a function $g: B \to T$.

The term-signature for the simply-typed lambda calculus generates the functor $\mathbf{Set}_\Sigma \to$ STSig that maps an algebra $(T, \Rightarrow, g)$ to the simply-typed binding signature given in Example 8.3.

Ahrens' category of models of a typed signature is recovered by computing the Grothendieck construction as for ExtModels(T, S): a model consists of a Σ-algebra T and a model M of the binding signature S_T. The existence of the initial model is again guaranteed by Lemma 8.13.

Proposition 8.15. *Any model (T, M) in the sense of Ahrens induces an* extended model $((T_0, S_{T_0}) \xrightarrow{i} (T, S), M)$*, where i is the image of the initial morphism $T_0 \to T$ by the functor* $\mathbf{Set}_\Sigma \to$ STSig*. Moreover, this assignement is functorial.*

Proof. The functor $\mathbf{Set}_\Sigma \to$ STSig factors as $\mathbf{Set}_\Sigma \to (T_0, S_{T_0})/\text{STSig} \to$ STSig, where T_0 is the initial object of $\mathbf{Set}_\Sigma$, and the first functor maps T to the image of the initial morphism $T_0 \to T$.

This induces a functor between the total categories built using the Grothendieck construction by [27, Theorem 1.10.7]. □

However, the converse does not hold: our category of extended models is *larger* than Ahrens' category of models, as shown by the following example.

Example 8.16. Example 8.9 is an extended model that is not a model of the simply-typed lambda calculus in the sense of Ahrens'. Indeed, T' is not a model of Σ, because the arrow construction on types is partial: it is not defined when one of the arguments is τ. Even if we artificially extend the domain of the arrow construction on types, the initial model of (T', S) lacks application and abstraction for τ.

8.3 Application: Translating PCF to the Untyped Lambda Calculus

Ahrens illustrated his initiality result with the translations from classical logic to intuitionistic logic [4, Section 4] and from PCF to the untyped lambda calculus [4, Section 5]. Since our category of extended models is larger, these examples are still valid. In this section, we detail the translation of PCF [37], following Ahrens' presentation of the calculus.

Definition 8.17 (Binding signature of PCF). The set T_{PCF} of simple types is generated by two base types, $\mathbb{B}$ and $\mathbb{N}$, and a binary operation $- \Rightarrow -\colon T_{PCF} \times T_{PCF} \to T_{PCF}$.

The simply-typed binding signature S_{PCF} of PCF consists of the family of binding arities for application and abstraction as in Example 8.3, as well of the following binding arities (where we additionally label them for clarity).

$$(\mathrm{true}\colon () \to \mathbb{B}) \qquad (\mathrm{false}\colon () \to \mathbb{B})$$
$$(0\colon () \to \mathbb{N}) \qquad (\mathrm{succ}\colon () \to \mathbb{N} \Rightarrow \mathbb{N}) \qquad (\mathrm{pred}\colon () \to \mathbb{N} \Rightarrow \mathbb{N})$$
$$(\mathrm{Fix}_\tau\colon (\tau \Rightarrow \tau) \to \tau)_{\tau \in T_{PCF}} \qquad (\mathrm{if}_\iota\colon () \to \mathbb{B} \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota)_{\iota \in \{\mathbb{B},\mathbb{N}\}}$$

Let M_{PCF} denotes the initial model.

Definition 8.18 (Binding signature of untyped lambda calculus). The set T_{ULC} consists of a single type $\star$. The simply-typed binding signature S_{ULC} consists of the following binding arities.

$$\mathsf{abs}\colon \star^{(\star)} \to \star \qquad \mathsf{app}\colon \star \times \star \to \star$$

Let M_{ULC} denotes the initial model. Exploiting the isomorphism $\mathbf{Set}^{\{\star\}} \cong \mathbf{Set}$, we consider M_{ULC} as a monad on $\mathbf{Set}$.

To translate PCF to the untyped lambda calculus, we need to give M_{ULC} a structure of extended model for (T_{PCF}, S_{PCF}). That is, we need to find a simply-typed binding signature S over a set T, with a morphism $(g, h)\colon (T_{PCF}, S_{PCF}) \to (S, T)$, and a (S, T)-model structure on M_{ULC}. An obvious choice is to take $T = T_{ULC}$ and $S = S_{ULC}$, since M_{ULC} is automatically a model of (T, S). We have indeed a function $g\colon T_{PCF} \to T$ mapping any type to $*$, but unfortunately, there is no suitable function h, because for example S_{ULC} does not have any binding arity such as $0\colon () \to g(\mathbb{N}) = \star$.

