Unbiasing symmetric monoidal categories in Lean

Robin Carlier*

February 28, 2026

Abstract

We present a formalization in Lean 4, within the framework of the mathematical library `Mathlib`, of the unbiasing process for symmetric monoidal categories. This is realized by extending the data of a symmetric monoidal category to a Cat-valued pseudofunctor from the (2,1)-category of spans of finite sets, encoding tensor products of higher arities and their coherences. The construction relies on a formalization of Mac Lane's coherence theorem using Piceghello's presentation of free symmetric monoidal categories as symmetric lists, and uses an encoding of universal formulas via an appropriate Kleisli bicategory.

1 Introduction

When setting up the basic algebraic hierarchy for mathematics, one defines a commutative monoid as a set M equipped with the data of a unit element 0_M and a commutative and associative composition law $\cdot +_M \cdot : M \times M \to M$. Given elements a, b, c, d of such a commutative monoid M, the data allows us to make formal sense of expressions such as $(a +_M b) +_M (c +_M d)$, $(a +_M (c +_M b)) +_M d$ or $b +_M (a +_M (d +_M c))$, and the properties satisfied by the function $+_M$ let us prove that all of these expressions are equal. For mathematical applications, it very quickly becomes important to extend the binary operation to operators with higher arities. For instance, a formula like

$$\sum_{\rho \in \mathrm{Irr}(G)} (\dim(\rho))^2 = \mathrm{Card}(G)$$

for a finite group G involves a summation operator that is defined over an arbitrary finite set of elements of the underlying monoid without a canonical order. The mere existence of such an operator requires a proof that it is well-defined, i.e., that the resulting value cannot depend on how the underlying list of summands is enumerated or associated.

*The author acknowledges support from the ANR project "HQDIAG", ANR-21-CE40-0015.

In the context of software proof assistants, libraries of formalized mathematics, such as `Mathlib` [19][12], have to go through that process in order to give their users a framework for working with such finite sums. For instance, `Mathlib` has a declaration

```
def Multiset.sum {M : Type u} [AddCommMonoid M] : Multiset M → M := …
```

that realizes the sum over a multiset of elements of `M` by using the definition of `Multiset M` as a quotient of the type `List M` modulo permutations. The declaration `Finset.sum` then builds on that definition to provide unbiased sums to users.[1]

From commutative monoids to symmetric monoidal categories

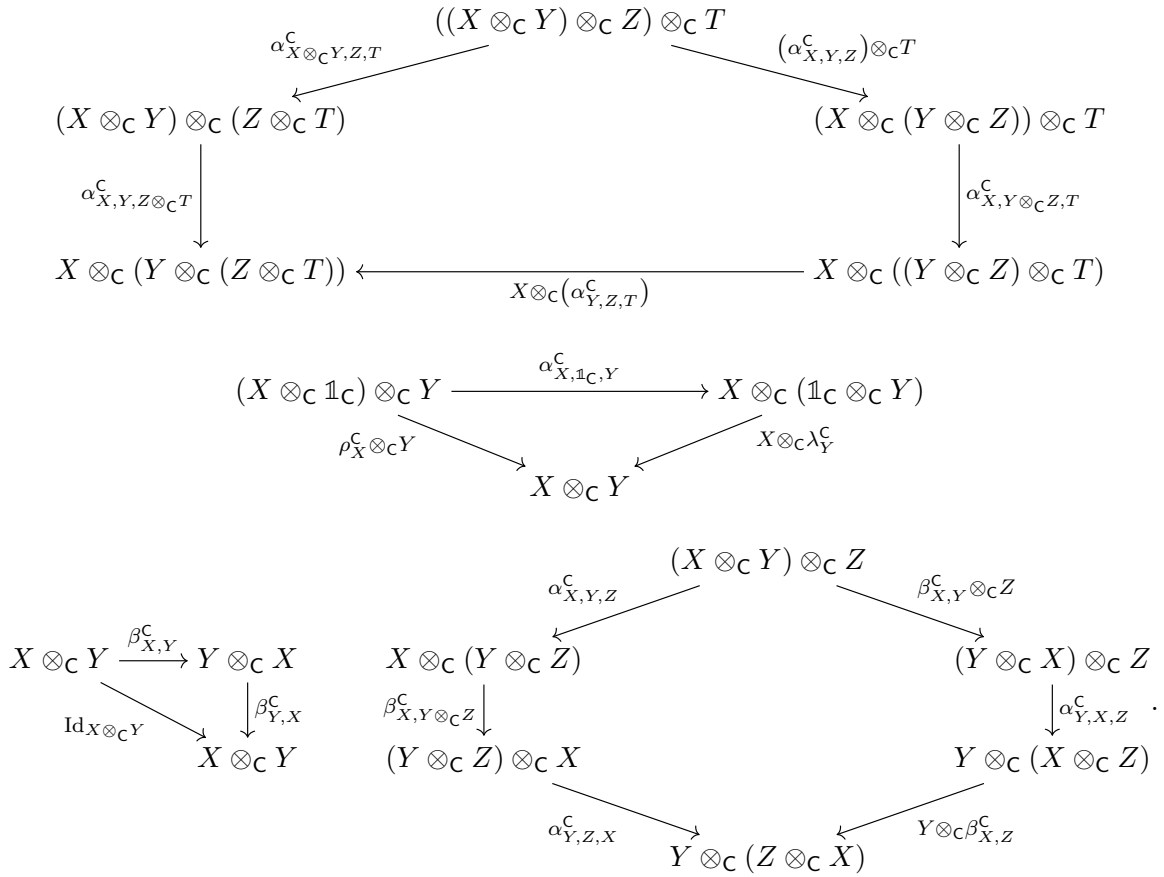
The natural categorification of commutative monoids is symmetric monoidal categories. Symmetric monoidal categories were independently introduced by Saunders Mac Lane [15] as an abstraction of the structure shared by the categorical products in categories with products, the tensor product on modules and the tensor product of chain complexes. This well-studied concept is now considered a classical example of one of the many *pseudoalgebraic* structures one can put on a category.

The classical definition of a symmetric monoidal category ([15], [5, Def. 6.1.2]) consists of a category C equipped with a tensor product bifunctor $\otimes_{\mathsf{C}} : \mathsf{C} \times \mathsf{C} \to \mathsf{C}$, a unit object $\mathbb{1}_{\mathsf{C}}$ and for all objects X, Y, Z of C, natural isomorphisms

$$\begin{aligned} \alpha^{\mathsf{C}}_{X,Y,Z} &: (X \otimes_{\mathsf{C}} Y) \otimes_{\mathsf{C}} Z \xrightarrow{\sim} X \otimes_{\mathsf{C}} (Y \otimes_{\mathsf{C}} Z) \\ \lambda^{\mathsf{C}}_{X} &: \mathbb{1}_{\mathsf{C}} \otimes_{\mathsf{C}} X \xrightarrow{\sim} X \\ \rho^{\mathsf{C}}_{X} &: X \otimes_{\mathsf{C}} \mathbb{1}_{\mathsf{C}} \xrightarrow{\sim} X \\ \beta^{\mathsf{C}}_{X,Y} &: X \otimes_{\mathsf{C}} Y \xrightarrow{\sim} Y \otimes_{\mathsf{C}} X \end{aligned}$$

respectively called associators, left unitors, right unitors, and braidings, such that the following diagrams commute:

[1] These sums still bundle a specific choice regarding associativity, coming from the fact that `List.sum` is defined as folding the sum operator on the left. The irrelevance of that choice is then proven in e.g., `Finset.sum_disjUnion`. Following the thread of proofs, it reduces to `Multiset.fold_add`, which is where associativity ends up being used.



Unlike the case of commutative monoids, associativity and commutativity in symmetric monoidal categories are expressed as a *structure* instead of a *property*. This is part of the standard philosophy of category theory: equality of objects in a category is a notion that is not invariant under equivalences of categories and one should instead replace the property of being equal with the structure of a chosen isomorphism. The properties appear one categorical level higher as commutativity of diagrams involving the isomorphisms.

The presentation above gives a rather easy and straightforward way to define symmetric monoidal categories: only low-arity data and coherences need to be supplied. On the other hand, similarly to the case of commutative monoids, some applications require an unbiased point of view on symmetric monoidal categories and require availability of tensor products and coherences in every possible arity. For instance, given a symmetric monoidal category C with countable coproducts and such that the tensor product of C preserves suitable colimits in each variable, the free commutative monoid on an object $c \in \mathsf{C}$ can be identified with an $\mathbb{N}$-indexed coproduct of objects $\mathrm{Sym}^{\mathsf{C}}_n(c)$ where $\mathrm{Sym}^{\mathsf{C}}_n$ sends an object a to the colimit of the $\mathrm{B}\Sigma_n$-indexed diagram in C that corresponds to the object $a^{\otimes n}$ with its natural Σ_n action.

This fact cannot even be stated unless the binary tensor product of C and its symmetries have been extended to higher arities.

This formality is usually resolved by invoking Mac Lane's celebrated coherence theorem [15, Thm. 4.2] which loosely states that there is only one "canonical" isomorphism made out of associators and unitors between two iterated tensor products that only differ by bracketing, and only one "canonical" morphism attached to a given permutation of the variables of an iterated tensor product. We are being intentionally vague with the statement here as we will be giving a more precise formulation in the course of this paper (see Theorem 1.3).

Symmetric monoidal higher categories

Another very important application of unbiased symmetric monoidal categories comes in the form of the higher-categorical point of view on symmetric monoidal categories. The definition of symmetric monoidal categories we gave in the previous paragraph does not scale very well as we increase the categorical level: the triangle, pentagon, hexagon and symmetry identities are about equalities of (1-)morphisms in C; if C is replaced by a bicategory, then all these equalities need to be replaced by the data of suitable 2-isomorphisms in C and new coherence identities need to be satisfied by these 2-isomorphisms (see e.g., [18, Appendix C] for an algebraic description of symmetric monoidal bicategories). To further convince oneself that this algebraic approach can hardly scale further in practice, one can look at the definition of a (non-symmetric) monoidal tricategory as a one-object tetracategory in [10].

In the context of the theory of $(\infty, 1)$-categories, several equivalent notions of symmetric monoidal $(\infty, 1)$-categories have been studied: Lurie's ∞-operads [13], Moerdijk and Weiss's dendroidal sets [16], Cranch's Lawvere symmetric monoidal $(\infty, 1)$-categories [6], to cite only a few.

All of the models mentioned above are similar in that the data of a symmetric monoidal $(\infty, 1)$-category is interpreted (either directly, or via suitable comparison theorems) as a pseudofunctor to the $(\infty, 1)$-category of $(\infty, 1)$-categories (or, equivalently, fibrations), with source a diagram category that contains already higher arity data and symmetries: the category of pointed finite sets (a.k.a. Segal's category Γ) for ∞-operads, the category of rooted trees Ω in the dendroidal sets setting and the (2,1)-category of spans of finite sets for Cranch's Lawvere symmetric monoidal $(\infty, 1)$-categories. In all these approaches, these source categories encode all the diagrams that are expected to "commute" in the resulting symmetric monoidal $(\infty, 1)$-category and packaging the data as an ∞-functor to Cat_∞ realizes the diagrams.

One of the important aspects of $(\infty, 1)$-category theory is that, when a notion is applied to objects that come from ordinary categories, it reduces in almost all cases to the corresponding notion in ordinary category theory: for instance, when viewing an ordinary category C as an $(\infty, 1)$-category, the ∞-groupoid of morphisms from x to y is isomor-

phic to the set of morphisms from x to y in C when viewing sets as discrete ∞-groupoids. Similarly, $(\infty, 1)$-categorical notions of limits and colimits reduce to ordinary limits and colimits when the source and targets of the diagrams are ordinary categories. Symmetric (and non-symmetric) monoidal categories are no exception to this principle: when viewed as an $(\infty, 1)$-category, the data of a symmetric monoidal ordinary category is equivalent to the data of a symmetric monoidal $(\infty, 1)$-category. Such a statement cannot avoid the "unbiasing" step that extends the binary tensor product to any finite family of objects with suitable inner symmetries. Mac Lane's coherence theorem is usually mentioned when this step occurs, although its invocation may not always be very explicit ([13, Construction 2.0.0.1, (ii) and (iii)]). We refer the reader to [11] for one of the few places in the literature where the full process of unbiasing symmetric monoidal categories is actually spelled out, though not in terms of higher categories.

State of current formalizations in Lean

In the Lean 4 proof assistant, and more specifically in the mathematics library `Mathlib`, symmetric monoidal categories have been formalized using the classical biased approach. This is very reasonable, as this is the most elementary approach (`Mathlib`'s definition could be performed in a file that imports nothing but the definition of categories and the definition of isomorphisms) as well as the one that leads to constructors with the least data and proof obligations. This ease of definition comes at the price that, while elementary manipulation can be performed in a nice way, some more complex constructions cannot easily be performed, for instance, the description of free commutative monoid objects we mentioned earlier. Another reason to seek an unbiased definition is to provide the necessary "glue" for the higher-categorical perspectives mentioned previously. It seems likely that `Mathlib` will eventually include foundational results for higher categories, making this connection essential. This feeling is based on the observation that `Mathlib` already has definitions relevant to quasi-categorical foundations of the theory of higher categories such as quasi-categories, simplicial nerves of simplicial categories, a growing amount of simplicial homotopy theory, as well as the existence of an ongoing project to formalize Riehl-Verity's theory of ∞-cosmoi and formal category theory inside an ∞-cosmos. As this theory develops, it will become important to be able to reinterpret objects coming from the already existing formalization of ordinary category theory that is available in `Mathlib` as the corresponding higher-categorical objects.

The coherence theorem for (non-symmetric) monoidal categories is already formalized in `Mathlib`, but no formal unbiasing of even non-symmetric monoidal categories is available.

Results

Borrowing terminology from [14, Def. 00AL], we will call the *pith* of a bicategory B the bicategory that is obtained by discarding non-invertible 2-morphisms in B; we will denote

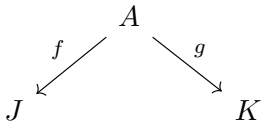
it by $\mathrm{Pith}(\mathsf{B})$. Given a category C with pullbacks, we will denote by $\mathrm{Span}(\mathsf{C})$ the bicategory of spans in C.

The main result that our project formalizes is the following:

Theorem 1.1. *Let* $\left(\mathsf{C}, \otimes_{\mathsf{C}}, \mathbb{1}_{\mathsf{C}}, \alpha^{\mathsf{C}}, \lambda^{\mathsf{C}}, \rho^{\mathsf{C}}, \beta^{\mathsf{C}}\right)$ *be a symmetric monoidal category. There exists a pseudofunctor*

$$\mathsf{C}^{\otimes} : \mathrm{Pith}\left(\mathrm{Span}\left(\mathrm{Fin}\right)\right) \to \mathrm{Cat}$$

that sends a finite set J *to the product category* C^J*, and sends a span*



to the functor $g_! f^*$*. These functors are defined as follows:*

- *The functor* $f^* : \mathsf{C}^J \to \mathsf{C}^A$ *is precomposition along* f*.*
- *The functor* $g_! : \mathsf{C}^A \to \mathsf{C}^K$ *sends an* A*-indexed family* $(x_a)_{a \in A}$ *to the* K*-indexed family*

$$\left(\bigotimes_{a \in g^{-1}(\{k\})} x_a \right)_{k \in K}.$$

This theorem expresses that a symmetric monoidal category as defined in the usual biased version defines a Lawvere symmetric monoidal $(\infty, 1)$-category (or rather, an appropriate ordinary-categorical version of this) in the terminology of [6].

Remark 1.2. Some readers might be surprised by our choice of stating this result in terms of Cranch's Lawvere symmetric monoidal categories instead of Lurie's ∞-operads, the latter being the most used model in the $(\infty, 1)$-categorical literature. The full comparison theorem stating that the $(\infty, 1)$-categories of Lawvere symmetric monoidal $(\infty, 1)$-categories and Lurie-style symmetric monoidal $(\infty, 1)$-categories are equivalent requires some work (see [2, Prop. C.1] for a rather concise proof). Fortunately, the easiest direction of the comparison result is the one that turns a Lawvere symmetric monoidal $(\infty, 1)$-category into a Lurie-style symmetric monoidal $(\infty, 1)$-category: this is achieved by precomposition along an explicitly defined pseudofunctor from Fin_* (seen as a locally discrete bicategory) to $\mathrm{Pith}\left(\mathrm{Span}\left(\mathrm{Fin}\right)\right)$, followed by taking the Grothendieck construction (which is the easy direction of the straightening/unstraightening equivalence). Hence, our result would easily translate into the construction of a pseudofunctor out of the category of pointed finite sets. On the other hand, constructing something out of the pith of the bicategory of spans of finite sets from the data of a suitable pseudofunctor out of pointed finite sets would be more challenging, as this would involve performing a right Kan extension of a pseudofunctor along another pseudofunctor rather than restricting it.

We believe that this work constitutes the first mechanized proof of the statement in Theorem 1.1.

As announced earlier in this introduction, a primary ingredient in proving the above theorem is the coherence theorem for symmetric monoidal categories. We implement the following form of this theorem:

Theorem 1.3. *Let J be a set. The free symmetric monoidal category on J is equivalent (as a symmetric monoidal category) to the category* $\mathrm{Core}\left(\mathrm{Fin}_{/J}\right)$ *of finite sets and bijections over J endowed with the symmetric monoidal structure induced by the cocartesian symmetric monoidal structure on sets.*

S. Piceghello previously formalized part of the coherence theorem in the setting of Homotopy Type Theory in [17] by identifying the free symmetric monoidal category on a type with the groupoid of symmetric lists, but the final link between morphisms of symmetric lists and permutation groups is only partially formalized in his work. We contribute the necessary adaptations and translations of Piceghello's work from the Homotopy Type Theory framework into the non-HoTT framework of categories in Lean/`Mathlib`, and our main contribution here is the formalized identification of morphisms of symmetric lists with permutations via labellings of their morphisms by elements of suitable Coxeter groups (Theorem 2.8 and Corollary 2.9).

We also believe our work is of interest in explaining *how* to precisely leverage the coherence theorem to obtain Theorem 1.1, and clarifying this process. The general ideas are certainly known to the mathematical community, but the actual details are more often than not left to the reader.

Our formalization is available online[2]. The full repository is about 12.5kLoC, excluding comments. Our code compiles using the lean toolchain `leanprover/lean4:v4.28.0`, and depends on `Mathlib`'s commit `8f9d9cff6bd728b17a24e163c9402775d9e6a365`.

Outline

In Section 2, we focus on the coherence theorem 1.3. We first introduce the category of symmetric lists in 2.1, adapting the work in [17] to the setting of Lean/`Mathlib`. This category will play a central role in the rest of the paper as a convenient model of the free symmetric monoidal category on a set. In 2.2 we study morphisms in the category of symmetric lists: we first briefly recall and formalize the link between Coxeter groups of type A_n and permutations in 2.2.1 and make use of this material in 2.2.2 to formalize that morphisms of symmetric lists are in one-to-one correspondence with permutations by showing that they can be faithfully labeled as elements of a Coxeter group of type A_∞ (Theorem 2.8). In 2.3, we follow Piceghello's method to prove that symmetric lists are indeed free symmetric monoidal categories, which completes the proof of Theorem 1.3.

[2] https://github.com/robin-carlier/SymmMonCoherence/tree/2070e1d536854a49d4c4f98cd73857ba40c4cb92

In Section 3, we recall the basics of bicategories of spans and explain our formalization of them. Then, we introduce in Definition 3.2 the precise structure needed to define pseudofunctors out of the pith of the span bicategory of a category with pullbacks.

In Section 4 we construct the necessary data to build the pseudofunctor of Theorem 1.1 using the machinery from Section 3. In 4.1, we first abstract the target bicategory from Cat to a suitable Kleisli bicategory $\mathcal{K}^{\mathrm{op}}_{\mathrm{SList}}$ for the theory of symmetric monoidal categories and we show that one can interpret a symmetric monoidal category as a Cat-valued pseudofunctor from the opposite of the Kleisli bicategory (Proposition 4.2). In 4.3, we explain how to leverage the coherence theorem 1.3 to show that in good cases, computations in the Kleisli bicategory behave like matrix computations with multisets. Finally, we build a pseudofunctor $\Lambda : \mathrm{Pith}\,(\mathrm{Span}\,(\mathrm{Fin})) \to \mathcal{K}^{\mathrm{op}}_{\mathrm{SList}}$ in Proposition 4.12, finalizing the construction of the pseudofunctor described in Theorem 1.1.

2 The coherence theorem

2.1 Symmetric lists: presenting a category via generators and relations

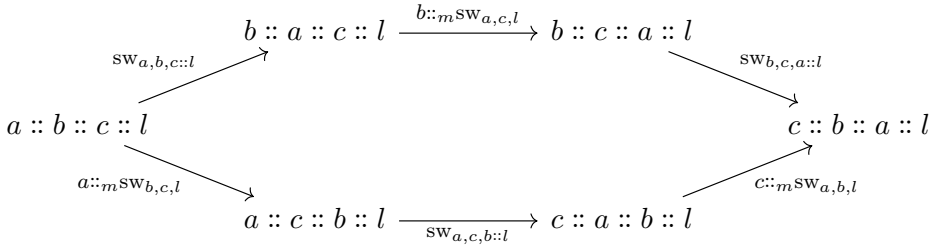
The coherence theorem for (non-symmetric) monoidal categories was already formalized in `Mathlib` by Markus Himmel, following a classical proof by normalization due to Beylin and Dybjer [3] that was suited for type-theory based proof assistants and originally formalized in the proof assistant ALF.

One of the key ideas of this proof is to make the statement of the theorem a statement about free monoidal categories on types: such free monoidal categories are groupoids and have at most one morphism between any pair of objects. By construction, the morphisms in the free monoidal category on a type C are exactly those one can build out of associators, left or right unitors and tensor products of other morphisms. The classical statement that "every diagram commutes" is thus encoded by this statement. To prove this property of morphisms in free monoidal categories, one normalizes objects in the free monoidal category on a type T to essentially show that it is equivalent to the discrete category on the set of lists.

The case of symmetric monoidal categories is harder as one cannot hope for the free symmetric monoidal category to have at most one morphism between objects: there are symmetric monoidal categories C where the identity $\beta^{\mathsf{C}}_{c,c} = \mathrm{Id}_{c\otimes c}$ does not hold, despite the morphisms having the same source and target. It remains reasonable to expect that free symmetric monoidal categories can have their associative and unital parts normalized into a list-like structure (as one can do so after forgetting the braidings), but the presence of braiding isomorphisms means that the resulting category of normal objects will not be discrete and will yield a non-trivial category structure on lists. Piceghello presented this category structure [17, Def. 4.11] as symmetric lists, a direct categorification of multisets, which we describe here:

Definition 2.1. *[17, Def. 4.11] Let J be a set. The category of* symmetric lists *on J, denoted* $\mathrm{SList}\,(J)$ *is defined as the category presented by the following generators and relations:*

- *Objects of* $\mathrm{SList}\,(J)$ *are lists of elements of J.*
- *Morphisms are generated by the following (inductive) rules:*
  1. *For all $a, b \in J$ and $l : \mathrm{SList}\,(J)$, there is a morphism*
  $$\mathrm{sw}_{a,b,l} : (a :: b :: l) \to (b :: a :: l).$$
  2. *If $f : a \to b$ is a morphism, there is a morphism $(x ::_m f) : (x :: a) \to (x :: b)$.*
- *Morphisms are subject to the following relations:*
  1. *The constructions $f \mapsto x ::_m f$ respect compositions and identities.*
  2. *The morphism $\mathrm{sw}_{a,b,l}$ is natural in l (when interpreting $x \mapsto (x :: -)$ as a functor using the previous relation).*
  3. *The symmetry relation $\mathrm{sw}_{b,a,l}\mathrm{sw}_{a,b,l} = \mathrm{Id}_{a::b::l}$ holds.*
  4. *The diagram*



  commutes for every $a, b, c \in J$ and $l \in \mathrm{SList}(J)$.

In the setting of Homotopy Type Theory, in which [17, Def. 4.11] takes place, this definition can be realized at once as a (1-truncated) higher inductive type. In the setting of `Mathlib`, we cannot make such a definition directly, and instead have to manually translate and interpret the 1-truncated HoTT definition in the model of groupoids provided by `Mathlib`.

Therefore, formalizing the definition above requires several intermediate steps, which together constitute the standard construction of a category presented via generators and relations. In the infrastructure provided by `Mathlib`, the steps take the following form

1. Define a type `V` of generators for the objects, and introduces a `Quiver` instance on this type of generators, endowing it with generating arrows.

2. Take the path category `Paths V` on the quiver from the previous step, giving a free category where arrows are formal compositions (paths) of generating morphisms.

3. Define a relation `homRel` between morphisms in `Paths V` corresponding to the relations in the final category. In `Mathlib`, the type of such relations is `HomRel (Paths V)`.

4. Take the quotient category of `Paths V` by the relation `homRel` from the previous point.

We carry out this program in our case:

```
inductive SListQuiv (C : Type u) where
  | nil : SListQuiv C
  | cons (head : C) (tail : SListQuiv C) : SListQuiv C

infixr:67 " ::… " ⇒ SListQuiv.cons

inductive Hom {C : Type u} : SListQuiv C → SListQuiv C → Type u
  | swap (x y : C) (l : SListQuiv C) :
      Hom (x ::… (y ::… l)) (y ::… (x ::… l))
  | cons (z : C) {l l' : SListQuiv C} :
      Hom l l' → Hom (z ::… l) (z ::… l')

instance : Quiver (SListQuiv C) where Hom := Hom
structure FreeSListQuiv where p : Paths (SListQuiv C)
instance : Category (FreeSListQuiv C) :=
  inferInstanceAs (Category ◁ InducedCategory _ FreeSListQuiv.p)
```

In the HoTT setting, the constructor `Hom.cons` is implicit: the fact that it is a function in HoTT automatically gives the constructor `cons` an action on paths. When interpreting this definition in a non-HoTT setting, extra constructors have to be added, and it has to be extended manually to an endofunctor on the paths category in order to represent its action on paths. In the code block below, `ι C` is the inclusion prefunctor from `SListQuiv C` to `FreeSListQuiv C`, `β₁_` is a notation for the generating swap morphism, and `::_` (resp. `::_m`) is the action on objects (resp. morphisms) of the extension to `FreeSListQuiv C` of the `cons` constructor.

```
inductive HomEquiv : HomRel (FreeSListQuiv C)
  | swap_naturality (X Y : C) {l l' : SListQuiv C} (f : l ⟶ l') :
      HomEquiv
        (β₁_ X Y ((ι C).obj l) ≫
          (Y ::_m (X ::_m ((ι C).map f))))
        ((X ::_m (Y ::_m ((ι C).map f))) ≫
          (β₁_ X Y ((ι C).obj l')))
  | swap_swap (X Y : C) (l : FreeSListQuiv C) :
      HomEquiv (β₁_ X Y l ≫ β₁_ Y X l) (𝟙 _)
  | triple (X Y Z : C) (l : FreeSListQuiv C) :
      HomEquiv
        (β₁_ X Y (Z ::_ l) ≫ (Y ::_m (β₁_ X Z l)) ≫ β₁_ Y Z (X ::_ l))
        ((X ::_m (β₁_ Y Z l)) ≫ β₁_ X Z (Y ::_ l) ≫ Z ::_m (β₁_ X Y l))
```

```
  | cons (X : C) {l l' : FreeSListQuiv C} (f f' : l ⟶ l') :
      HomEquiv f f' → HomEquiv (X ::_m f) (X ::_m f')

def SList := CategoryTheory.Quotient (FreeSListQuiv.HomEquiv C)
  deriving Category
```

Again, due to the non-HoTT nature of `Mathlib`'s framework, extra relations have to be added in order to make the definition sound: `HomEquiv.swap_naturality` and `HomEquiv.cons` are not explicitly present in the HoTT presentation, as higher inductive types in HoTT ensure that all constructors are natural and functorial with respect to paths.

Remark 2.2. Besides the adaptations from the HoTT setting to the interpretation in the non-HoTT setting of Lean, there are two minor differences between our construction and Piceghello's.

1. If we were to strictly interpret everything in the "groupoid model" provided by `Mathlib`'s groupoids, we would need to replace the free category on `SListQuiv` by the free groupoid on `SListQuiv` to ensure that we stay within the realm of groupoids. This would add extra generators for formal inverses of the generating morphisms, as well as extra relations ensuring that the formal inverses define inverses in the quotient category. In the case of symmetric lists, it is easy to show that the category resulting from our definitions is a groupoid, and so we prefer the more direct construction in the setting of categories.

2. Our definition is less general: if we were to strictly interpret the construction in [17] in the groupoid model, we would be building symmetric lists on a groupoid, rather than symmetric lists on a set (which would correspond to the case of a discrete groupoid). We would then need extra relations ensuring naturality of the generating morphisms with respect to every parameter in C, making the presentation of the category and the study of its morphisms more complex. For the purposes of stating our version of the coherence theorem and unbiasing symmetric monoidal categories, this extra generality is not needed, so we do not implement it.

A version of the universal property of the category of symmetric lists as a category presented by generators and relations can be spelled out explicitly as the following:

Lemma 2.3. *Let* D *be a category.*

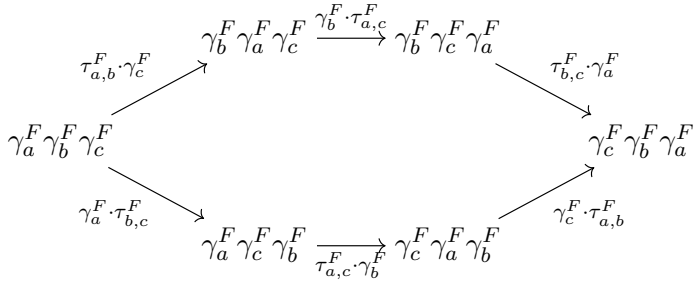
(i) Suppose given the following data:

- *An object* $F_{\mathrm{nil}} \in \mathsf{D}$.
- *For every* $c \in C$, *an endofunctor* $\gamma_c^F : \mathsf{D} \to \mathsf{D}$.
- *For all* $a, b \in C$, *a natural isomorphism*

$$\tau_{a,b}^F : \gamma_a^F \gamma_b^F \xrightarrow{\sim} \gamma_b^F \gamma_a^F .$$

Assume the data satisfies the following conditions:

- *For all $a, b \in C$, the equality $\tau^F_{a,b}\tau^F_{b,a} = \mathrm{Id}$ holds.*
- *For all $a, b, c \in C$, the diagram*



  commutes.

This data defines a functor $F : \mathrm{SList}(C) \to \mathsf{D}$ equipped with isomorphisms

$$\begin{array}{lrl} \upsilon_{\mathrm{nil},F} : & F([\,]) & \simeq F_{\mathrm{nil}}, \\ \upsilon_{\mathrm{cons},F,c} : & F \circ (c :: \cdot) & \simeq \gamma^F_c \circ F, \end{array}$$

and such that $F(\mathrm{sw}_{a,b,l})$ is identified with $\left(\tau^F_{a,b}\right)_{F(l)}$ through these isomorphisms.

(ii) Let $F, G : \mathrm{SList}(C) \to \mathsf{D}$ be functors. Suppose we are given the following data:

- *A morphism $\phi_{\mathrm{nil}} : F([\,]) \to G([\,])$.*
- *For every $c \in C$, $l \in \mathrm{SList}(C)$ and morphism $\mathrm{ind} : F(l) \to G(l)$, a morphism $\phi_{c,l}(\mathrm{ind}) : F(c :: l) \to G(c :: l)$.*

Assume further that the following conditions hold:

- *For all $a, b \in C$, for all $l \in \mathrm{SList}(C)$ and for all morphisms $\mathrm{ind} : F(l) \to G(l)$, the diagram*

$$\begin{array}{ccc} F(a :: b :: l) & \xrightarrow{F(\mathrm{sw}_{a,b,l})} & F(b :: a :: l) \\ {\scriptstyle \phi_{a,b::l}(\phi_{b,l}(\mathrm{ind}))}\downarrow & & \downarrow{\scriptstyle \phi_{b,a::l}(\phi_{a,l}(\mathrm{ind}))} \\ G(a :: b :: l) & \xrightarrow[G(\mathrm{sw}_{a,b,l})]{} & G(b :: a :: l) \end{array}$$

  commutes.

- *Given $c \in C$, a morphism $f : l \to l'$ in $\mathrm{SList}(C)$ and morphisms $\mathrm{ind}_l : F(l) \to G(l)$, $\mathrm{ind}_{l'} : F(l') \to G(l')$ satisfying $\mathrm{ind}_{l'} \circ F(f) = G(f) \circ \mathrm{ind}_l$, the diagram*

$$\begin{array}{ccc} F(c :: l) & \xrightarrow{F(c ::_m f)} & F(c :: l') \\ {\scriptstyle \phi_{c,l}(\mathrm{ind}_l)}\downarrow & & \downarrow{\scriptstyle \phi_{c,l'}(\mathrm{ind}_{l'})} \\ G(c :: l) & \xrightarrow[G(c ::_m f)]{} & G(c :: l') \end{array}$$

  commutes.

  The data defines a unique natural transformation $\phi : F \to G$ such that $\phi_{[\,]} = \phi_{\mathrm{nil}}$ and such that $\phi_{c::l} = \phi_{c,l}(\phi_l)$.

We implement the first point of the above lemma using a dedicated structure `RecursiveFunctorData` encapsulating the data defined in the first point, giving rise to a declaration `RecursiveFunctorData.functor` providing the functor attached to the data. The second point is implemented as a declaration `recNatTrans` taking directly the necessary data as parameter.

Remark 2.4. Our statement of Lemma 2.3 is bicategorical in the sense that it characterizes functors out of the category of symmetric lists up to a unique isomorphism. In fact, the data in the first point defines functors *uniquely*, and the isomorphisms υ_{nil} and υ_{cons} that characterize the resulting functors can be (componentwise) definitional equalities. We intentionally avoid stating things this way, for reasons we explain below.

A recurring theme when formalizing category theory in the setting of a dependent type theory like Lean is that, generally speaking, equalities of objects of categories should be avoided when possible. One of the reasons is that types of morphisms, and hence functions like composition of morphisms, actively depend on objects. Given a category `C` and terms `x, y, z : C`, in the presence of an equality `h : x = y` in context, a morphism `f : x → z` will not directly type check as a morphism `y → z` and a casting operation has to be performed on `f`. When the equality `h` is an equality of free variables in contexts, performing a cast is easily done. If `h` is a more complex expression, direct substitution is usually not possible, and, while it is *theoretically* possible to use the induction principle on equalities with carefully crafted induction motives to perform substitutions, it is in practice extremely tedious to do so, especially since this has to be done repeatedly in every proof where the situation arises. Situations like these are colloquially referred to as "DTT Hell" within the Lean community.

To make the situation with equality of objects slightly more manageable, `Mathlib` made the choice to provide casts only for identity morphisms, these are called `eqToHom`: given `h : x = y`, `eqToHom h : x → y` is the equality `h` as a morphism. Usage of `eqToHom` as a correction term when composing morphisms whose sources and targets only match up to propositional equality is a standard method in `Mathlib`, but it only partially alleviates

the inherent trouble of working with equalities of objects and it is still considered better practice to try to not end up in situations where these are needed in the first place.