Instead, we consider the binding signature $g^* S_{ULC}$ over T_{PCF}. This means that we need to provide operations $1 \to M_{ULC}$ for each constant of S_{ULC}, a fix operation $M_{ULC} \to M_{ULC}$. Note that any closed term $t \in M_{ULC}(\emptyset)$ induces an operation $1 \to M_{ULC}$ whose X-component maps the unique element of 1 to $M_{ULC}(i)(t)$, where $i\colon \emptyset \to X$ is the initial function. Exploiting the Church encoding of booleans, we pick the closed term $\lambda t.\lambda f.t$ for true, and $\lambda t.\lambda f.f$ for false, so that we can define the if constant as $\lambda b.\lambda t.\lambda f.b\ t\ f$. Natural numbers can be similarly encoded. For the fixpoint operation, we can define the operation $M_{ULC} \to M_{ULC}$ as mapping t to $Y\ t$, where Y is any fixpoint combinator, e.g., Curry's combinator $\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$.

Let us make explicit what the initial morphism gives us. We just showed that M_{ULC} induces a model M'_{ULC} of $g^* S_{ULC}$ with the same underlying monad. Initiality provides us with a morphism $M_{PCF} \to (g, h)^* M'_{ULC}$. Unfolding the codomain using Proposition 8.8, we see that $(g, h)^* M'_{ULC}$ maps $\Gamma\colon T_{PCF} \to \mathbf{Set}$ and $\tau \in T_{PCF}$ to the set $M(\coprod_{t \in T_{ULC}} \Gamma_t)$. That is $(g, h)^* M'_{ULC}$ just forgets the types of the variables in the input context Γ. As a model morphism, the initial morphism translates all the PCF operations recursively using the operations of M'_{ULC} that we sketched above.

9 Related Work

Different frameworks have been considered over time for initial semantics [16, 25, 36]. Lamiaux and Ahrens [31] recently presented a unified account of initial semantics, parametrized by a choice of a base monoidal category on which we base this work. We review briefly the relevant part of their work in Section 2. They have written an extensive overview of the different traditions [31, Section 6], and we refer to it for discussion on the links with others approaches.

Relating the framework of Fiore et al. [16] using $[\mathbb{F}, \mathbf{Set}]$ and that of Hirschowitz and Maggesi [24] using $[\mathbf{Set}, \mathbf{Set}]$ was first studied by Zsidó [43] in her PhD dissertation. In that work [43, Chapter 4], Zsidó lifts the monoidal adjunction between $[\mathbb{F}, \mathbf{Set}]$ and $[\mathbf{Set}, \mathbf{Set}]$ to the categories of monoids, then uses it to directly build the initial model of $[\mathbb{F}, \mathbf{Set}]$ out of the one on $[\mathbf{Set}, \mathbf{Set}]$, and vice versa. We have generalized this result in Section 6, by using 2-functoriality to lift the monoidal adjunction directly to models (Theorem 6.5), recovering Zsidó's result as a particular case using that the adjunction is a coreflection (Corollary 6.6). We have further refined it by factoring it through the equivalence $[\mathbb{F}, \mathbf{Set}] \simeq [\mathbf{Set}, \mathbf{Set}]_f$.

In her PhD, Zsidó also studied the links in the simply-typed case [43, Chapter 7]. She lifted the adjunction of the respective monoidal categories by hand to models — which technically also gives her an adjunction in the untyped case, though it is not mentioned — but failed to conclude that initial objects were preserved.[2] We believe that this preservation could be proved following an argument similar to that of the proof of Corollary 6.6 – that is, given a coreflection, both adjoint functors preserve the initial object.

Hirschowitz et al. [23] introduced De Bruijn monads and modules over them and defined models of binding signatures. We showed how they fit into our framework as monoids and modules in a binding-friendly monoidal category. Moreover, we deduce their equivalence between finitary De Bruijn models and intersectional models in $[\mathbb{F}, \mathbf{Set}]$ from our 2-categorical machinery.