When an equality of objects `h : x = y` is *definitional*, the situation is better and the type checker will accept `f : x → z` as a valid morphism `y → z`, but there remains an important technical subtlety: in Lean, for performance reasons, checks for definitional equalities depend on a setting called *transparency*, which controls how definitions are allowed to be unfolded when checking for definitional equality of terms. Most of the automation tactics in lean (`simp`, `grind`, `rw`, etc.,) work at the "reducible" transparency level, where most definitions are *not* unfolded. If a definitional equality `h : c = c'` does not hold at reducible transparency (for instance, if it is gated behind a `def`), tactics may be unable to use terms that depends on it. Over-reliance on non-reducible definitional equalities is often colloquially referred to as "defeq abuse" within the Lean community.

In our situation, the definitional equalities involving `RecursiveFunctorData.functor` may hold, but not at reducible transparency and relying on them could cause problems when trying to automate proofs and computations involving such recursively-defined functors. Hence, we take the opinionated route of considering that these definitional equalities are an implementation detail, and reflect this in our statement of Lemma 2.3.

In practice, in our implementation, we leverage the Lean module system introduced in `v4.26.0` to make it so that the body of `RecursiveFunctorData.functor` is not "exposed": this ensures that no declaration in the "public" scope can use the definitional equalities that underlie υ_{nil} and the components of υ_{cons}. We still export lemmas about existence of equalities in the public scope, as they can provide a convenient shortcut to show that some diagrams made exclusively of morphisms of the form υ_{nil} and υ_{cons} commute, but their definitional nature is sealed away, making it impossible to abuse.

2.2 Studying morphisms of symmetric lists

The key point in Mac Lane's original proof of the coherence theorem is the idea that the hexagon and symmetry relations correspond to the relations that present the symmetric group on $n+1$ letters Σ_{n+1} as a Coxeter group of type A_n: this presentation is the isomorphism

$$\left\langle (s_i)_{1\leq i\leq n} \;\middle|\; \begin{array}{rcll} s_i^2 & = & e & \\ s_i s_{i+1} s_i & = & s_{i+1} s_i s_{i+1} & \\ s_i s_j & = & s_j s_i & \text{if } |i-j|>1 \end{array} \right\rangle \cong \Sigma_{n+1}$$

that sends the generator s_i to the permutation $(i\ i+1)$ (once the set of letters has been enumerated). One can indeed see a clear link between this presentation and our categories of symmetric lists: given a symmetric list $[x_0,\dots,x_n]$, we could label s_0 a morphism of the form

$$\mathrm{sw}_{x,y,[x_2,\dots,x_n]} : [x,y,\dots,x_n] \to [y,x,\dots,x_n]$$

which conveniently happens to realize the permutation (0 1) on the indices of the list, and the symmetry relation would then read $s_0^2 = 1$. We could also label s_1 any morphism of the form $x :: \mathrm{sw}_{y,z,[x_3,\ldots,x_n]}$, and the hexagon relation would then conveniently read as $s_0 s_1 s_0 = s_1 s_0 s_1$. Finally, labelling e.g., s_2 a morphism of the form $x :: y :: \mathrm{sw}_{z,t,[x_4,\ldots,x_n]}$, the relation on morphisms of symmetric lists that asserts naturality of the swap would then read as the equation $s_0 s_2 = s_2 s_0$.

The procedure described above provides the core idea for establishing a correspondence between morphisms of symmetric lists and permutations. However, this idea must be refined before it becomes formal enough for a proof assistant to accept.

2.2.1 Coxeter groups of type A_n and permutation groups

Before even attempting to formalize the above procedure, the link between Coxeter groups of type A_n and permutations of the set $\{0, \ldots, n\}$ needs to be formalized. The library `Mathlib` defines a Coxeter system as the structure of an isomorphism between a group and the group presented by generators and relations from a Coxeter matrix. For `Mathlib`, a Coxeter matrix is a (possibly infinite) square matrix M on a set I with positive integer coefficients, that is symmetric, such that all diagonal coefficients are one and such that all off-diagonal coefficients are not equal to one.

Every Coxeter matrix M on a set I defines a group $\overline{M}$ by quotienting the free group on I by the *Coxeter relations* $\left((s_i s_j)^{M_{ij}} = e\right)_{i,j \in I^2}$, where s_i is the generator corresponding to $i \in I$.

In this work, we will mainly be working with the Coxeter matrix A_n on the set $\{0, \cdots, n-1\}$, represented by

$$\mathrm{A}_n = \begin{pmatrix} 1 & 3 & 2 & \cdots & 2 \\ 3 & 1 & 3 & \ddots & \vdots \\ 2 & 3 & 1 & \ddots & 2 \\ \vdots & \ddots & \ddots & \ddots & 3 \\ 2 & \cdots & 2 & 3 & 1 \end{pmatrix}$$

as well as with its infinite variant A_∞ (as a Coxeter matrix on the set $\mathbb{N}$).

At the time of writing, despite some theory on abstract Coxeter systems, `Mathlib` does not produce any non-trivial term of type `CoxeterSystem G` (non-trivial meaning here that G is not definitionally the group presented by a Coxeter matrix). Following the textbook proof of [4, §1.5], we formalize a (classical) criterion to determine when a given set of degree 2 generators in a group G satisfying correct relations extends to a presentation of G as a Coxeter group: we introduce a structure

```
/- Below, M.simple i is the element of the Coxeter group attached to M that
corresponds to the generator `i`. -/
```

```
structure PreCoxeterSystem {B : Type*} (M : CoxeterMatrix B) (G : Type*) [Group G] where
  hom : M.Group →* G
  surjective_hom : Function.Surjective hom
  orderOf_eq (i j : B) : M i j = orderOf (hom (M.simple i) * hom (M.simple j))
  hom_simple_ne_one (i : B) :  hom (M.simple i) ≠ 1
```

encoding a set of degree 2 generators in G satisfying suitable equations as a surjective group homomorphism from the corresponding Coxeter group to G. The criterion is then called the *exchange property* [4, p. 18], which we explain with Lean code in the listing below:

```
variable {B : Type*} (M : CoxeterMatrix B) (G : Type*) [Group G] (S : PreCoxeterSystem M G)
/-- The length of an element `g` is the minimal length (in the sense of Coxeter systems)
of the preimages of `g` in `M.Group`. This corresponds to the minimal length of
a word of generators needed to express `g`. -/
noncomputable def length (g : G) : ℕ :=
  Nat.find ◁ show (M.toCoxeterSystem.length '' (S.hom⁻¹' {g})).Nonempty by …

local prefix:100 "ℓ " ⇒ S.length

/-- For ω a list of generators, π ω is the product in the Coxeter group attached to M
of the word ω. -/
local prefix:100 "π " ⇒ M.toCoxeterSystem.wordProd

/-- For ω a list of generators, φ ω is the product in G of the generators, via
the morphism S.hom :  -/
local notation "φ " x:max ⇒ S.hom (π x)

/-- A word is reduced if its length in the Coxeter group M.Group
(which is the minimal number of generators required to express it)
is equal to its length in G. -/
abbrev IsReduced (ω : List B) : Prop := ℓ (S.hom ◁ π ω ) = ω.length

/-- The "exchange property" for a pre-Coxeter system `S`: if a word of generators
`w = s₁⋯sₖ` in `G` is reduced and reduces further when multiplying on the left by a
generator `s`, then there exists `1 ≤ i ≤ k` such that
`sw = s₁⋯ŝᵢ⋯sₖ`. -/
def ExchangeProperty : Prop :=
  ∀ (ω : List B) (_ : S.IsReduced ω) (b : B) (_ : ℓ (φ (b :: ω)) ≤ ω.length),
    ∃ (i : ℕ) (_ : i < ω.length), φ (b :: ω) = φ (ω.eraseIdx i)
```

Following the exact same proof as in [4, Thm. 1.5.1], we can formalize that the group homomorphism `S.hom` of a pre-Coxeter system `S` that satisfies the exchange property is injective. Our formalization follows closely loc. cit. and we refer the reader to our implementation for more details.

The criterion then applies to symmetric groups: we can define a pre-Coxeter system `Fin.preCoxeterSystem n` on the group Σ_{n+1} for the matrix A_n by sending the generator

i to the permutation $(i\ i+1)$[3]. Following again the material from [4, Thm. 1.5.1], we can formalize that `Fin.preCoxeterSystem n` satisfies the exchange property for all n. This involves relating the notion of length for this pre-Coxeter system with the inversion count of a permutation.

Finally, we record a lemma that we will make use of in our applications

Lemma 2.5. *Let M be a Coxeter Matrix on a set I, the group $\overline{M}$ is also presented as a monoid by the Coxeter relations, i.e., the kernel of the monoid homomorphism* $\mathrm{FreeMonoid}(I) \to \overline{M}$ *is the smallest multiplicative congruence on* $\mathrm{FreeMonoid}(I)$ *containing the words* $\left((s_i s_j)^{M_{ij}}\right)_{i,j}$.

The proof of this lemma reduces to the fact that $s_i^{-1} = s_i$ in $\overline{M}$ for every i.

2.2.2 Morphisms of symmetric lists and permutations

We are now ready for the study of morphisms of symmetric lists. One of the first things to notice is that the way we label morphisms forgets their source and target. We abstract the idea of multiplicatively labelling morphisms of a category by elements of a monoid:

Definition 2.6. *Let C be a category and M be a monoid. A* M-weight *on C is a functor from C to the opposite*[4] *of the category* $\mathrm{B}M$ *that has a single object and the monoid M as endomorphisms of that single object.*

This definition provides an interface between morphisms in a category and elements of a monoid in a way that turns composition into multiplication and that sends identities to the unit of the monoid.

The labeling of morphisms that we described in the opening paragraph of this section should be thought of as a weight on the category of symmetric lists valued on a Coxeter group. In fact, since we want to relate the relations that define symmetric lists with the relations that present permutation groups as Coxeter groups, it is more convenient to first define weights at the level of free objects.

Proposition-definition 2.7. *The inductive assignment w_0 on arrows of* $\mathrm{SListQuiv}(C)$ *characterized by the formulas*

$$w_0(\mathrm{sw}_{a,b,l}) = 0$$
$$w_0(x ::_m f) = 1 + w_0(f)$$

[3] In the pull request #35218 to `Mathlib`, independent of our work, Kim Morrison defined that same morphism and showed its surjectivity as well.

[4] In `Mathlib`, compositions of morphisms in a category are reversed compared to the usual way it is written in pen-and-paper mathematics, i.e., it "follows the arrows"; this is why an opposite is taken here.

extends to a weight w_0 on $\mathrm{FreeSListQuiv}(C)$ *with values in the free monoid on* $\mathbb{N}$*. The weight w_0 descends to a weight w on* $\mathrm{SList}(C)$ *with values in the Coxeter group* $\overline{\mathrm{A}_\infty}$ *in a way such that the diagram*

$$\begin{array}{ccc} \mathrm{FreeSListQuiv}(C) & \xrightarrow{w_0} & (\mathrm{BFreeMonoid}(\mathbb{N}))^{\mathrm{op}} \\ \downarrow{\scriptstyle \pi_C} & & \downarrow \\ \mathrm{SList}(C) & \xrightarrow[w]{} & \left(\mathrm{B}\overline{\mathrm{A}_\infty}\right)^{\mathrm{op}} \end{array}$$

commutes.

Informally, the weight w_0 records the indices at which the sequence of swaps corresponding to a morphism in $\mathrm{FreeSListQuiv}(C)$ is happening, and the weight w realizes a morphism of symmetric list as an element of $\overline{A_\infty}$. The latter can also be thought of as a permutation group: the Coxeter group $\overline{\mathrm{A}_\infty}$ admits a group homomorphism to the group of permutations of $\mathbb{N}$ by sending the generator s_k to the permutation $(k\ k+1)$. One can use the identification of $\overline{\mathrm{A}_n}$ with Σ_{n+1} for every n and the fact that $\overline{\mathrm{A}_\infty}$ is a filtered colimit of the groups $\overline{\mathrm{A}_n}$ to show that this homomorphism is injective.

We can prove that, through this weight and the interpretation of elements of $\overline{\mathrm{A}_\infty}$ as permutations, morphisms of symmetric lists permute elements of the source and target lists in the expected way

```
def toPerm : weight (SList C) (Equiv.Perm ℕ) := …

lemma SList.toPerm_app_lt_of_lt {L₁ L₂ : SList C} (f : L₁ ⟶ L₂) (k : ℕ) (hk : k < L₂.length) :
    (toPerm.app f) k < L₁.length := …

theorem SList.getElem_toList_toPerm {L₁ L₂ : SList C} (f : L₁ ⟶ L₂) (i : ℕ) (hi : i < L₂.length) :
    L₂.toList[i] = L₁.toList[toPerm.app f i]'(toPerm_app_lt_of_lt f i hi) := …
```

An induction argument on morphisms of the category $\mathrm{FreeSListQuiv}\,(C)$ further shows that w_0 is faithful (`eq_of_w₀_app_eq`) as a functor, i.e., that morphisms in $\mathrm{FreeSListQuiv}\,(C)$ with equal labels are equal. Now, part of the main theorem takes the following form:

Theorem 2.8. *The functor w is faithful, that is, two morphisms of symmetric lists are equal if and only if they have the same label in* $\overline{\mathrm{A}_\infty}$*.*

In our formalization, this theorem is called `injective_toAinf_app`, which is itself a direct consequence of the declaration `SList.map_eq_of_w₂_eq` in the listing below and which is a version of the theorem once morphisms have been appropriately lifted to `FreeSListQuiv C`.

```
/- Here, w₂ is the A∞-valued weight on FreeSListQuiv C induced by w₀
and (π C) is the quotient functor FreeSListQuiv C ⥤ SList C. -/
theorem SList.map_eq_of_w₂_eq {L L' : FreeSListQuiv C} (f g : L ⟶ L')
    (h : w₂.app f = w₂.app g) :
    (π C).map f = (π C).map g := …
```

We will prove `SList.map_eq_of_w₂_eq`.

Proof. In this proof, we will be using the same notations as in the Lean code listing above: L and L' will be lists, seen as elements of the free category on the quiver generating morphisms of symmetric lists, and f, g will be morphisms in this category. We will let toAinf be the canonical projection FreeMonoid($\mathbb{N}$) $\to \overline{\mathrm{A}_\infty}$. A first slightly technical step is to cast the hypothesis `h` as an equality in a smaller monoid. We are using here the fact from Lemma 2.5 that Coxeter groups are presented as monoids with the same generators and relations as their presentations as groups. Recall that `w₂.app f` is `toAinf (w₀.app f)`. By induction, one sees that the lengths of the underlying lists of objects are preserved along morphisms in `SListQuiv C` (and hence in `FreeSListQuiv C` and `SList C`) and that the formal word $w_0(f)$ attached to a morphism between symmetric lists of length n cannot involve labels greater than or equal to $n-1$. Thus, in our situation, the words $w_0(f)$ and $w_0(g)$ lift through the injective morphism

$$\iota_{n-1} : \mathrm{FreeMonoid}\,(\mathrm{Fin}(n-1)) \hookrightarrow \mathrm{FreeMonoid}\,(\mathbb{N})$$

induced by the injection $\mathrm{Fin}(n-1) \hookrightarrow \mathbb{N}$, where n is the length of the list underlying the source object L. There is furthermore a commutative diagram

$$\begin{array}{ccc} \mathrm{FreeMonoid}(\mathrm{Fin}(n)) & \xhookrightarrow{\iota_n} & \mathrm{FreeMonoid}(\mathbb{N}) \\ {\scriptstyle \mathrm{toA_n}}\downarrow & & \downarrow{\scriptstyle \mathrm{toAinf}} \\ \overline{\mathrm{A}_n} & \longrightarrow & \overline{\mathrm{A}_\infty} \end{array}$$

and the bottom horizontal map is injective, which we can see via the explicit equivalence between $\overline{\mathrm{A}_n}$ and Σ_{n+1}, and the fact that $\overline{\mathrm{A}_\infty}$ is a filtered colimit of the groups $\overline{\mathrm{A}_n}$, compatible with these injections.

Hence, the equality `h` can be refined as an equality of words in the generators of the group $\overline{\mathrm{A}_{n-1}}$. In fact, we can assume that the words in FreeMonoid(Fin$(n-1)$) that lift $w_0(f)$ and $w_0(g)$ are values on f and g of some weight w'_{n-1} on FreeSListQuiv(C) with values in FreeMonoid(Fin$(n-1)$). A weight w'_{n-1} with this requirement can be constructed out of w_0 by extending to the free monoids any retraction of the injection $\mathrm{Fin}(n-1) \hookrightarrow \mathbb{N}$. In our implementation, we use the function $k \mapsto k\ \%(n-1)$ as such a retraction, and the resulting weight is named `w₀Fin`.

Hence, the equality $\mathrm{toAinf}(w_0(f)) = \mathrm{toAinf}(w_0(g))$ is equivalent to the equality

$$\mathrm{toA_{n-1}}(w'_{n-1}(f)) = \mathrm{toA_{n-1}}(w'_{n-1}(g)) \tag{1}$$

and we can use the characterizing property of a quotient monoid: this equality means that $w'_{n-1}(f)$ and $w'_{n-1}(g)$ are words in the free monoid on $\mathrm{Fin}\,(n-1)$ that are related by the minimal congruence on that monoid generated by the relations defining the Coxeter group $\overline{\mathrm{A}_{n-1}}$. In `Mathlib`, the minimal congruence generated by a relation is set up inductively, and we recall its definition to fix the notations

```
inductive ConGen.Rel [Mul M] (r : M → M → Prop) : M → M → Prop
  | of : ∀ x y, r x y → ConGen.Rel r x y
  | refl : ∀ x, ConGen.Rel r x x
  | symm : ∀ {x y}, ConGen.Rel r x y → ConGen.Rel r y x
  | trans : ∀ {x y z}, ConGen.Rel r x y → ConGen.Rel r y z → ConGen.Rel r x z
  | mul : ∀ {w x y z}, ConGen.Rel r w x → ConGen.Rel r y z → ConGen.Rel r (w * y) (x * z)
```

In our formalization, for Coxeter monoids, the relation taken on words is also inductively generated

```
variable {B : Type*} (M : CoxeterMatrix B) in
inductive CoxeterMatrix.monoidRelations : FreeMonoid B → FreeMonoid B → Prop
  | intro (i j : B) : monoidRelations ((.of i * .of j) ^ M i j) 1
```

The proof proceeds by induction on the equality (1) using the recursor characterizing `ConGen.Rel` above, generalizing both lists L and L' in the process so that inductive hypotheses that may appear in some cases can be applied to morphisms with possibly different sources and targets. The cases `ConGen.refl` and `ConGen.symm` pose no technical difficulties. The cases `ConGen.trans` and `ConGen.mul` are slightly more involved, and make clearer why we decided to first reduce to the case of the group $\overline{\mathrm{A}_{n-1}}$: as we are performing an induction on *words* instead of *morphisms* here, the inductive hypotheses in these cases will in fact involve extra words in $\mathrm{FreeMonoid}\,(\mathrm{Fin}\,(n-1))$ that *a priori* might not be attached to any morphism in $\mathrm{FreeSListQuiv}\,(C)$. For instance, the `ConGen.trans` case will provide us with an extra word z related to both $\mathrm{toA}_{n-1}(w'_{n-1}(f))$ and $\mathrm{toA}_{n-1}(w'_{n-1}(g))$, and the inductive hypothesis can only be applied if we can find an actual morphism h to or from L such that $\mathrm{toA}_{n-1}(w'_{n-1}(h)) = z$. Fortunately, this is the case, and we can prove

```
-- Here, `w₀Fin n` is the weight called w'ₙ₋₁ in the text above.
lemma exists_hom_of_weight_eq (i : FreeSListQuiv C) (n : ℕ) (hj : i.length = n + 2)
    (w : FreeMonoid (Fin (n + 1))) :
    ∃ j : FreeSListQuiv C, ∃ f : i ⟶ j, (w₀Fin n).app f = w := …
```

Note that the object j constructed as above is in fact unique as we have observed previously (through `SList.getElem_toList_toPerm`) that the target of a morphism is constrained by its source and the underlying permutation of the morphism. If we were to work in $\overline{\mathrm{A}_\infty}$ instead of $\overline{\mathrm{A}_{n-1}}$, such morphisms cannot exist if the word z involves letters that are greater than or equal to the length of the sources and targets of f and g. The case `ConGen.mul` is similar, as extra words appear in the inductive hypothesis; we refer the reader to our formalization for more details.

Finally, the case `ConGen.of` follows the same pattern for all possible relations: in this case, the words for $\mathrm{toA}_{n-1}(w'_{n-1}(f))$ and $\mathrm{toA}_{n-1}(w'_{n-1}(g))$ will be explicit small words of generators, either of the form 1, s_i^2, $(s_i s_{i+1})^3$ or $(s_i s_j)^2$. In each of these cases, we can explicitly build a morphism (using our constructors) starting from L, with the expected label under w'_{n-1}. Since we already know that at the level of free monoids and $\mathrm{FreeSListQuiv}\,(C)$, the labels determine morphisms, these new morphisms built by hand

must in fact be equal to f and g and we can then observe that these second morphisms are equal as morphisms of symmetric lists via computations. As hinted in the opening paragraph of this subsection, the relation $s_i^2 = 1$ reduces to the relation `swap_swap`, the relation $(s_i s_{i+1})^3 = 1$ is essentially the relation `hexagon` and the relation $(s_i s_j)^2 = 1$ (for $|i - j| > 1$) corresponds to `swap_naturality`. □

Thanks to the fact that $\overline{A}_\infty$ injects into Perm $(\mathbb{N})$ as permutations with finite support, we can fully interpret morphisms in SList (C) as actual permutations.

```
def toEquiv {x y : SList C} (f : x ⟶ y) :
    Fin y.length ≃ Fin x.length where
  toFun j := ⟨toPerm.app f j, toPerm_app_lt_of_lt _ _ j.prop⟩
  invFun j := ⟨(toPerm.app f).symm j, by simpa using toPerm_app_lt_of_lt (inv f) _ j.prop⟩
  left_inv j := by simp
  right_inv j := by simp
```

This function respects composition and inverses.

Motivated by this construction, given a symmetric list L, we will call the *set of indices* of L the set $\{0, \ldots, \text{length}(L) - 1\}$. On the Lean side, we will refer to `Fin L.length` as the *type of indices* of `L`. Thanks to Theorem 2.8, we obtain

```
theorem SList.getElem_toList_toEquiv {x y : SList C} (f : x ⟶ y) (i : Fin y.length) :
     y.toList[i] = x.toList[(toEquiv f) i] := …

theorem SList.hom_eq_iff_toEquiv_eq {x y : SList C} (f g : x ⟶ y) :
    f = g ↔ (toEquiv f) = (toEquiv g) where …
```

and we can further cement the link between permutations (or rather, bijections of sets of indices) and symmetric lists by performing an induction to show

```
theorem SList.exists_lift_equiv {x y : SList C} (φ : Fin y.length ≃ Fin x.length)
    (hφ : ∀ i : Fin y.length, y.toList[i] = x.toList[φ i]) :
    ∃ f : x ⟶ y, toEquiv f = φ := …

def SList.liftEquiv {x y : SList C} (φ : Fin y.length ≃ Fin x.length)
    (hφ : ∀ i : Fin y.length, y.toList[i] = x.toList[φ i]) :
    x ⟶ y :=
  (exists_lift_equiv φ hφ).choose

lemma SList.toEquiv_liftEquiv {x y : SList C} (φ : Fin y.length ≃ Fin x.length)
    {hφ : ∀ i : Fin y.length, y.toList[i] = x.toList[φ i]} :
    toEquiv (liftEquiv φ hφ) = φ :=
  exists_lift_equiv φ hφ ▷.choose_spec
```

This last batch of declarations along with `SList.hom_eq_iff_toEquiv_eq` from the listing before provides in practice all of the API we need for working with symmetric lists. Altogether, this formalizes the following result:

Corollary 2.9. *Let L_1 and L_2 be objects of* $\mathrm{SList}(C)$*. The function that sends a morphism $L_1 \to L_2$ to the corresponding bijection between the associated sets of indices realizes a bijection between morphisms $L_1 \to L_2$ and bijections*

$$\phi : \{0, \ldots, \mathrm{length}(L_2) - 1\} \xrightarrow{\sim} \{0, \ldots, \mathrm{length}(L_1) - 1\}$$

such that $L_1[\phi(i)] = L_2[i]$.

Together, the definitions in this subsection construct a functor

$$\mathrm{I}_C : \mathrm{SList}\,(C) \to \mathrm{Core}\left(\mathrm{Fin}_{/C}\right)$$

by sending a symmetric list L to its set of indices, along with the map $i \mapsto L[i]$, and sending a map f to the inverse of the equivalence between index types induced by `toEquiv`. The results above assert that this functor is fully faithful. Essential surjectivity can be proven easily using that a finite set J is equivalent to $\mathrm{Fin}\,(k)$ where k is the cardinality of J. Hence, the functor I_C is an equivalence of categories, though we cannot say anything about the symmetric monoidal aspect of this equivalence yet.

2.3 From symmetric lists to the coherence theorem

The remaining step to prove a statement somewhat resembling Theorem 1.3 is to link symmetric lists to the free monoidal category on a set. This is part of the content of S. Piceghello's thesis [17], which he formalized in the Rocq proof assistant within the HoTT library.

Theorem 2.10. *[17, Cor. 4.49] Let C be a set, the category* $\mathrm{SList}\,(C)$ *of symmetric lists on C is equivalent to the free symmetric monoidal category on C.*

Through this equivalence, the category $\mathrm{SList}\,(C)$ admits an essentially unique symmetric monoidal structure in such a way that the equivalence of Theorem 2.10 becomes a symmetric monoidal equivalence of categories. In fact, defining this monoidal structure is part of proving the equivalence. We essentially translate the relevant constructions from [17] in our context.

Free symmetric monoidal categories The free monoidal category on a set was already formalized in `Mathlib` as a category presented by generators and relations. In this presentation, objects are inductively generated by a unit object, objects coming from the base type, and a binary tensor product operation. Morphisms are then inductively generated by formal associators, unitors, and tensor products of morphisms. Finally, relations are added to ensure that associators and unitors are natural, that tensor products are functorial, and that the triangle and pentagon identities hold. With such a definition, the fact that the resulting object is a free monoidal category on the input type is essentially tautological.

In our formalization, we merely adapted this definition to the symmetric monoidal case by adding the relevant extra generators and relations: the type of objects remains constructed the same way, generators are added for the braiding isomorphisms, with extra relations for the hexagon and symmetry identities.

As in the non-symmetric case, it is straightforward to show that this construction has the expected universal property: every function from the base type to any symmetric monoidal category lifts to a symmetric monoidal functor out of the free symmetric monoidal category we constructed. Similarly, monoidal natural transformations of symmetric monoidal functors (or, more generally, from a symmetric monoidal functor to a lax braided functor[5]) out of this free symmetric monoidal category can be defined from their components at objects coming from the base category, and are uniquely determined by these components.

The symmetric monoidal structure on symmetric lists One of the main steps for proving Theorem 2.10 consists of defining a monoidal structure on the category of symmetric lists. To define the tensor product bifunctor, one lifts the operation of appending lists to a bifunctorial operation on $\mathrm{SList}(C)$. This is done using Lemma 2.3 and implemented through the `RecursiveFunctorData` structure that we described in subsection 2.1, which allows one to define functors out of symmetric lists in an inductive way.

```
abbrev SList.appendRecData : RecursiveFunctorData C (SList C ⥤ SList C) where
  nilObj := 𝟭 (SList C)
  consF c := Functor.whiskeringRight _ _ _ ▷.obj (c>~) -- c>~ is the functor c ::~ -
  swapIso x y := NatIso.ofComponents
    (fun F ↦
      (Functor.associator ..) ≪≫
        (Functor.isoWhiskerLeft F ◁ swapNatIso _ _) ≪≫
        (Functor.associator ..).symm)
    (fun {x y} f ↦ by ext; simp [swap_natural])
  ⋯

def SList.appendFunctor : SList C ⥤ (SList C ⥤ SList C) := appendRecData.functor
```

Notice that this definition is the direct categorification of the usual inductive definition of appending two lists, which shows that `appendFunctor` is the usual operation on the underlying lists.

Extending this operation to a (not yet symmetric) monoidal structure on $\mathrm{SList}(C)$ is easily done, as this operation is strictly associative and unital[6]. The definition of the

[5] In fact, we only need the source functor to be oplax braided, i.e., an oplax monoidal functor compatible with the braiding, but `Mathlib` currently lacks type-classes for braided structure on oplax monoidal functors.

[6] This is one of the very few places where we make use of the fact that the isomorphisms v_{nil} and v_{cons} that appear in Lemma 2.3 are underlain by equalities. This is used as a shortcut to skip some computations, and is not essential in any way: we could also follow the same computations as in [17] to avoid making use of any equality of objects.

braiding is slightly more involved, and we follow the exact same constructions as in [17, Def. 4.25]. One first inductively defines an isomorphism

$$Q_{x,l_1,l_2} : (x :: l_1) \otimes l_2 \simeq l_1 \otimes (x :: l_2)\,,$$

that is natural in l_1, by using the induction principle on l_1 for natural transformations out of the category of symmetric lists (Lemma 2.3). The base case is the isomorphism

$$\begin{aligned} (x :: [\,]) \otimes l_2 &\rightarrowtail x :: ([\,] \otimes l_2) && \text{characterization of } \otimes \\ &\rightarrowtail x :: l_2 && \text{right unitor.} \end{aligned}$$

Then, given some isomorphism $(x :: l_1) \otimes l_2 \rightarrowtail l_1 \otimes (x :: l_2)$ and $y \in C$, one constructs $Q_{x,y::l_1,l_2}$ as the composition

$$\begin{aligned} (x :: y :: l_1) \otimes l_2 &\rightarrowtail x :: ((y :: l_1) \otimes l_2) && \text{characterization of } \otimes \\ &\rightarrowtail x :: y :: (l_1 \otimes l_2) && \text{characterization of } \otimes \\ &\rightarrowtail y :: x :: (l_1 \otimes l_2) && \text{using } \mathrm{sw}_{x,y,l_1 \otimes l_2} \\ &\rightarrowtail y :: ((x :: l_1) \otimes l_2) && \text{characterization of } \otimes \\ &\rightarrowtail y :: (l_1 \otimes (x :: l_2)) && \text{induction} \\ &\rightarrowtail (y :: l_1) \otimes (x :: l_2) && \text{characterization of } \otimes. \end{aligned}$$

One then inductively builds the full braiding out of Q. The base case is again easily provided, and assuming that some isomorphism $l_1 \otimes l_2 \rightarrowtail l_2 \otimes l_1$ is constructed and $x \in C$, one builds an isomorphism $(x :: l_1) \otimes l_2 \rightarrowtail l_2 \otimes (x :: l_1)$ as the composition

$$\begin{aligned} (x :: l_1) \otimes l_2 &\rightarrowtail x :: (l_1 \otimes l_2) && \text{characterization of } \otimes \\ &\rightarrowtail x :: (l_2 \otimes l_1) && \text{induction} \\ &\rightarrowtail (x :: l_2) \otimes l_1 && \text{characterization of } \otimes \\ &\rightarrowtail l_2 \otimes (x :: l_1) && \text{via } Q_{x,l_2,l_1}. \end{aligned}$$

The inclusion functor and the equivalence By the universal property of the free symmetric monoidal category on a type, the symmetric monoidal structure from the previous paragraph induces a symmetric monoidal functor N from the free symmetric monoidal category on C to symmetric lists, characterized up to a unique monoidal isomorphism by the fact that $N(c)$ is isomorphic to the singleton symmetric list $[c]$.[7]

[7] In fact, our implementation of free symmetric monoidal categories can ensure that the isomorphisms $N(c) \rightarrowtail [c]$ are (definitional) equalities. But for the purpose of stating things, we prefer to avoid as much

In the other direction, one defines a functor J from symmetric lists to the free symmetric monoidal category on C using the induction principle for functors out of symmetric lists, i.e., Lemma 2.3 and the structure `RecursiveFunctorData` that we introduced earlier. The base case sends the empty list to the unit for the symmetric monoidal structure, and sends the functor $L \mapsto c :: L$ to the functor $x \mapsto c \otimes x$. Through direct computations (and several uses of `SList.recNatTrans`, or rather its natural isomorphism variant `SList.recNatIso`), we can put a monoidal structure on J as well. Compatibility with the braiding is arguably the most computational part, as it ends up relying on the recursive definition of the braiding in symmetric lists in terms of the partial braiding Q mentioned above.

The unit isomorphism is again constructed by induction, leveraging the characterizing properties of the functors J and N: the case of an empty list is easy, and assuming an isomorphism $N(J(L)) \xrightarrow{\sim} L$ is already constructed, its component at a symmetric list of the form $c :: L$ is characterized as the following composite isomorphism

$$\begin{aligned}
N\left(J\left(c :: L\right)\right) &\xrightarrow{\sim} N\left(c \otimes J\left(L\right)\right) && \text{by characterization of } J \\
&\xrightarrow{\sim} N\left(c\right) \otimes N\left(J\left(L\right)\right) && \text{by monoidality of } N \\
&\xrightarrow{\sim} [c] \otimes N\left(J\left(L\right)\right) && \text{by characterization of } N \\
&\xrightarrow{\sim} [c] \otimes L && \text{by induction} \\
&\xrightarrow{\sim} c :: L && \text{by definition.}
\end{aligned}$$

The fact that this construction respects swaps boils down to a direct computation. One can furthermore show that the unit isomorphism constructed this way is monoidal.

The counit isomorphism for the equivalence is easily defined, as it is supposed to be a monoidal natural isomorphism between symmetric monoidal functors out of the free symmetric monoidal category on C: such a natural isomorphism is determined by its components at elements of C, i.e., it suffices to give a family of isomorphisms $(J(N(c)) \xrightarrow{\sim} c)_{c \in C}$, and such isomorphisms are provided by the definitions of J and N. One can then use a small trick to show that the equivalence constructed this way forms an adjoint equivalence:

Lemma 2.11. *Let $F, G : \mathsf{C} \to \mathsf{D}$ be lax monoidal functors between monoidal categories, given monoidal isomorphisms $\eta : F \circ G \simeq \mathrm{Id}$ and $\epsilon : G \circ F \simeq \mathrm{Id}$, so that (F, G, η, ϵ) forms a (not necessarily adjoint) equivalence of categories. The counit of the adjointification (F, G, η, ϵ') of (F, G, η, ϵ) is monoidal.*

as possible a language that is not invariant under equivalences of categories. Phrasing things in terms of isomorphisms does bring an additional subtlety in the fact that the family of isomorphisms $(N(c) \xrightarrow{\sim} [c])_{c \in C}$ should be considered as additional data characterizing N in the category of symmetric monoidal functors and monoidal natural transformations. Such families could *a priori* have non-trivial automorphisms yielding non-trivial monoidal automorphisms of N. In this particular case, there are no such automorphisms, as the objects $[c] \in \mathrm{SList}(C)$ do not have any non-trivial automorphism and hence N is characterized up to a unique monoidal isomorphism by the mere existence of such isomorphisms.

Proof. The adjointification process can be performed in any bicategory, in particular, it can be performed in the bicategory of monoidal categories, lax monoidal functors and monoidal natural transformations. □

In Lean, the proof of the above lemma is essentially the same: being a monoidal natural transformation is stated as a type class `NatTrans.IsMonoidal` and instances of this typeclass are provided for left and right whiskerings of monoidal natural transformations by a lax monoidal functor, as well as for the associators and unitors. The adjointification process only involves such constructions, so the Lean proof that adjointification preserves monoidality is completely automated by the typeclass inference system.