Generalizing the recursion principle of typed languages has only been studied for simply-typed languages by Ahrens [4]. To do so, Ahrens designed a specific notion of signatures and associated models to integrate the simply-typed systems directly into it, fixing the shape of the base categories to be of the form $[\mathbf{Set}^T, \mathbf{Set}^T]$ with T an appropriate algebra for the type signature. As discussed by Lamiaux and Ahrens [31, Section 6.3.5], the links with the usual monoidal setting was unclear. In Section 8, we have used instead the 2-functoriality of models to provide a generalized recursion principle based on the standard notion of simply-typed binding signature, without the need of introducing new signatures. We showed that like ours, Ahrens' category of models can be recovered as a total category by the Grothendieck construction; however our recursion principle is more general.

Power and Tanaka [39, 42] provide a general recipe to construct monoidal categories such as $[\mathbb{F}, \mathbf{Set}]$ and some linear, simply-typed variants. Their recipe also yields a notion of (generalised) binding signature for each such monoidal category.

In the context of initial semantics, skew-monoidal categories have been used in particular by Hirschowitz et al. [23], Fiore and Szamozvancev [13], and Borthelle et al. [11]. As Proposition 7.7 suggests, Hirschowitz et al. [23] could have focused on the monoidal category $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$ instead, at the cost of working with a more complicated monoidal product than theirs, which does not involve quotients. Similarly, Fiore and Szamozvancev [13] took advantage of their skew monoidal product to formalise their framework in a quotient-free type theory. Finally, the work by Borthelle et al. [11] has since been superseded by that of Hirschowitz and Lafont [26] which works with truly monoidal categories.

10 Conclusion

In this work, we have shown that module signatures, when parametrized by a monoidal category, have a 2-categorical structure, and that in this case the associated category of models can be computed by a 2-functor. We have then leveraged this 2-functoriality to propagate adjunctions and equivalences on the base monoidal categories, corresponding to the different implementations of abstract syntax, to the associated categories of models. This enabled us to recover the results presented by Zsidó [43] and Hirschowitz et al. [22] using a generic proof method, whereas previously, these results only had instance-specific proofs. Using 1-functoriality of the 2-functoriality of models, we have also designed a generalized recursion principle for simply-typed languages with variables binding. This enables us to recover and understand the framework developed by Ahrens [4] as a

[2] See the end of Section 1.2 of Zsidó's dissertation [43].

total category, and to generalize it. In both cases, we hope that the solutions we have designed using 2-functoriality will scale to more complex languages with variable binding, such as polymorphic languages like System F.

10.1 Open Problems

With our work, we open doors to comparing different implementations and proving more general recursion principles. While we have addressed some instances, numerous approaches remain to be compared or generalized.

(1) In the untyped case, we have compared unscoped syntax with well-scoped syntax. It remains to understand how the nominal approach [17] fits into the picture, starting from Power's monoidal structure on the category of nominal sets [38, Theorem 4.6].
(2) The 2-functor of Theorem 5.16 only deals with untyped syntax. It would be interesting to generalize **BindMonCat** to simply-typed languages so as to generalize our category of extended models to other categories than $[\mathbf{Set}^T, \mathbf{Set}^T]$ which were considered by Fiore and Hur [15] and Hirschowitz et al. [23].
(3) Several accounts of initial semantics for equations or reductions between terms have been given, using different notions of context [3, 5, 12]. It would be interesting to extend our 2-functor to incorporate equations or reductions, in order to compare these approaches.
(4) Initial semantics results for polymorphic type systems like system F or F_ω exist [14, 21], but are currently underdeveloped. One of the challenges is that the binding of type variables makes it necessary, formally, to consider changing the base category modelling contexts. We hope that our techniques will make it easier to construct useful initial semantics for polymorphic type systems, and scale to it.
(5) Some recent accounts of initial semantics have been working on skew monoidal categories [11, 13, 23]. As discussed in Section 9, skew-monoidal categories can sometimes be avoided, and our framework directly applied. Nonetheless, we are not sure to which extent our constructions readily extend to this more general setting.

Acknowledgments

We thank Tom Hirschowitz and Thea Li for helpful discussions, and the anonymous referees for their detailed feedback. This work received government funding managed by the French National Research Agency under the France 2030 program, reference "ANR-22-EXES-0013".

A Proof of Theorem 5.6

Lemma A.1. *Let C be a category with finite products and T be a monad on C preserving those products. Let c be an exponentiable object of C, that is, such that given an algebra $TX \to X$, the algebra structure on X^c is given by the transpose of the following morphism makes it the exponential X^{Tc} in the category of T-algebras.*

$$T(X^c) \times c \to T(X^c) \times Tc \cong T(X^c \times c) \to T(X) \to X$$

Proof. We exploit the following characterisation of the category of T-algebras, as the following pullback [41, Theorem 14].

$$\begin{array}{ccc} Alg(T) & \longrightarrow & [\mathrm{Kl}(T)^{\mathrm{op}}, \mathbf{Set}] \\ \downarrow & \lrcorner & \downarrow{\scriptstyle [L^{\mathrm{op}}, \mathbf{Set}]} \\ C & \xrightarrow[\mathrm{y}]{} & [C^{\mathrm{op}}, \mathbf{Set}] \end{array}$$

where $\mathbf{y}$ denotes the Yoneda embedding, and $L\colon C \to \mathrm{Kl}(T)$ denotes the canonical left adjoint into the Kleisli category of T.

Let us show that $\mathbf{y}_{Lc}$ is exponentiable in $[\mathrm{Kl}(T)^{\mathrm{op}}, \mathbf{Set}]$. Note that since $\mathrm{Kl}T$ is not small in general, $[\mathrm{Kl}T^{\mathrm{op}}, \mathbf{Set}]$ is not necessarily cartesian closed. However, it fully faithfully embeds into the category $[\mathrm{Kl}(T)^{\mathrm{op}}, \mathbf{SET}]$ where $\mathbf{SET}$ is the category of large sets[3], which is cartesian closed by replaying the proof of [29, Proposition A.1.5.5] for the larger universe $\mathbf{SET}$.

Let A be a functor from $\mathrm{Kl}(T)^{\mathrm{op}}$ to $\mathbf{Set}$. Let us check that $A^{\mathbf{y}_{Lc}}(X)$ is small, so that the exponential $A^{\mathbf{y}_{Lc}}$ in $[\mathrm{Kl}T^{\mathrm{op}}, \mathbf{SET}]$ actually lives in $[\mathrm{Kl}T^{\mathrm{op}}, \mathbf{Set}]$. By the Yoneda Lemma, $A^{\mathbf{y}_{Lc}}$ maps X to

$$\begin{aligned} \hom_{[\mathrm{Kl}T^{\mathrm{op}},\mathbf{SET}]}(\mathbf{y}_X, A^{\mathbf{y}_{Lc}}) &\cong \hom_{[\mathrm{Kl}T^{\mathrm{op}},\mathbf{SET}]}(\mathbf{y}_X \times \mathbf{y}_{Lc}, A) \\ &\cong \hom_{[\mathrm{Kl}T^{\mathrm{op}},\mathbf{SET}]}(\mathbf{y}_{X\times Lc}, A) && \text{By continuity of the Yoneda embedding} \\ &\cong A(X \times Lc) && \text{By the Yoneda Lemma} \end{aligned} \tag{7}$$

which is indeed small.

Now, let us assume that $A \circ L^{\mathrm{op}} \cong \mathbf{y}_d$ for some object d of C. Let us show that $A^{\mathbf{y}_{Lc}} \circ L^{\mathrm{op}}$ is representable.

$$\begin{aligned} A^{\mathbf{y}_{Lc}} LX &\cong A(LX \times Lc) && \text{By (7)} \\ &\cong A(L(X \times c)) && \text{By preservation of products} \\ &\cong \hom(X \times c, d) && \text{By representability of } A \\ &\cong \hom(X, d^c) && \text{By exponentiability of } c \\ &= \mathbf{y}_{d^c}(X) \end{aligned}$$

Let us explain why L preserves products, Kleisli morphisms $A \to T(X \times Y)$ are in bijection with morphisms $A \to TX \times TY$, and the latter are in bijection with pairs of Kleisli morphisms $A \to TX$ and $A \to TY$.

What we have shown is that $A^{\mathbf{y}_{Lc}}$ is in the pullback characterising the category of T-algebras, the underyling object of C being d^c, if d is the underlying object of A. Moreover, it is the really the exponential of A by Lc in $[\mathrm{Kl}(T)^{\mathrm{op}}, \mathbf{SET}]$ in which T-algebras fully faithfully embeds. It follows that it is also the exponential in T-algebras.

□

Lemma A.2. *Let D be a category with finite products, d an exponentiable object of D. Then given any category C, the constant functor mapping any object of c to d is also exponentiable, with F^d defined as $F(-)^d$.*

We are now ready to prove Theorem 5.6.