By adjointifying the equivalence formed by J and N, we get an adjoint equivalence in which the counit is a monoidal natural isomorphism, and it suffices to show that this new counit is the same as the one we constructed above. Since the new counit of the adjoint equivalence is monoidal, it is uniquely determined by its components $J(N(c)) \xrightarrow{\sim} c$ at objects $c \in C$. Using merely the fact that N is fully faithful (which does not require any adjointness) and is such that $N(c) \simeq [c]$, we can show that such a family of isomorphisms must be unique, as symmetric lists of the form $[c]$ have no automorphisms.

2.3.1 Symmetric lists and finite types

To guide automation and streamline proofs related to the monoidal structure on symmetric lists, it is useful to define proxy equivalences

```
def SList.Ψ (x y : SList C) : Fin x.length ⊕ Fin y.length ≃ Fin (x ⊗ y).length := …

def SList.Φ (x : C) (l : SList C) : Unit ⊕ Fin l.length ≃ Fin (x ::ₛ l).length := …
```

Under the hood, the equivalence `SList.Ψ` is a cast along the equality

$$\mathrm{length}(x \otimes y) = \mathrm{length}(x) + \mathrm{length}(y)$$

followed by the equivalence (from `Mathlib`) `finSumFinEquiv : Fin m ⊕ Fin n ≃ Fin (m + n)`, and similarly for `SList.Φ`. Abstracting these equivalences via definitions rather than using their underlying nature as casts gives a very systematic way of proving equalities of morphisms out of tensor products of symmetric lists. To prove an equality `f = g` where `f, g : x ⊗ y → _` are morphisms of symmetric lists, one first uses `hom_eq_iff_toEquiv_eq`, then one calls the `ext` tactic to introduce a term `i : Fin (x ⊗ y).length` in context, and then one uses surjectivity of `SList.Ψ` to obtain a term `Fin x.length ⊕ Fin y.length` on which the induction principle for sum types can be used, reducing to proving the two equalities

```
∀ i : Fin x.length, (toEquiv f) (Ψ x y (.inl i)) = (toEquiv g) (Ψ x y (.inl i))
```

and

```
∀ i : Fin y.length, (toEquiv f) (Ψ x y (.inr i)) = (toEquiv g) (Ψ x y (.inr i))
```

This design becomes very efficient at automating proofs involving the monoidal structure on symmetric lists when paired with a systematic characterization of the various basic operations of the monoidal structure, such as tensor products of morphisms, in terms of `toEquiv` and `Ψ`. For instance, we can show key formulas like

```
lemma whiskerRight_apply_right {x y : SList C} (f : x → y) (z : SList C)
    (i : Fin y.length) :
    toEquiv (f ▷ z) (Ψ y z ◁ .inl i) = Ψ _ _ (.inl (toEquiv f i)) := by …
```

and marking lemmas of similar shape with attributes like `@[simp, grind =]` enables automation tactics to use them. All formulas of this kind are proved by induction on symmetric lists, using the inductive definition of the tensor product of symmetric lists and the defining property of `toEquiv` as induced by the permutation of $\mathbb{N}$ corresponding to the label of a morphism, which acts exactly as expected on generating morphisms.

The formula in the code listing above is one of many that, show that through the equivalence `SList.Ψ`, the equivalences between types of indices induced by a tensor product of morphisms act on each factor in the expected way, and in fact proves that through the correspondence between morphisms of symmetric lists and equivalences between their types of indices, the braiding we constructed via Piceghello's method indeed corresponds to swapping the factors of a sum type (i.e., the braiding in the cocartesian monoidal structure on types).

The last remaining part of Theorem 1.3 hence comes from the fact that the functor from symmetric lists on C to the groupoid of finite types over C is symmetric monoidal. As we can bundle the equivalence `SList.Ψ` to an equivalence lying over C, the equivalence `SList.Ψ` itself *is* the monoidal structure isomorphism on the functor that sends a symmetric list to its type of indices. Naturality of this equivalence and its compatibility with associators, unitors, and braidings is exactly the content of the characterizing lemmas on these objects that we mentioned above, and thus the Lean formalization of the fact that the equivalence

$$\mathrm{I}_C : \mathrm{SList}(C) \to \mathrm{Core}(\mathrm{Fin}_{/C})$$

is a symmetric monoidal equivalence of categories can be mainly discharged to automation tactics. This finishes the proof of Theorem 1.3.

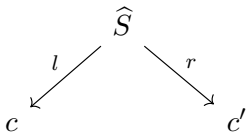
3 Pseudofunctors out of the pith of bicategories of spans

We now turn to the second aspect of our work, which is to use the coherence theorem to produce unbiased symmetric monoidal categories as in Theorem 1.1.

3.1 Bicategories of spans

The statement of Theorem 1.1 involves bicategories of spans. We briefly recall their definitions and explain how we implement them, as they are currently absent from `Mathlib`.

Given two classes of morphisms W_l and W_r in a category C, one can consider spans with left legs in W_l and right legs in W_r: these are by definition diagrams in C



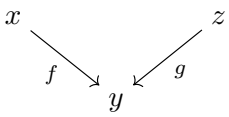
such that $l \in W_l$ and $r \in W_r$. This definition can be formalized as follows:

```
structure Span {C : Type*} [Category* C]
    (Wₗ : MorphismProperty C) (Wᵣ : MorphismProperty C)
    (c c' : C) where
  apex : C
  l : apex → c
  r : apex → c'
  wl : Wₗ l
  wr : Wᵣ r
```

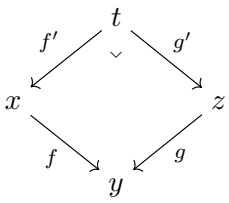
Spans admit a category structure, where a morphism $S \to S'$ is a morphism between the apices of the spans that is compatible with the projections.

We will say the classes of morphisms W_l and W_r are *adequate* if they satisfy the following conditions:

- They both contain identities and are stable under compositions.
- W_l has pullbacks against W_r, i.e., for every diagram

  such that $f \in W_r$ and $g \in W_l$, there exists a pullback of g along f in C.
- W_l and W_r are stable under base change against each other, i.e., if

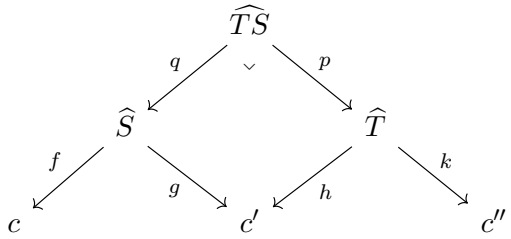
  

  is a pullback square in C where $f \in W_r$ and $g \in W_l$, then $f' \in W_l$ and $g' \in W_r$.

When the classes of morphisms W_l and W_r are adequate, one can define a bicategory of spans:

Definition 3.1. *Let C be a category and let W_l, W_r be adequate classes of morphisms. The* bicategory of spans *of C with respect to W_l and W_r, denoted by* $\mathrm{Span}(C, W_l, W_r)$*, is the bicategory defined by the following data:*

- *Objects are objects of C.*
- *Morphisms from c to c' are spans $c \xleftarrow{f} \widehat{S} \xrightarrow{g} c'$.*
- *2-morphisms are morphisms of spans.*
- *The identity morphism on c is the span $c \xleftarrow{\mathrm{Id}_c} c \xrightarrow{\mathrm{Id}_c} c$.*
- *Composition of two spans $c \xleftarrow{f} \widehat{S} \xrightarrow{g} c'$ and $c' \xleftarrow{h} \widehat{T} \xrightarrow{k} c''$ is defined as the total span in the following diagram*



  where the square is a pullback square.

- *Horizontal composition is induced by the universal property of pullbacks.*
- *Associator and unitor isomorphisms are defined as the canonical morphisms between pullbacks.*

When W_l and W_r consist of all morphisms, adequacy means that C has pullbacks, and we denote the corresponding bicategory of spans by $\mathrm{Span}(C)$ in this case.

The most notable point when implementing this bicategory structure is the quirk that the implementation uses an arbitrary pullback provided by the axiom of choice for the definition of the composition of spans. To make the actual pullback square underlying a composition in Span as irrelevant as possible and to reduce the amount of possible leakage from `Mathlib`'s `Limits.HasLimit` API (which provides the pullbacks via the axiom of choice), we introduce definitions

```
variable {X Y Z : Spans C Wₗ Wᵣ} (S₁ : X ⟶ Y) (S₂ : Y ⟶ Z)
def Spans.πₗ : (S₁ ≫ S₂).apex ⟶ S₁.apex := Limits.pullback.fst _ _
def Spans.πᵣ : (S₁ ≫ S₂).apex ⟶ S₂.apex := Limits.pullback.snd _ _
```

```
def Spans.compPullbackCone : Limits.PullbackCone S₁.r S₂.l :=
  Limits.PullbackCone.mk (πₗ _ _) (πᵣ _ _) (comp_comm _ _)

def Spans.isLimitCompPullbackCone : Limits.IsLimit (compPullbackCone S₁ S₂) := …

def Spans.compLiftApex {c : C} (fₗ : c → S₁.apex) (fᵣ : c → S₂.apex) (hₘ : fₗ ≫ S₁.r = fᵣ ≫ S₂.l) :
    c → (S₁ ≫ S₂).apex :=
  Limits.PullbackCone.IsLimit.lift …
```

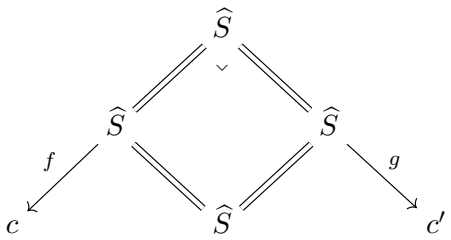
and make these objects the primary API for working with composition of spans. For instance, instead of using `Mathlib`'s existing `CategoryTheory.Limits.pullbackAssoc`, the associators and unitors for the bicategory structure are redefined using `Spans.compLiftApex` and are characterized via `Spans.πₗ` and `Spans.πᵣ`, e.g.,

```
lemma associator_hom_hom_πₗ {W X Y Z : Spans C Wₗ Wᵣ} (S₁ : W → X) (S₂ : X → Y) (S₃ : Y → Z) :
    (α_ S₁ S₂ S₃).hom.hom ≫ πₗ .. = πₗ .. ≫ πₗ .. := …
```

A design like this minimizes the amount of refactoring one would need to make to use a bundled family of chosen pullbacks instead of the ones provided by the axiom of choice. A systematic characterization of the associators and unitors via `Spans.πₗ` and `Spans.πᵣ` also creates an environment that helps guide automation tactics like `simp` and `grind`.

3.2 Building pseudofunctors out of the pith of spans

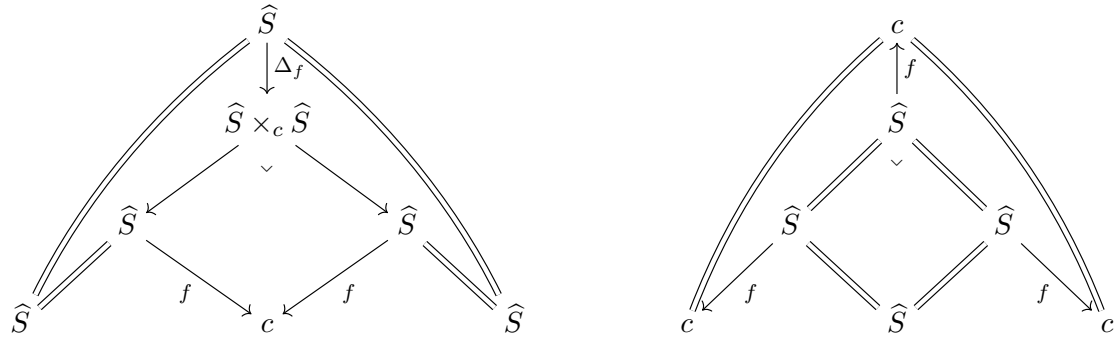
In the following, we will now assume that C is a category with pullbacks, and we will assume that the left and right classes for our spans consist of all morphisms (so that adequacy reduces to the fact that C has pullbacks). Given a span $S : c \xleftarrow{f} \widehat{S} \xrightarrow{g} c'$ in C, the diagram



exhibits S as isomorphic to the composition of two smaller spans $f^* : c \xleftarrow{f} \widehat{S} \xrightarrow{\mathrm{Id}_{\widehat{S}}} \widehat{S}$ and $g_! : \widehat{S} \xleftarrow{\mathrm{Id}_{\widehat{S}}} \widehat{S} \xrightarrow{g} c'$. The constructions $f \mapsto f^*$ and $f \mapsto f_!$ are both pseudofunctorial, the first one being contravariant and the second one being covariant.

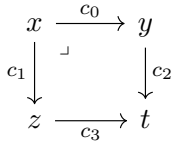
Internally to the bicategory of spans, the 1-morphism $f_!$ is left adjoint to the morphism

f^*: this can be seen by contemplating the diagrams

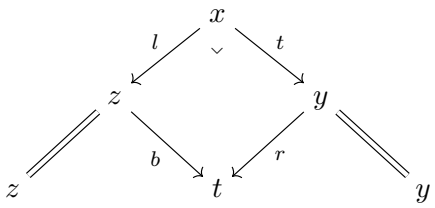


which we can interpret as defining the unit $\mathrm{Id}_{\widehat{S}} \to f^* f_!$ and counit $f_! f^* \to \mathrm{Id}_c$ morphisms of an adjunction. This adjunction is also pseudofunctorial with respect to f, i.e., the construction lifts to a pseudofunctor from C to a suitable bicategory of adjunctions in $\mathrm{Span}(C)$.

Furthermore, given a pullback square



in C, we can look at the diagram



as an isomorphism of 1-morphisms

$$\left(x \xleftarrow{l} x \xrightarrow{t} y\right) \simeq (r^* b_!)$$

and we have seen that the first 1-morphism is also isomorphic to $t_! l^*$. Hence, the pullback square above can be interpreted as an isomorphism

$$r^* b_! \simeq t_! l^*$$

which looks like many "base-change formulas".[8] While these general facts are not strictly needed for our purpose of unbiasing symmetric monoidal categories, we still implement them as a verification of the robustness of our implementation of span bicategories.

[8] In the usual presentation of base change formulas, one direction of the isomorphism is the canonical isomorphism obtained via calculus of adjunctions. In this setting of spans, we can verify that the morphism we constructed is indeed that morphism for the adjunctions and adjointable squares at hand.

Since adjunctions internal to bicategories transport through pseudofunctors, all of these constraints on adjunctions, as well as the base change formula, must hold in the target bicategory. This gives a rather strong constraint on the restriction of the pseudofunctors along the inclusions from base categories, and it is a classical theorem (see e.g., [9, Th. A.2]) that these constraints in fact characterize pseudofunctors out of spans bicategories, i.e., given a bicategory B, a pseudofunctor $\mathrm{Span}(C) \to B$ can be defined out of the data of two pseudofunctors

$$U : C \to B$$
$$V : C^{\mathrm{op}} \to B$$

such that $U(x) = V(x)$ for all x, such that U is left adjoint to V in a suitably pseudofunctorial sense, and such that the base change morphisms induced by pullback squares via this data are isomorphisms.

In the situation of Theorem 1.1, we are interested in defining pseudofunctors out of the pith of the bicategory of spans, i.e., the sub-bicategory of spans one obtains by only keeping morphisms of spans where the morphism between the apices is an isomorphism. We cannot expect the pseudofunctor that appears in Theorem 1.1 to be the restriction of a pseudofunctor defined on the whole bicategory of spans, as in general the functors $f_!$ and f^* that appear in the theorem are not adjoint to each other[9]. We can see that the adjunction $f_! \dashv f^*$ we described earlier does *not* lift to the pith unless the morphism f is an isomorphism. Yet, a pullback square still defines isomorphisms $r^* b_! \simeq t_! l^*$; but these isomorphisms no longer interpret as a general base change morphism obtained via calculus of adjunctions internal to the pith of the bicategory. Hence, to describe a pseudofunctor out of $\mathrm{Pith}(\mathrm{Span}(C))$ through its restrictions along the inclusions of C, one must add the base-change isomorphisms for pullback squares as external extra data. Hence, we introduce the following definition:

Definition 3.2. *Let C be a category and B be a bicategory. A B-valued* Pith-Beck-Chevalley *system on C consists of the following data:*

- *A pseudofunctor $U : C \to B$ and $V : C^{\mathrm{op}} \to B$, such that U and V agree on objects.*
- *For every cartesian square*

$$\begin{array}{ccc} c_0 & \xrightarrow{t} & c_1 \\ {\scriptstyle l}\downarrow & \lrcorner\ S & \downarrow{\scriptstyle r} \\ c_2 & \xrightarrow[b]{} & c_3 \end{array}$$

in C, an isomorphism

$$\mathrm{BC(S)} : V(b)U(r) \xrightarrow{\sim} U(l)V(t)$$

[9] For instance, when f is the unique morphism $\{0,1\} \to \{*\}$, the functor $f_!$ in Theorem 1.1 identifies with the tensor product *bifunctor*, which in general does not preserve colimits as a bifunctor (it sometimes preserves colimits separately in each variable, which is different) and hence cannot be a left adjoint.

in B.

The data must satisfy the following conditions:

- *The isomorphisms* BC *respect horizontal pasting of squares: given a diagram*

$$\begin{array}{ccccc} a & \xrightarrow{t_0} & b & \xrightarrow{t_1} & c \\ {\scriptstyle v_0}\downarrow & \lrcorner\ L & \downarrow{\scriptstyle v_1} & \lrcorner\ R & \downarrow{\scriptstyle v_2} \\ d & \xrightarrow[b_0]{} & e & \xrightarrow[b_1]{} & f \end{array},$$

the diagram

$$\begin{array}{ccccc} & & U(v_0)V(t_1t_0) & & \\ & {\scriptstyle \mathrm{BC}(\mathrm{hcomp}(L,R))}\nearrow & & \searrow{\scriptstyle U(v_0)\cdot V_{\mathrm{comp}}} & \\ V(b_1b_0)U(v_2) & & & & U(v_0)\,(V(t_0)V(t_1)) \\ {\scriptstyle V_{\mathrm{comp}}\cdot U(v_2)}\downarrow & & & & \uparrow{\scriptstyle \alpha} \\ (V(b_0)V(b_1))\,U(v_2) & & & & (U(v_0)V(t_0))\,V(t_1) \\ {\scriptstyle \alpha}\downarrow & & & & \uparrow{\scriptstyle \mathrm{BC}(L)\cdot V(t_1)} \\ V(b_0)\,(V(b_1)U(v_2)) & \xrightarrow[V(b_0)\cdot\mathrm{BC}(R)]{} & V(b_0)\,(U(v_1)V(t_1)) & \xrightarrow[\alpha^{-1}]{} & (V(b_0)U(v_1))\,V(t_1) \end{array}$$

commutes.

- *The isomorphisms* BC *respect vertical pasting of squares: given a diagram*

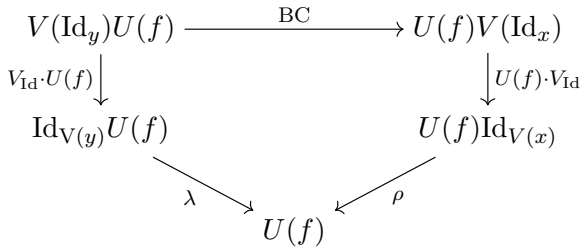
$$\begin{array}{ccc} a & \xrightarrow{h_0} & b \\ {\scriptstyle l_0}\downarrow & \lrcorner\ T & \downarrow{\scriptstyle r_0} \\ c & \xrightarrow[\lrcorner\ h_1]{} & d \\ {\scriptstyle l_1}\downarrow & B & \downarrow{\scriptstyle r_1} \\ e & \xrightarrow[h_2]{} & f \end{array},$$

the diagram

$$\begin{array}{ccccc} & & U(l_1l_0)V(h_0) & & \\ & {\scriptstyle \mathrm{BC}(\mathrm{vcomp}(T,B))}\nearrow & & \searrow{\scriptstyle U_{\mathrm{comp}}\cdot V(h_0)} & \\ V(h_2)U(r_1r_0) & & & & (U(l_1)U(l_0))\,V(h_0) \\ \downarrow{\scriptstyle V(h_2)\cdot U_{\mathrm{comp}}} & & & & \uparrow{\scriptstyle \alpha^{-1}} \\ V(h_2)\,(U(r_1)U(r_0)) & & & & U(l_1)\,(U(l_0)V(h_0)) \\ \downarrow{\scriptstyle \alpha^{-1}} & & & & \uparrow{\scriptstyle U(l_1)\cdot\mathrm{BC}(T)} \\ (V(h_2)U(r_1))\,U(r_0) & \xrightarrow{\mathrm{BC}(B)\cdot U(r_0)} & (U(l_1)V(h_1))\,U(r_0) & \xrightarrow{\alpha} & U(l_1)\,(V(h_1)U(r_0)) \end{array}$$

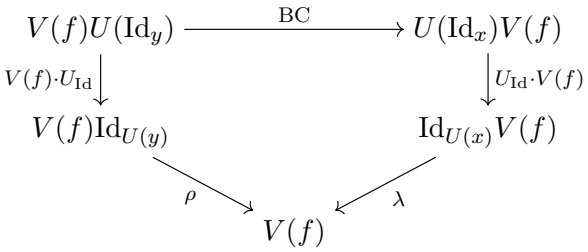
commutes.

- *The isomorphisms* BC *respect horizontal unit squares: given a morphism* $f : x \to y$ *in* C*, the diagram*



commutes.

- *The isomorphisms* BC *respect vertical unit squares: given a morphism* $f : x \to y$ *in* C*, the diagram*



commutes.

We formalize the above definition as follows.

```
structure PseudofunctorCore
    (C : Type u₁) [Category.{v₁} C] (B : Type u₂) [Bicategory.{w₁, v₂} B]
where
  obj : C → B
  u {x y : C} : (x ⟶ y) → (obj x ⟶ obj y)
  v {x y : C} : (x ⟶ y) → (obj y ⟶ obj x)
  uId' {x : C} (f : x ⟶ x) (hf : f = 𝟙 x := by cat_disch) : u f ≅ 𝟙 (obj x)
  vId' {x : C} (f : x ⟶ x) (hf : f = 𝟙 x := by cat_disch) : v f ≅ 𝟙 (obj x)
  uComp' {x y z : C} (f : x ⟶ y) (g : y ⟶ z) (h : x ⟶ z) (hfg : f ≫ g = h := by cat_disch) :
    u h ≅ u f ≫ u g
  vComp' {x y z : C} (f : x ⟶ y) (g : y ⟶ z) (h : x ⟶ z) (hfg : f ≫ g = h := by cat_disch) :
    v h ≅ v g ≫ v f
  -- pseudofunctoriality of u
  u_associator {c₀ c₁ c₂ c₃ : C} (f : c₀ ⟶ c₁) (g : c₁ ⟶ c₂) (h : c₂ ⟶ c₃) : …
  u_left_unitor {c₀ c₁ : C} (f : c₀ ⟶ c₁) : …
  u_right_unitor {c₀ c₁ : C} (f : c₀ ⟶ c₁) : …
  -- pseudofunctoriality of v
```

```
v_associator {c₀ c₁ c₂ c₃ : C} (f : c₁ ⟶ c₀) (g : c₂ ⟶ c₁) (h : c₃ ⟶ c₂) : …
v_left_unitor {c₀ c₁ : C} (f : c₁ ⟶ c₀) : …
v_right_unitor {c₀ c₁ : C} (f : c₁ ⟶ c₀) : …
-- base change isomorphisms and their compatibilities
baseChangeIso {c₀ c₁ c₂ c₃ : C} (t : c₀ ⟶ c₁) (l : c₀ ⟶ c₂) (r : c₁ ⟶ c₃) (b : c₂ ⟶ c₃)
  (S : IsPullback t l r b) :
  u r ≫ v b ≅ v t ≫ u l
baseChangeIso_unit_vert {x y : C} (f : x ⟶ y) :
  (baseChangeIso f (𝟙 x) (𝟙 y) f (IsPullback.of_vert_isIso .mk)).hom =
  (uId' (𝟙 y)).hom ▷ v f ≫ (λ_ _).hom ≫ (ρ_ _).inv ≫ v f ◁ (uId' (𝟙 x)).inv
baseChangeIso_unit_horiz {x y : C} (f : x ⟶ y) :
  (baseChangeIso (𝟙 x) f f (𝟙 y) (IsPullback.of_horiz_isIso .mk)).hom =
  u f ◁ (vId' (𝟙 y)).hom ≫ (ρ_ _).hom  ≫ (λ_ _).inv ≫ (vId' (𝟙 x)).inv ▷ u f
baseChangeIso_comp_horiz : …
baseChangeIso_comp_vert : …
```

We can observe a few technical things about this design:

1. In Remark 2.4, we detailed the woes of working with propositional equalities of objects in categories. This discussion applies equally to propositional equalities of 1-morphisms in bicategories, as they are objects of categories of morphisms. In our case, though, some equalities of 1-morphisms must appear: any 2-morphism in a locally discrete bicategory comes from such an equality. To avoid a proliferation of `eqToHom`-type morphisms in our expressions, we employ a technique also used in `Mathlib` to deal with pseudofunctors out of locally discrete bicategories, which is to give definitional control over the return type of 2-isomorphisms: this is seen, for instance, in the field `uComp'`, where the composition `h` is made a parameter along with a proof that `f ≫ g = h`.

2. In order to phrase the fact that the pseudofunctors U and V agree on objects, we need to unpack all of their fields (`uComp'`, `u_associator`, etc.) in the structure to make it so that the agreement on objects is definitional. Trying to formulate this agreement on objects via propositional equalities of objects in the target bicategory would not be reasonable, as propositional equalities of objects in bicategories further amplifies the problems with dependent types described in Remark 2.4.

   An alternative design would have been to package this part of the structure as the data of two pseudofunctors along with the extra data of a family of equivalences (in the bicategorical sense) between their objects. The added complexity of constantly using these extra equivalences to correct the source or target of the pseudofunctors every time we need to compose 1-morphisms, as well as the extra reassociations of compositions that become needed, would make this other possible approach really hard to work with in practice.

Given $\mathcal{S} := (U, V, \mathrm{BC})$ a Pith-Beck-Chevalley system on a category C, and an isomorphism $\phi : x \to y$ in C, we will denote by $\eta_{\mathcal{S}}(\phi)$ the morphism $U(\phi)V(\phi) \to \mathrm{Id}_y$ obtained as the

composition

$$U(\phi)V(\phi) \xrightarrow{\mathrm{BC}(S)^{-1}} V(\mathrm{Id})U(\mathrm{Id}) \xrightarrow{V_{\mathrm{Id}} \cdot U_{\mathrm{Id}}} \mathrm{Id} \circ \mathrm{Id} \xrightarrow{\lambda} \mathrm{Id}$$

where S is the cartesian square

$$\begin{array}{ccc} x & \xrightarrow{\phi} & y \\ {\scriptstyle\phi}\downarrow & \lrcorner & \downarrow{\scriptstyle \mathrm{Id}\cdot} \\ y & \xrightarrow[\mathrm{Id}]{} & y \end{array}$$

The main result we formalize is

Theorem-definition 3.3. *Let C be a category with pullbacks, B be a bicategory, and let $\mathcal{S} := (U, V, \mathrm{BC})$ be a B-valued Pith-Beck-Chevalley system on C. Then, there is a pseudofunctor*

$$\mathcal{F}_{\mathcal{S}} : \mathrm{Pith}\,(\mathrm{Span}(C)) \to B$$

with the following properties:

- *For every $c \in C$, $\mathcal{F}_{\mathcal{S}}(c) = U(c)$.*
- *Given $c, c' \in C$ and a span $S : c \xleftarrow{f} \widehat{S} \xrightarrow{g} c'$ in C,*

$$\mathcal{F}_{\mathcal{S}}(S) = U(g)V(f).$$

- *Given $c, c' \in C$, spans $S : c \xleftarrow{f} \widehat{S} \xrightarrow{g} c'$ and $S' : c \xleftarrow{f'} \widehat{S} \xrightarrow{g'} c'$, and given a morphism of spans $\phi : \widehat{S} \simeq \widehat{S'}$, the diagram*

$$\begin{array}{ccc} U(g)V(f) & \xrightarrow{\qquad\qquad \mathcal{F}_{\mathcal{S}}(\eta) \qquad\qquad} & U(g')V(f') \\ \| & & \uparrow{\scriptstyle \rho\cdot V(f')} \\ (U(g'\phi))\,(V(f'\phi)) & & (U(g')\mathrm{Id})\,V(f') \\ {\scriptstyle U_{\mathrm{comp}}\cdot V_{\mathrm{comp}}}\downarrow & & \uparrow{\scriptstyle (U(g')\cdot\eta_{\mathcal{S}}(\phi))\cdot V(f')} \\ (U(g')U(\phi))\,(V(\phi)V(f')) & & (U(g')\,(U(\phi)V(\phi)))\,V(f') \\ & {\scriptstyle\alpha^{-1}}\searrow \quad \nearrow{\scriptstyle \alpha\cdot V(f')} & \\ & ((U(g')U(\phi))\,V(\phi))\,V(f') & \end{array}$$

commutes.

- *Given $c, c', c'' \in C$, spans $S : c \xleftarrow{f} \widehat{S} \xrightarrow{g} c'$ and $S' : c' \xleftarrow{h} \widehat{S'} \xrightarrow{k} c''$ in C. Denote by π_1 (resp. π_2) the projection $\widehat{S'S} \to \widehat{S}$ (resp. $\widehat{S'S} \to \widehat{S'}$), the diagram*

$$\begin{array}{ccc}
U(k\pi_2)V(f\pi_1) & \xrightarrow{(\mathcal{F}_{\mathcal{S}})_{\mathrm{comp},S,S'}} & (U(k)V(h))\,(U(g)V(f)) \\
{\scriptstyle U_{\mathrm{comp}}\cdot V_{\mathrm{comp}}}\downarrow & & \uparrow{\scriptstyle \alpha} \\
(U(k)U(\pi_2))\,(V(\pi_1)V(f)) & & \uparrow{\scriptstyle \alpha} \\
{\scriptstyle \alpha^{-1}}\downarrow & & \uparrow{\scriptstyle \alpha} \\
((U(k)U(\pi_2))\,V(\pi_1))\,V(f) & & ((U(k)V(h))\,U(g))\,V(f) \\
{\scriptstyle \alpha\cdot V(f)}\downarrow & & \uparrow{\scriptstyle \alpha^{-1}\cdot V(f)} \\
(U(k)\,(U(\pi_2)V(\pi_1)))\,V(f) & \xrightarrow[(U(k)\cdot \mathrm{BC}(\mathrm{Cs})^{-1})\cdot V(f)]{} & (U(k)\,(V(h)U(g)))\,V(f)
\end{array}$$

commutes, where Cs *is the cartesian square*

$$\begin{array}{ccc}
\widehat{S'S} & \xrightarrow{\pi_1} & \widehat{S} \\
{\scriptstyle \pi_2}\downarrow & \lrcorner & \downarrow{\scriptstyle g} \\
\widehat{S'} & \xrightarrow[g']{} & c'
\end{array}.$$

- *Given $c \in C$, the diagram*

$$\begin{array}{ccc}
U(\mathrm{Id}_c)V(\mathrm{Id}_c) & \xrightarrow{(\mathcal{F}_{\mathcal{S}})_{\mathrm{Id}_c}} & \mathrm{Id} \\
& {\scriptstyle U_{\mathrm{Id}_c}\cdot V_{\mathrm{Id}_c}} \searrow \quad \nearrow {\scriptstyle \lambda} & \\
& \mathrm{Id}_c \circ \mathrm{Id}_c &
\end{array}$$

commutes.

Note that the pseudofunctor that appears in the theorem is necessarily unique, as all of the data parts of a pseudofunctor are fully described here. The fact that these components assemble into a pseudofunctor is purely propositional and carries no extra data.

The formalization of Theorem 3.3 is obtained by bicategorical computations. The computations are rather tedious as the diagrams involve many structural isomorphisms in bicategories (associators, unitors). The most tedious compatibility to check is that of composition with associators in bicategories of spans, which produces goals and expressions with a lot of bicategorical noise. `Mathlib`'s `bicategoricalComp`, combined with `Mathlib`'s `bicategory` tactic is helpful, but not enough so that the proofs could all be fully automated. We believe that the development of specialized 2-dimensional rewriting techniques and tactics could greatly improve the formalization experience in `Mathlib`'s framework for bicategories. We refer the reader to our formalization for the computational details.

Theorem 3.3 is surely well-known, but we could not find it precisely stated in this way in the bicategorical literature (most entries tackle the problem of defining pseudofunctors out of the full bicategory of spans, but not necessarily out of its pith). An $(\infty, 1)$-categorical variant (stated in terms of cocartesian fibrations of $(\infty, 1)$-categories of spans) can be found as [14, Th. 04AE].

4 Packaging symmetric monoidal categories as pseudofunctors

As the shape of the required data to produce a pseudofunctor out of $\mathrm{Pith}\,(\mathrm{Span}\,(\mathrm{Fin}))$ is now clearly identified by Theorem 3.3, it remains to actually produce such data from the data of a symmetric monoidal category C to prove Theorem 1.1.

4.1 The Kleisli bicategory of symmetric lists

Given a finite set J and a symmetric monoidal category C, the product category C^J that appears in Theorem 1.1 should be interpreted as the category of symmetric monoidal functors from the free symmetric monoidal category on J to C, or equivalently, according to Theorem 2.10, as the category of symmetric monoidal functors $\mathrm{SList}(J) \to \mathsf{C}$.

Let J, K be finite sets and let g be a function $J \to K$. When inspecting the formula describing the functor $g_! : \mathsf{C}^J \to \mathsf{C}^K$ that appears in Theorem 1.1 and keeping in mind the interpretation of product categories as symmetric monoidal functors from symmetric lists, it is reasonable to expect that the construction $g \mapsto g_!$ and formulas about these types of functors are instantiations in C of formulas that hold universally in all symmetric monoidal categories, i.e., formulas about symmetric lists.

Hence, to formalize the construction in Theorem 1.1, it would be useful to be able to separate things that happen at the universal level and to abstract the idea that the construction only instantiates the universal formulas in C as the last step.

For an ordinary algebraic theory, universal formulas are encoded by subcategories of free objects. Equivalently, such formulas are encoded by the Kleisli categories for the corresponding monads: the Kleisli category of a monad T on a 1-category C has by definition the same objects as C, and has morphisms from x to y defined as morphisms $x \to T(y)$ in C. For instance, a universal formula for the theory of commutative rings (where $T := \mathbb{Z}[-]$) is an equality of families of polynomials with integer coefficients.

Symmetric monoidal categories do not fit within this conceptual framework directly. This is because, at heart, the structure of a symmetric monoidal category is pseudoalgebraic: associativity and unitality only hold *weakly*, i.e., up to specified natural isomorphisms rather than up to equalities. It is much more natural to interpret a symmetric monoidal category as a *pseudoalgebra* for a *pseudomonad*: these notions are the bicategorical analogues of the 1-categorical concepts of monads and algebras. According to this interpretation, the

appropriate object for encoding universal formulas would be a *Kleisli bicategory*, a bicategorical analogue of the usual Kleisli category, for the pseudomonad corresponding to the theory of symmetric monoidal categories.