Proof of Theorem 5.6. By Theorem 5.3, the category of module signatures is monadic over $[\mathsf{Mon}(C), C]$, and the monad T maps Σ to $T(\Sigma) := \Sigma(-) \otimes -$. Because $- \otimes -$ preserves binary products on the left, this monad preserves binary products.

By Lemma A.2, the endofunctor $- \times I$ on $\mathsf{Mon}(C) \to C$ has a right adjoint R mapping Σ to $\Sigma(-)^I$.

By Lemma A.1, the exponential functor R lifts to the category of module signatures as claimed.

It is the exponential Σ^Θ in the category of module signatures since the monad of Theorem 5.3 maps I to Θ (up to isomorphism). □

[3]Assuming a set universe is not strictly necessary but makes the proof simpler.

B Proof of Proposition 5.15

Definition B.1. Let $\mathbf{ModSig}_{\mathrm{ps}}$ be the sub-2-category of $\mathbf{ModSig}$ with 1-cells $(C, \Sigma) \xrightarrow{(F,\alpha)} (D, \Sigma')$ such that α is an isomorphism.

Definition B.2. A *binding-friendly signature* is a 2-functor from $\mathbf{BindMonCat}$ to $\mathbf{ModSig}_{\mathrm{ps}}$ commuting with the projection to the 2-category of monoidal categories, lax monoidal functors and monoidal transformations.

Example B.3. The tautological binding-friendly signature maps a binding-friendly monoidal category C to the module signature $\Theta\colon \mathsf{Mon}(C) \to \mathsf{Mod}(C)$.

Proposition 5.15 follows from the fact that binding-friendly signatures are stable under finite products, non-empty coproducts, and exponential by the unit, as stated in the following lemma. It is crucial there that we restrict the definition of binding-friendly signature to the sub-2-category $\mathbf{ModSig}_{\mathrm{ps}}$ of $\mathbf{ModSig}$, otherwise those stability properties do not hold.

Lemma B.4. *(1) Let $(\Sigma_i)_{i\in I}$ be a finite family of binding-friendly signatures. Then their pointwise product $\Pi_i\Sigma_i$ defined as $C \mapsto \prod_i \Sigma_i C$ is a binding-friendly signature.*
(2) Let $(\Sigma_i)_{i\in I}$ be a non-empty family of binding-friendly signatures. Then their pointwise coproduct $\coprod_i \Sigma_i$ defined as $C \mapsto \coprod_i \Sigma_i C$ is a binding-friendly signature.
(3) Let Σ be a binding-friendly signature. Then the pointwise exponential by unit Σ^I defined as $C \mapsto (\Sigma C)^I$ is a binding-friendly signature.

C Correspondence Between De Bruijn Modules and Modules over Monoids

Proposition 7.7 states that the category of De Bruijn monads is equivalent to the category of monoids in the monoidal category $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$. In this section, we show that the category of De Bruijn modules [23, Definition 6.8] is equivalent to the category of modules over monoids in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$.

Definition C.1. The category of De Bruijn modules over a De Bruijn monad X is the category of modules relative to X (see [3, Definition 14]).

Proposition C.2. *The category of modules relative to a monad X relative to some $J\colon \mathbb{C} \to \mathbb{D}$ is isomorphic to the category from the Kleisli category [6, 2.3] of R to $\mathbb{D}$.*

Proof. Immediate, by unfolding the definitions. □

The main theorem is the following.

Theorem C.3. *Let X be a relative monad along some functor $J\colon \mathbb{C} \to \mathbb{D}$ such that the pointwise left Kan extension $\mathrm{Lan}_J\colon [\mathbb{C}, \mathbb{D}] \to [\mathbb{D}, \mathbb{D}]$ exists. Then, the category of relative modules over X, is isomorphic to the category of modules over X, i.e., algebras for the monad $- \otimes X$, for the skew monoidal structure on $[\mathbb{C}, \mathbb{D}]$ induced by [6, Theorem 3.1].*

Theorem C.4. *The previous isomorphisms of categories for each relative monad X gather into an isomorphism of categories between the total category of relative modules and the total category of modules over monoids in $[\mathbb{C}, \mathbb{D}]$.*

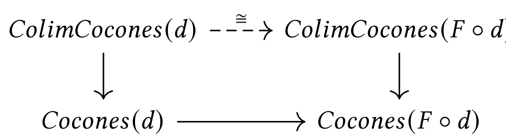
Corollary C.5. *The category of De Bruijn modules is isomorphic to the category of modules over monoids in $[\mathbf{B}\mathbb{N}, \mathbf{Set}]$.*

In the rest of this section we focus on the proof of Theorem C.3. This follows from the following general lemmas.

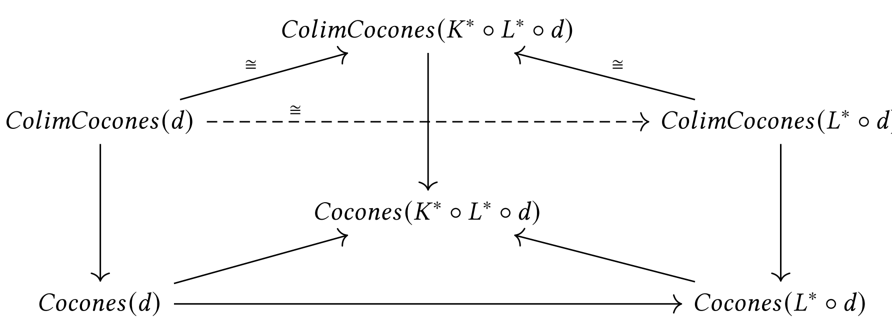
Lemma C.6. *Let* $L\colon \mathcal{A} \to \mathcal{B}$ *be a bijective-on-objects functor and* C *be category such that the left Kan extension* $\mathrm{Lan}_L\colon [\mathcal{A}, C] \to [\mathcal{B}, C]$ *exists. Then, the precomposition functor* $- \circ L\colon [\mathcal{B}, C] \to [\mathcal{A}, C]$ *is strictly monadic.*

Proof. We apply Beck monadicity theorem [34, Theorem VI.7.1]. It is enough to show that the precomposition functor $L^* = - \circ L$ creates pointwise colimits, since absolute coequalisers are pointwise (they are preserved by evaluation functors).

Note that given a functor $F\colon \mathcal{A} \to \mathcal{B}$ and a diagram $d\colon J \to \mathcal{A}$, if there exists an isomorphism $\mathit{ColimCocones}(d) \cong \mathit{ColimCocones}(F \circ d)$ such that the following diagram commutes and is a pullback, where $\mathit{Cocones}(G)$ is the category of cocones over G and $\mathit{ColimCocones}(G)$ is the full subcategory of initial objects therein, then the functor F creates the d-colimits, and this is a necessary condition in case the colimit $F \circ d$ exists.



Assume a diagram $d\colon \mathcal{D} \to [\mathcal{B}, C]$ such that the colimit of $L^* \circ d$ is computed pointwise. Denoting $K\colon |\mathcal{A}| \to \mathcal{A}$ the discrete category inclusion, this means that the colimit $K^* \circ L^* \circ d$ exist, and thus, by [34, Theorem V.3.2], the plain squares in the below diagram are pullbacks with noted isomorphisms



Therefore, we get the dashed arrow, which is an isomorphism by the cancellation property of isomorphisms. Moreover, by the pasting law for pullbacks [34, Exercise III.4.8], the front square is a pullback, as desired. □

Lemma C.7. *Let* $\mathbb{C}(LA, B) \cong \mathbb{D}(JA, UB)$ *be a relative adjunction. If the pointwise left Kan extension* Lan_J *exists, then the (pointwise) left Kan extension* $\mathrm{Lan}_L(-)$ *can be computed as* $\mathrm{Lan}_J(-) \circ U$.

Proof.

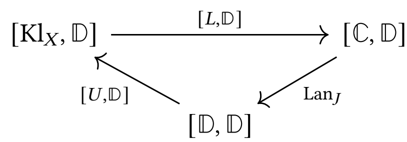
$$
\begin{aligned}
&\mathrm{Lan}_J\, F \circ U(x) \\
&\qquad \cong \int^{c \in \mathbb{C}} \mathbb{D}(Jc, Ux) \times Fc \quad &&\text{(By definition of pointwise left Kan extension [6, §3.1])} \\
&\qquad \cong \int^{c \in \mathbb{C}} \mathbb{C}(Lc, x) \times Fc &&\text{(By the relative adjunction)} \\
&\qquad \cong \mathrm{Lan}_L(F)(x) &&\text{(By definition of pointwise left Kan extension)}
\end{aligned}
$$

□

Proof of Theorem C.3. Let L and U denote the free and forgetful functors $\mathbb{C} \xrightarrow{L} \mathrm{Kl}_X \xrightarrow{U} \mathbb{D}$ inducing a relative adjunction

$$\mathrm{Kl}_X(LA, B) \cong \mathbb{D}(JA, UB). \tag{8}$$

We can define back and forth functors between $[\mathrm{Kl}_X, \mathbb{D}]$ and $[\mathbb{C}, \mathbb{D}]$ defined as follows:



By Lemma C.7, these functors are adjoint. It is straightforward to check that the induced monad is $- \otimes X$.