This interpretation turns out to be fruitful: the construction $I \mapsto \mathrm{SList}(I)$ with its canonical inclusion $I \hookrightarrow \mathrm{SList}(I)$ and with the universal property of symmetric lists that extends any function $J \to \mathrm{SList}(I)$ to a symmetric monoidal functor $\mathrm{SList}(J) \to \mathrm{SList}(I)$, defines a *relative pseudomonad* [8, Def. 3.1] over the inclusion $\mathrm{Set} \to \mathrm{Cat}$[10]. The Kleisli bicategory of a relative pseudomonad is also defined in [8]. As we will not need generalities on relative pseudomonads to state or formalize our results, we refer the reader to loc. cit. for more details on the notion.

The interpretation of symmetric monoidal categories as pseudoalgebras motivates the following definition, which is the specialization of [8, Th. 4.1] to our situation:

Proposition-definition 4.1. *The* Kleisli bicategory of symmetric lists $\mathcal{K}_{\mathrm{SList}}$ *is the bicategory described by the following data:*

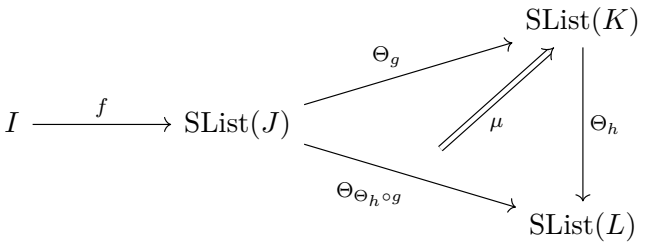
- *Objects are sets.*
- *Morphism categories $I \rightsquigarrow J$ are product categories $(\mathrm{SList}\,(J))^I$.*
- *Identity morphisms $I \rightsquigarrow I$ are given by the inclusions $\iota_I : I \hookrightarrow \mathrm{SList}(I)$ that send an element $i \in I$ to the singleton symmetric list.*
- *The composition of $f : I \rightsquigarrow J$ with $g : J \rightsquigarrow K$ is given by the composition*

$$I \xrightarrow{f} \mathrm{SList}(J) \xrightarrow{\Theta_g} \mathrm{SList}(K)$$

  where Θ_g is the essentially unique extension of g to a symmetric monoidal functor.
- *Given $f, f' : I \rightsquigarrow J$, $g, g' : J \rightsquigarrow K$, $\varphi : f \to f'$ and $\psi : g \to g'$, the horizontal composition of φ and ψ is the horizontal composition $\Theta_\psi \cdot \varphi$ where Θ_ψ is the unique monoidal natural transformation $\Theta_g \to \Theta_{g'}$ induced by ψ.*
- *Given $f : I \rightsquigarrow J$, the right unitor $f \circ \mathrm{Id}_I \xrightarrow{\sim} f$ is the isomorphism $\Theta_f \circ \iota_I \xrightarrow{\sim} f$ that characterizes Θ_f as the extension of f to symmetric lists.*
- *Given $f : I \rightsquigarrow J$, the left unitor $\mathrm{Id}_J \circ f \xrightarrow{\sim} f$ is the isomorphism $\Theta_{\iota_J} \circ f \xrightarrow{\sim} f$ induced by the isomorphism $\Theta_{\iota_J} \simeq \mathrm{Id}_{\mathrm{SList}(J)}$ that is deduced by the fact that both functors are monoidal and are isomorphic to the identity functor when restricted to singleton symmetric lists.*

[10] The fact that we have to use *relative* pseudomonads here is somewhat artificial to our setup: the operation of sending a *category* C to the free symmetric monoidal category on C defines a *bona fide* pseudomonad on Cat, but since we only formalized free symmetric monoidal categories on a set, and only established a link with symmetric lists in that setting, we are constrained to relative pseudomonads over the inclusion $\mathrm{Set} \to \mathrm{Cat}$.

- *Given $f : I \rightsquigarrow J$, $g : J \rightsquigarrow K$, $h : K \rightsquigarrow L$, the associator is described by the diagram*



  where μ is the unique monoidal natural isomorphism whose component at a singleton symmetric list $[j]$ is the isomorphism

$$\Theta_{\Theta_h \circ g}([j]) \xrightarrow{\sim} \Theta_h(g(j)) \xrightarrow{\sim} \Theta_h(\Theta_g([j])).$$

All the proofs required to show that the data of Proposition 4.1 indeed defines a bicategory structure follow the exact same pattern: the required equalities are equalities of families of morphisms of symmetric lists which are components of monoidal natural transformations between monoidal functors out of some category of symmetric lists. It thus suffices to show that these transformations are equal at components consisting of singleton symmetric lists. In this case, the equalities reduce to equalities involving nothing but characterizing isomorphisms for Θ. The fact that they suitably cancel each other in the diagrams is bookkeeping that Lean can discharge via the `simp` tactic.

In ordinary 1-category theory, it is a classical fact that algebras for a monad T interpret as suitable presheaves on the Kleisli category of the monad [7, Th. 1.2]. This interpretation is obtained by restricting the Yoneda embedding of the category of T-algebras along the canonical fully faithful functor from the Kleisli category to the category of T-algebras.

Interpreting symmetric monoidal categories as pseudoalgebras for a (relative) pseudomonad, it is thus reasonable to expect that symmetric monoidal categories interpret as Cat-valued pseudofunctors out of the opposite of the Kleisli bicategory of Proposition 4.1:

Proposition-definition 4.2. *Let C be a symmetric monoidal category. There is a pseudofunctor*

$$\mathsf{C}_{\mathcal{K}} : (\mathcal{K}_{\mathrm{SList}})^{\mathrm{op}} \to \mathrm{Cat}$$

with the following properties:

- *Given a set J, $\mathsf{C}_{\mathcal{K}}(J)$ is the category C^J.*
- *Given a morphism $f : J \rightsquigarrow K$ in $\mathcal{K}_{\mathrm{SList}}$, the functor $\mathsf{C}_{\mathcal{K}}(f)$ is the composition*

$$\mathsf{C}^K \xrightarrow{\Psi_{K,\mathsf{C}}} \mathsf{C}^{\mathrm{SList}(K)} \xrightarrow{f^*} \mathsf{C}^J$$

  where $\Psi_{K,\mathsf{C}}$ is the functor that extends a function $K \to \mathsf{C}$ into a symmetric monoidal functor $\mathrm{SList}(K) \to \mathsf{C}$.

- *Given morphisms $f, f' : J \rightsquigarrow K$ in $\mathcal{K}_{\mathrm{SList}}$ and a 2-morphism $\varphi : f \to f'$, the action of $\mathsf{C}_{\mathcal{K}}$ on φ is described by the diagram*

$$\mathsf{C}^K \xrightarrow{\Psi_{K,\mathsf{C}}} \mathsf{C}^{\mathrm{SList}(K)} \underset{(f')^*}{\overset{f^*}{\Downarrow \varphi^*}} \mathsf{C}^J \ .$$

- *Given a set J, the unitality isomorphism*

$$\mathsf{C}_{\mathcal{K}}(\mathrm{Id}_J) \xrightarrow{\sim} \mathrm{Id}_{C^J}$$

is such that its component at $x : J \to \mathsf{C}$ is the isomorphism

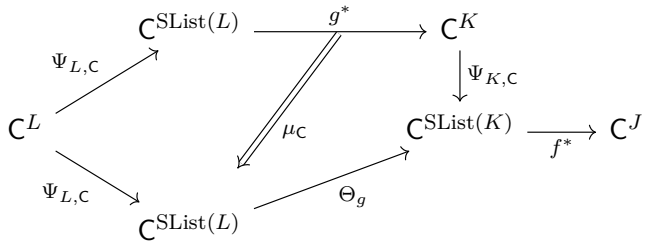
$$\iota_J^* \left(\Psi_{J,\mathsf{C}}(x) \right) \overset{\mathrm{def}}{=} \Theta_x \circ \iota_J \xrightarrow{\sim} x$$

that is part of the defining characterization of Θ_x as the unique symmetric monoidal functor that extends x along ι_J.

- *Given morphisms $f : J \rightsquigarrow K$ and $g : K \rightsquigarrow L$ in $\mathcal{K}_{\mathrm{SList}}$, the composition isomorphism*

$$\mathsf{C}_{\mathcal{K}}(gf) \xrightarrow{\sim} \mathsf{C}_{\mathcal{K}}(g) \circ \mathsf{C}_{\mathcal{K}}(f)$$

is described by the diagram



where μ_{C} is a natural isomorphism such that the component at $x : L \to \mathsf{C}$ is the unique monoidal isomorphism

$$\Psi_{K,\mathsf{C}} \left(g^* \left(\Psi_{L,\mathsf{C}}(x) \right) \right) \overset{\mathrm{def}}{=} \Theta_{\Theta_x \circ g} \xrightarrow{\sim} \Theta_g \circ \Theta_x \overset{\mathrm{def}}{=} \Theta_g \circ \Psi_{L,\mathsf{C}}(x)$$

induced by the symmetric monoidal structures on both sides and by the family of isomorphisms

$$\left(\Theta_{\Theta_x \circ g}([l]) \xrightarrow{\sim} \Theta_x(g(l)) \xrightarrow{\sim} \Theta_x(\Theta_g([l])) \right)_{l \in L}$$

coming from the characterizing property of Θ_h as the essentially unique monoidal functor out of symmetric lists extending a given function h.

Furthermore, for every $f : J \rightsquigarrow K$ in $\mathcal{K}_{\mathrm{SList}}$, the functor $\mathsf{C}_{\mathcal{K}}(f)$ carries a structure of a symmetric monoidal functor, induced by composition of symmetric monoidal structures on $\Psi_{K,\mathsf{C}}$ and on f^*. The action on 2-morphisms as well as the isomorphisms for composition and unitality are monoidal with respect to this structure.

Remark 4.3. One feature of the pseudofunctor described in Proposition 4.2 is that it relies exclusively on the universal property of symmetric lists as a free symmetric monoidal category. We could define a bicategory analogue to $\mathcal{K}_{\mathrm{SList}}$ by replacing symmetric lists with any other model for free symmetric monoidal categories (e.g., the "tautological" one that we used in subsection 2.3 as part of the proof that symmetric lists are free symmetric monoidal categories) and a similar construction would be possible.

This is explained by the fact that the pseudofunctor described in Proposition 4.2 is evidently the restriction of the bicategorical Yoneda embedding for the bicategory SymmMonCat of symmetric monoidal categories, symmetric monoidal functors and monoidal natural transformations along a pseudofunctor $\mathcal{K}_{\mathrm{SList}} \to \mathrm{SymmMonCat}$ that sends J to the category $\mathrm{SList}(J)$. If one is willing to believe that SymmMonCat is biequivalent to the bicategory of pseudoalgebras for the relative pseudomonad (in the sense of [1]) of symmetric lists, then the pseudofunctor in the above definition is exactly the restriction of the (bicategorical) Yoneda embedding along the canonical fully faithful functor from the Kleisli bicategory. While these generalities on pseudomonads may help in conceptually understanding the construction in Proposition 4.2 and see how it is parallel to the 1-categorical phenomenon of [7, Th. 1.2], they are not strictly needed for our development, so we do not formalize them.

The bulk of the formalization of Proposition 4.2 consists of constructing the symmetric monoidal structure on the monoidal extension functor $\Psi_{K,C} : \mathsf{C}^K \to \mathsf{C}^{\mathrm{SList}(K)}$, and proving that all the 2-morphisms that appear (the action on 2-morphisms, as well as the unitality and composition isomorphisms) are monoidal with respect to this structure. The monoidal structure on the extension functor is essentially provided by the fact that in a symmetric monoidal category C, the tensor product bifunctor is itself symmetric monoidal. Once this monoidal structure is obtained, one can construct all the 2-isomorphisms that appear in the proposition by applying the universal property of symmetric lists as a free symmetric monoidal category.

The remaining proof obligations (compatibilities with associators and unitors) are all formalized in the same way as the ones from Proposition 4.1: when unfolded and when working componentwise, every formula that appears is an equality of components of some *monoidal* natural transformation between monoidal functors from a category of symmetric lists to C, and once again, the proofs reduce to bookkeeping (and the characterizing properties of the associators and unitors of the Kleisli bicategory) that automation tactics like `simp` can discharge once we reduce to singleton symmetric lists.

As one would expect from its interpretation as a suitable Yoneda embedding, the encoding of symmetric monoidal categories as pseudofunctors out of the opposite of the Kleisli

bicategory is compatible with morphisms of symmetric monoidal categories:

Lemma 4.4. *Let* C *and* D *be symmetric monoidal categories and let* $F : \mathsf{C} \to \mathsf{D}$ *be a lax braided functor (i.e., a lax monoidal functor that respects the braiding). Postcomposition with* F *extends to a lax natural transformation of pseudofunctors* $\mathsf{C}_{\mathcal{K}} \to \mathsf{D}_{\mathcal{K}}$*. The naturality square for a morphism* $f : J \rightsquigarrow K$ *is given by the diagram,*

$$\begin{array}{ccccc} \mathsf{C}^K & \xrightarrow{\Psi_{\mathsf{C},K}} & \mathsf{C}^{\mathrm{SList}(K)} & \xrightarrow{f^*} & \mathsf{C}^J \\ {\scriptstyle F_*}\downarrow & \swarrow^{\mu} & \downarrow{\scriptstyle F_*} & \swarrow & \downarrow{\scriptstyle F_*} \\ \mathsf{D}^K & \xrightarrow[\Psi_{\mathsf{D},K}]{} & \mathsf{D}^{\mathrm{SList}(K)} & \xrightarrow[f^*]{} & \mathsf{D}^J \end{array}$$

where the component at $X : K \to \mathsf{C}$ *of* μ *is the unique monoidal natural transformation* $\Theta_{F_*(X)} \to F_*(\Theta_X)$ *such that the component at the singleton symmetric list* $[k]$ *is the composite*

$$\Theta_{F_*(X)}([k]) \xrightarrow{\mathrm{def}} F(X(k)) \xrightarrow{F(\mathrm{def})} F(\Theta_X([k]))$$

If F *is (strong) monoidal,* μ *is a natural isomorphism and thus the natural transformation is a strong natural transformation of pseudofunctors.*

We implement the above lemma in our formalization, but unfortunately cannot go further than this and cannot formalize yet that monoidal natural transformations of lax braided monoidal functors induce modifications of the corresponding lax natural transformations: at the time of writing, modifications in `Mathlib` are currently only defined between oplax natural transformations or between strong natural transformations.

As the pseudofunctor $\mathsf{C}_{\mathcal{K}}$ fully encodes the process of instantiating universal formulas of symmetric lists in a given symmetric monoidal category C, we can now give a precise (and hence formalizable) meaning to the idea that the formulas involved in Theorem 1.1 are all universal: the pseudofunctor

$$\mathsf{C}^{\otimes} : \mathrm{Pith}\,(\mathrm{Span}\,(\mathrm{Fin})) \to \mathrm{Cat}$$

we are trying to construct is actually the composition of $\mathsf{C}_{\mathcal{K}}$ with a pseudofunctor

$$\Lambda : \mathrm{Pith}\,(\mathrm{Span}\,(\mathrm{Fin})) \to \mathcal{K}^{\mathrm{op}}_{\mathrm{SList}}.$$

Constructing such a pseudofunctor Λ is hence our new main goal that we must achieve in order to complete the proof of Theorem 1.1.

4.2 Duality for symmetric lists

As an important step towards building a pseudofunctor Λ satisfying our requirements, we need to abstract yet a few more intermediate constructions. As we can see from the

informal description of $\mathsf{C}^{\otimes}$ in Theorem 1.1, the covariant part of the pseudofunctor involves taking tensor products of families indexed by the fibers of a function at a point. It thus makes sense to abstract first the corresponding operation at the level of groupoids of finite sets lying over a given set, before moving on to symmetric lists. To this end, we abstract the following

Proposition-definition 4.5. *Let J be a set and let $j \in J$. The construction that sends a set X equipped with the function $f : X \to J$ to $f^{-1}(\{j\})$ and sends a bijection of finite sets over J to the restriction of the bijection to the fiber at j extends to a symmetric monoidal functor* $\mathrm{fib}_j : \mathrm{Core}\left(\mathrm{Fin}_{/J}\right) \to \mathrm{Core}(\mathrm{Fin})$.

Taken in families, this operation realizes the classical correspondence between sets over a base and families of sets:

Lemma 4.6. *Let J be a finite set, the symmetric monoidal functor*

$$\mathrm{Fib}_J : \mathrm{Core}\left(\mathrm{Fin}_{/J}\right) \to \mathrm{Core}(\mathrm{Fin})^J$$

induced by the family of symmetric monoidal functors $(\mathrm{fib}_j)_{j\in J}$ extends to a symmetric monoidal equivalence of categories, where an inverse is given by the functor that sends a family of sets $(X_j)_{j\in J}$ to their disjoint union, equipped with the sum of the projections.

The formalization of the last two constructions is rather straightforward. In our implementation, we work with `SList Unit` instead of Core(Fin), but this difference is purely cosmetic, as the coherence theorem shows they are equivalent as symmetric monoidal categories, which is a fact we also include in our formalization.