□

References

[1] Peter Aczel. 1978. A General Church-Rosser Theorem. https://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf
[2] J. Adámek and J. Rosicky. 1994. *Locally Presentable and Accessible Categories*. Cambridge University Press. doi:10.1017/CBO9780511600579
[3] Benedikt Ahrens. 2011. Modules over relative monads for syntax and semantics. *CoRR* abs/1107.5252 (2011).
[4] Benedikt Ahrens. 2012. Extended Initiality for Typed Abstract Syntax. *Log. Methods Comput. Sci.* 8, 2 (2012). doi:10.2168/LMCS-8(2:1)2012
[5] Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. 2020. Reduction monads and their signatures. *PACMPL* 4, POPL (2020), 31:1–31:29. doi:10.1145/3371099
[6] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2015. Monads need not be endofunctors. *Logical Methods in Computer Science* 11, 1 (2015). doi:10.2168/LMCS-11(1:3)2015
[7] Thorsten Altenkirch, Paul Blain Levy, and Sam Staton. 2010. Higher-Order Containers. In *Programs, Proofs, Processes, 6th Conference on Computability in Europe, CiE 2010, Ponta Delgada, Azores, Portugal, June 30 - July 4, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6158)*, Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes (Eds.). Springer, 11–20. doi:10.1007/978-3-642-13962-8_2
[8] Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1683)*, Jörg Flum and Mario Rodríguez-Artalejo (Eds.). Springer, 453–468. doi:10.1007/3-540-48168-0_32
[9] Françoise Bellegarde and James Hook. 1994. Substitution: A Formal Methods Case Study Using Monads and Transformations. *Sci. Comput. Program.* 23, 2-3 (1994), 287–311. doi:10.1016/0167-6423(94)00022-0
[10] Richard S. Bird and Ross Paterson. 1999. De Bruijn notation as a nested datatype. *Journal of Functional Programming* 9, 1 (1999), 77–91. doi:10.1017/S0956796899003366
[11] Peio Borthelle, Tom Hirschowitz, and Ambroise Lafont. 2020. A Cellular Howe Theorem. In *Proc. 35th ACM/IEEE Symposium on Logic in Computer Science*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM. doi:10.1145/3373718.3394738
[12] Marcelo Fiore and Chung-Kil Hur. 2009. On the construction of free algebras for equational systems. *Theoretical Computer Science* 410 (2009), 1704–1729. doi:10.1016/j.tcs.2008.12.052
[13] Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal Metatheory of Second-Order Abstract Syntax. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 53 (2022), 29 pages. doi:10.1145/3498715
[14] Marcelo P. Fiore and Makoto Hamana. 2013. Multiversal Polymorphic Algebraic Theories: Syntax, Semantics, Translations, and Equational Logic. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 520–529. doi:10.1109/LICS.2013.59
[15] M. P. Fiore and C.-K. Hur. 2010. Second-order equational logic. In *Proceedings of the 19th EACSL Annual Conference on Computer Science Logic (CSL 2010)*.
[16] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 193–202. doi:10.1109/LICS.1999.782615

[17] Murdoch Gabbay and Andrew M. Pitts. 1999. A New Approach to Abstract Syntax Involving Binders. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 214–224. doi:10.1109/LICS.1999.782617
[18] Joseph A. Goguen and James W. Thatcher. 1974. Initial algebra semantics. In *15th Annual Symposium on Switching and Automata Theory (SWAT)*. IEEE, 63–77.
[19] Joseph A. Goguen, James W. Thatcher, and E.G. Wagner. 1978. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, IV: Data Structuring*, R. Yeh (Ed.). Prentice-Hall, 80–144.
[20] John W. Gray. 1974. *Quasi-adjointness.* Springer Berlin Heidelberg, Berlin, Heidelberg, 166–271. doi:10.1007/BFb0061287
[21] Makoto Hamana. 2011. Polymorphic Abstract Syntax via Grothendieck Construction. In *Proc. 14th Foundations of Software Science and Computational Structures.*
[22] André Hirschowitz, Tom Hirschowitz, Ambroise Lafont, and Marco Maggesi. 2022. Variable binding and substitution for (nameless) dummies. In *Proc. 25th Foundations of Software Science and Computational Structures (Lecture Notes in Computer Science)*, Lutz Schröder and Patricia Bouyer (Eds.). Springer. doi:10.1007/978-3-030-99253-8_20
[23] André Hirschowitz, Tom Hirschowitz, Ambroise Lafont, and Marco Maggesi. 2024. Variable binding and substitution for (nameless) dummies. *Logical Methods in Computer Science* Volume 20, Issue 1 (March 2024). doi:10.46298/lmcs-20(1:18)2024
[24] André Hirschowitz and Marco Maggesi. 2007. Modules over Monads and Linearity. In *Proc. 14th International Workshop on Logic, Language, Information and Computation (Lecture Notes in Computer Science, Vol. 4576).* Springer, 218–237. doi:10.1007/978-3-540-73445-1_16
[25] André Hirschowitz and Marco Maggesi. 2010. Modules over monads and initial semantics. *Inf. Comput.* 208, 5 (2010), 545–564. doi:10.1016/J.IC.2009.07.003
[26] Tom Hirschowitz and Ambroise Lafont. 2022. A categorical framework for congruence of applicative bisimilarity in higher-order languages. *Logical Methods in Computer Science* 18, 3 (2022). doi:10.46298/lmcs-18(3:37)2022
[27] Bart Jacobs. 1999. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam.
[28] Niles Johnson and Donald Yau. 2021. *2-Dimensional Categories.* Oxford University Press. doi:10.1093/oso/9780198871378.001.0001
[29] Peter T. Johnstone. 2002. *Sketches of an Elephant: A Topos Theory Compendium - Volume 1.* Oxford University Press.
[30] Stephen Lack. 2010. *Towards Higher Categories.* The IMA Volumes in Mathematics and its Applications, Vol. 152. Springer, Chapter A 2-categories companion. doi:10.1007/978-1-4419-1524-5
[31] Thomas Lamiaux and Benedikt Ahrens. 2025. A Unified Framework for Initial Semantics. arXiv:2502.10811 [cs.PL] https://arxiv.org/abs/2502.10811
[32] F. William Lawvere. 1963. *Functorial semantics of algebraic theories.* Ph. D. Dissertation. Columbia University. http://tac.mta.ca/tac/reprints/articles/5/tr5abs.html
[33] F. E. J. Linton. 1966. Some Aspects of Equational Categories. In *Proceedings of the Conference on Categorical Algebra*, S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–94.
[34] Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (2nd ed.). Number 5 in Graduate Texts in Mathematics. Springer. doi:10.1007/978-1-4757-4721-8
[35] Ernest Manes. 1976. *Algebraic Theories.* Graduate Texts in Mathematics, Vol. 26. Springer.
[36] Ralph Matthes and Tarmo Uustalu. 2004. Substitution in non-wellfounded syntax with variable binding. *Theoretical Computer Science* 327, 1 (2004), 155–174. doi:10.1016/j.tcs.2004.07.025 Selected Papers of CMCS '03.
[37] Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223–255.
[38] John Power. 2007. Abstract Syntax: Substitution and Binders: Invited Address. *Electronic Notes in Theoretical Computer Science* 173 (04 2007), 3–16. doi:10.1016/j.entcs.2007.02.024
[39] John Power and Miki Tanaka. 2005. Binding Signatures for Generic Contexts. In *Typed Lambda Calculi and Applications*, Paweł Urzyczyn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–323.
[40] Emily Riehl. 2017. *Category theory in context.* Courier Dover Publications.
[41] Ross Street. 1972. The formal theory of monads. *Journal of Pure and Applied Algebra* 2 (1972), 149–168.
[42] Miki Tanaka and John Power. 2006. Pseudo-distributive laws and axiomatics for variable binding. *High. Order Symb. Comput.* 19, 2-3 (2006), 305–337. doi:10.1007/S10990-006-8750-X
[43] Julianna Zsidó. 2010. *Typed Abstract Syntax.* Theses. Université Nice Sophia Antipolis. https://theses.hal.science/tel-00535944

Received 2025-02-27; accepted 2025-06-27