Through the symmetric monoidal equivalence of categories I_C, the symmetric monoidal functor from last lemma unearths an interesting construction about symmetric lists:

Definition 4.7. *Let J and K be finite sets. We define the* duality functor

$$\mathcal{D}_{J,K} : (\mathrm{SList}(K))^J \to (\mathrm{SList}(J))^K$$

as the following composition

$$\begin{aligned}
(\mathrm{SList}(K))^J &\xrightarrow{\sim} \left(\mathrm{Core}\left(\mathrm{Fin}_{/K}\right)\right)^J \\
&\xrightarrow{\sim} \left(\mathrm{Core}\left(\mathrm{Fin}\right)^K\right)^J \\
&\xrightarrow{\sim} \left(\mathrm{Core}\left(\mathrm{Fin}\right)^J\right)^K && \textit{by flipping variables} \\
&\xrightarrow{\sim} \left(\mathrm{Core}\left(\mathrm{Fin}_{/J}\right)\right)^K \\
&\xrightarrow{\sim} (\mathrm{SList}(J))^K .
\end{aligned}$$

This functor is a duality for symmetric lists: the composite equivalence

$$\mathrm{SList}(K)^J \leftrightarrows \left(\mathrm{Core}\,(\mathrm{Fin})^K\right)^J$$

allows us to think of vectors of K-valued symmetric lists as matrices of finite sets (at least heuristically), and the functor $\mathcal{D}_{J,K}$ is essentially the transposition operation in this interpretation. Note that as a composition of symmetric monoidal equivalences of categories, $\mathcal{D}_{J,K}$ is a symmetric monoidal equivalence of categories.

4.3 Symmetric lists and multisets

As we should make no assumptions on the ordering of elements of symmetric lists that go through the inverse of I_C, a natural context to phrase some properties of $\mathcal{D}_{J,K}$ is the language of multisets.

Recall that a *multiset* of elements of K is a list up to permutation, i.e., a finite set with multiplicities. In `Mathlib`, given a type `α`, the type `Multiset α` is defined as the quotient type of the type `List α` by the relation `Perm`, which is defined inductively in the core library of Lean and expresses that a list is a permutation of some other list. The definition of `Perm` makes it very apparent that symmetric lists are a categorification of multisets: more precisely, the set of isomorphism classes of symmetric lists of elements of K is in bijection with multisets of elements of K, and so we may talk about the *underlying multiset* of a symmetric list. Multisets form a commutative monoid where the addition is induced by concatenation of lists, and it is thus not hard to see that the multiset underlying a tensor product of symmetric lists is the sum of the multisets underlying each factor.

We will be using set-like notations for multisets, keeping in mind that multiplicities are allowed (so that $\{i, i\}$ is a multiset distinct from $\{i\}$). Given a multiset M of elements of K and $k \in K$, we will denote by $\mathfrak{c}(k, M)$ the multiplicity of k in M. Given a symmetric list L in some set, we will denote by $||L||$ its underlying multiset.

Continuing the analogy between 1-morphisms in $\mathcal{K}_{\mathrm{SList}}$ and matrices of finite sets, given that isomorphism classes of finite sets are described by natural numbers, we can expect that the underlying multisets of components of 1-morphisms in $\mathcal{K}_{\mathrm{SList}}$ behave like matrices of natural numbers, and indeed, there is a matrix-like formula for the multiset underlying the composition:

Lemma 4.8. *Let J, K and L be sets seen as objects of $\mathcal{K}_{\mathrm{SList}}$, let $f : J \rightsquigarrow K$, $g : K \rightsquigarrow L$ be 1-morphisms in $\mathcal{K}_{\mathrm{SList}}$ and let $j \in J$, then if K is finite,*

$$\|(g \circ f)(j)\| = \sum_{k \in K} \mathfrak{c}\left(k, \|f(j)\|\right) \cdot \|g(k)\|.$$

Proof. By inspecting the definition of composition in the Kleisli bicategory, one sees that it suffices to show that for every function $f : K \to \mathrm{SList}(L)$ and for every symmetric list

$X \in \mathrm{SList}(K)$, the equality

$$||\Theta_f(X)|| = \sum_{k \in K} \mathfrak{c}\,(k, ||X||) \cdot ||f(k)||$$

holds. This formula turns tensor product of symmetric lists into additions of multisets, so it suffices to show it in the case where X is a singleton symmetric list $[k]$. In this case, the formula is tautological as $\Theta_f([k])$ and $f(k)$ are isomorphic by definition of Θ, and so have the same underlying multiset. □

Similarly, the multisets underlying the duality functor $\mathcal{D}_{J,K}$ can be computed with matrix-like formulas, cementing the interpretation of $\mathcal{D}_{J,K}$ as a transposition operator.

Lemma 4.9. *Let J and K be finite sets, and let $(X_j)_{j \in J}$ be a family of symmetric lists of elements of K. Let $k \in K$, then*

$$\|\mathcal{D}_{J,K}(X)(k)\| = \sum_{j \in J} \mathfrak{c}(k, X(j)) \cdot \{j\}.$$

Proof. Since two multisets are equal if and only if they have the same multiplicity at every element, the formula reduces to proving that

$$\mathfrak{c}(j, \|\mathcal{D}_{J,K}(X)(k)\|) = \mathfrak{c}(k, \|X(j)\|)$$

for every $j \in J$ and every $k \in K$. By unfolding the definition of $\mathcal{D}_{J,K}$, it suffices to see that for every finite set S over J, $\mathfrak{c}(j, \mathrm{I}_J^{-1}(\mathrm{Fib}_J^{-1}(S)))$ is the cardinality of the preimage of j in S, which we can see from the definitions. □

Because multisets are precisely isomorphism classes of symmetric lists, proving equalities of underlying multisets is a convenient computational tool to assert the existence of *some* isomorphism between symmetric lists. We have seen already that there are situations where there might exist many such non-equal isomorphisms: the set of automorphisms of the symmetric list that repeats the same element n times is the symmetric group Σ_n, as follows from Corollary 2.9. As we explain and formalize below, the presence of repeated elements in the list is in fact the only possible reason for the existence of multiple isomorphisms between two symmetric lists.

Definition 4.10. *Let J be a set and let L be a symmetric list. We say that L is* linear *if the underlying list of L has no duplicate.*

Equivalently, a symmetric list L is linear if all elements of its underlying multiset have multiplicity at most one. Correspondingly, in our formalization, we introduce

```
class Linear (L : SList C) where
  nodup : L.toList.Nodup
```

This notion recovers another classical form of the coherence theorem for symmetric monoidal categories:

Lemma 4.11. *Let L_1 and L_2 be two symmetric lists with values in a set J.*

1. *If there is a morphism $L_1 \to L_2$, then L_2 is linear if and only if L_1 is linear.*
2. *If either L_1 or L_2 is linear, there is at most one morphism from L_1 to L_2.*

Proof. Since there is a morphism between two symmetric lists if and only if the underlying lists are permutations of each other, the first point follows from the fact that a list has no duplicate if and only if it is a permutation of a list that has no duplicates.

For the second point, assuming that one of the lists is linear, by Corollary 2.9 the set of morphisms from L_1 to L_2 is in bijection with bijections

$$\phi : \{0, \dots, \text{length}(L_2) - 1\} \xrightarrow{\sim} \{0, \dots, \text{length}(L_1) - 1\}$$

such that $L_2[\phi(i)] = L_1[i]$. Since the lists both have no duplicate (by the first point), if such a bijection ϕ exists, the value of $\phi(i)$ must be the unique index of L_1 that has the value $L_2[\phi(i)]$. Hence, there is at most one such bijection. □

In Lean, the conclusion of the second point of the above lemma is naturally expressed by the `Subsingleton` class from the core library of the proof assistant that expresses that a type has at most a single inhabitant. Thus, the formalization of Lemma 4.11 takes the form of a registration of a series of `Subsingleton` instances:

```
lemma linear_iff_of_iso {x y : SList C} (e : x ≅ y) : Linear x ↔ Linear y := by …
instance subsingleton_hom_of_linear_left (L L' : SList C) [h₀ : Linear L] :
    Subsingleton (L → L') where …
instance subsingleton_iso_of_linear_right (L L' : SList C) [Linear L'] :
    Subsingleton (L ≅ L') where …
instance subsingleton_hom_of_linear_right (L L' : SList C) [Linear L'] :
    Subsingleton (L → L') where …
instance subsingleton_iso_of_linear_left (L L' : SList C) [Linear L'] :
    Subsingleton (L' ≅ L) where …
```

Using this infrastructure lets us use the `subsingleton` tactic from `Mathlib` in proofs, which identifies the goal of a proof with an equality between terms of a type with a `Subsingleton` instance and closes said goal if Lean can infer this instance.

4.4 Constructing Λ

In subsection 4.2, we constructed a duality for vectors of symmetric lists, which are the 1-morphisms in $\mathcal{K}_{\text{SList}}$. Spans on a category C with pullbacks also have a canonical duality: the operation that sends a span $S : c \xleftarrow{f} \widehat{S} \xrightarrow{g} c'$ to its transpose ${}^tS : c' \xleftarrow{g} \widehat{S} \xrightarrow{f} c$. Given $f : c \to c'$, this duality exchanges the spans f^* and $f_!$. If one is willing to believe that the pseudofunctor $\Lambda : \text{Pith}\,(\text{Span}\,(\text{Fin})) \to \mathcal{K}^{\text{op}}_{\text{SList}}$ we are trying to construct respects the dualities on its source and target, this leaves only one possible definition:

Theorem-definition 4.12. *There is a unique* $\mathcal{K}_{\mathrm{SList}}^{\mathrm{op}}$*-valued Pith-Beck-Chevalley system on* Fin *with the following properties:*

- *The pseudofunctor* $V : \mathrm{Fin}^{\mathrm{op}} \to \mathcal{K}_{\mathrm{SList}}^{\mathrm{op}}$ *sends a set* J *to itself, and sends a function* $f : J \to K$ *to the composition*

$$J \xrightarrow{f} K \xrightarrow{\iota_K} \mathrm{SList}(K).$$

- *The pseudofunctor* $U : \mathrm{Fin} \to \mathcal{K}_{\mathrm{SList}}^{\mathrm{op}}$ *sends a set* J *to itself, and sends* $f : J \to K$ *to* $\mathcal{D}_{J,K}(V(f))$.

The description in the theorem above might *a priori* appear incomplete, as it does not spell out the pseudofunctoriality data for V and U and does not describe the required base change isomorphisms. In fact, as we will prove below, all of these isomorphisms are uniquely determined by the values prescribed in Theorem 4.12. We will use the notion of linear symmetric lists introduced in 4.3 and Lemma 4.11 to make this apparent.

Proof. Observe that, given a function $f : J \to K$, the composition

$$J \xrightarrow{f} K \xrightarrow{\iota_K} \mathrm{SList}(K)$$

is the function $V(f)$ that sends j to the singleton symmetric list $[f(j)]$, which is linear. This description provides a unique unitality isomorphism: the map $\iota_J : j \mapsto [j]$ is the identity morphism on J in $\mathcal{K}_{\mathrm{SList}}$, so that $V(\mathrm{Id}_J)$ is tautologically isomorphic to the identity, and since the value at every $j \in J$ is a linear symmetric list, this isomorphism must be unique by Lemma 4.11. Similarly, there is a unique composition isomorphism: we can see from the definition of composition in $\mathcal{K}_{\mathrm{SList}}$ that given $f : J \to K$ and $g : K \to L$, the composition $V(g) \circ V(f)$ is a function that sends $j \in J$ to a symmetric list isomorphic to $[g(f(j))]$; uniqueness is again ensured by the fact that the lists are linear.

Set $U(f) := \mathcal{D}_{J,K}(V(f))$ and let $k \in K$. Proposition 4.9 lets us compute that

$$\|U(f)(k)\| = \sum_{j \in J} \mathfrak{c}(k, \|V(f)(j)\|) \cdot \{j\},$$

and since by definition of $V(f)$, $\|V(f)(j)\| = \{f(j)\}$, we obtain that $\mathfrak{c}(k, \|V(f)(j)\|)$ is equal to 1 if $k = f(j)$ and is zero otherwise. Hence,

$$\|U(f)(k)\| = f^{-1}(\{k\}) \tag{2}$$

where the set $f^{-1}(\{k\})$ is seen as a multiset where all of its elements have multiplicity one. In particular, $U(f)(k)$ is linear as a symmetric list. Similarly, given $f : J \to K$ and

$g : K \to L$, we can use Lemma 4.8 to compute that for all $l \in L$,

$$\begin{aligned}\|(U(g) \circ U(f))(l)\| &= \sum_{k \in K} \mathfrak{c}(k, \|U(g)(l)\|) \cdot \|U(f)(k)\| \\ &= \sum_{k \in K} \mathfrak{c}\left(k, g^{-1}\left(\{l\}\right)\right) \cdot f^{-1}\left(\{k\}\right) \\ &= \sum_{k \in g^{-1}(\{l\})} f^{-1}\left(\{k\}\right) \\ &= f^{-1}\left(g^{-1}(\{l\})\right).\end{aligned}$$

Hence, since both sides of the previous equality are pointwise linear symmetric lists with the same underlying multisets, there is a unique isomorphism

$$U(g \circ f) \xrightarrow{\sim} U(f) \circ U(g)$$

by Lemma 4.11. A similar argument shows that there is a unique isomorphism $U(\mathrm{Id}_J) \xrightarrow{\sim} \mathrm{Id}_J$. Because of the linearity of all the symmetric lists involved, compatibility with the associators and unitors for U and V is automatically satisfied, as both sides of each required identity take place in sets with at most one element.

Defining the base change isomorphisms and proving that they respect vertical and horizontal composition make similar use of linearity of symmetric lists: given a cartesian square

$$\begin{array}{ccc} I & \xrightarrow{t} & J \\ {\scriptstyle l}\downarrow & \lrcorner\ S & \downarrow{\scriptstyle r} \\ K & \xrightarrow[b]{} & L \end{array}$$

of finite sets, we can compute using Lemma 4.8, Lemma 4.9 and equation (2) that on the one hand, for $k \in K$,

$$\begin{aligned}\|(U(r) \circ V(b))(k)\| &= \sum_{l \in L} \mathfrak{c}(l, \{b(k)\}) \cdot r^{-1}(\{l\}) \\ &= r^{-1}(\{b(k)\})\end{aligned}$$

which is in particular linear. On the other hand

$$\begin{aligned}\|(V(t) \circ U(l))(k)\| &= \sum_{i \in I} \mathfrak{c}\left(i, l^{-1}\left(\{k\}\right)\right) \cdot \{t(i)\} \\ &= \sum_{i \in l^{-1}(\{k\})} \{t(i)\}\end{aligned}$$

and the multiplicity of $j \in J$ in $\|(V(t) \circ U(l))(k)\|$ is thus the cardinality of the subset of elements $i \in I$ such that $l(i) = k$ and $t(i) = j$. Since the square S is cartesian, this cardinality is at most one and is exactly one when $r(j) = b(k)$, so that the sum $\sum_{i \in l^{-1}(\{k\})} \{t(i)\}$ also describes the set $r^{-1}(\{b(k)\})$.

Hence, since both symmetric lists have equal underlying multisets, there is *some* base change isomorphism $U(r) \circ V(b) \xrightarrow{\sim} V(t) \circ U(l)$, and, as the symmetric lists involved are linear, this base change isomorphism must in fact be unique by Lemma 4.11.

Compatibility of this base change isomorphism with respect to vertical and horizontal pasting of pullback squares is automatic, as well as its compatibility with vertical and horizontal unit squares. Indeed all of these identities can be expressed as a given base change isomorphism being equal to some other composition of morphisms. By the computations above, the sources and targets of all base change isomorphisms are pointwise linear symmetric lists, hence by the second point of Lemma 4.11, there is a unique 2-morphism from the source to the target, so all of these identities must be true. □

The formalization of this theorem follows the argument above. Most of its content is carrying out the relevant multiset computations to prove that all the symmetric lists involved implement the `Linear` typeclass we introduced earlier. The compatibilities between 2-morphisms can then all be discharged by using the already-mentioned `subsingleton` tactic.

Proposition 4.12 finishes the proof of Theorem 1.1, as the Pith-Beck-Chevalley system defined in that proposition defines a pseudofunctor $\Lambda : \mathrm{Pith}\,(\mathrm{Span}\,(\mathrm{Fin})) \to \mathcal{K}^{\mathrm{op}}_{\mathrm{SList}}$ using Theorem 3.3. Finally, we can define the sought-after pseudofunctor unbiasing a symmetric monoidal category, and complete the proof of Theorem 1.1:

```
abbrev Λ : Bicategory.Pith (Spans FintypeCat ⊤ ⊤) ⇒ᵖ Kleisliᵒᵖ := …
-- `Kleisli.pseudoOfSymmMonCat C` is the pseudofunctor called C_𝒦 in the text
universe v u in
def pseudoOfSymmMonCat
    (C : Type u) [Category.{v} C] [MonoidalCategory C] [SymmetricCategory C] :
    Bicategory.Pith (Spans FintypeCat.{0} ⊤ ⊤) ⇒ᵖ Cat.{v, u} :=
  Λ.comp (Kleisli.pseudoOfSymmMonCat C)
```

Note that in the listing above, to ensure good universe properties, we are using the lowest possible universe level for the Kleisli bicategory and for the pith of the bicategory of spans of finite types. In Lean, there is a category of finite types `FintypeCat.{u}` for every universe `u`, which corresponds to the universe in which the underlying types of the objects live in. Similarly, there is in fact a Kleisli bicategory `SList.Kleisli.{u}` of symmetric lists for every universe `u`, and the pseudofunctor Λ is in fact a family of pseudofunctors `Λ.{u} : EffBurnsideFintype.{u} ⇒ᵖ Kleisli.{u}ᵒᵖ` for every universe `u`. If `C : Type u` and `I : Type u'` are types in different universes, the type of functions `I → C` is only guaranteed to be in `Type (max u u')`. Hence, in general, the pseudofunctor $\mathsf{C}_{\mathcal{K}}$ that we defined in Proposition 4.2 might introduce a universe bump if one performs its construction using

`SList.Kleisli.{u}` for a monoidal category that is not `u`-small. By picking `u` to be the lowest universe level `0` for the category of finite types and for the Kleisli bicategory, we ensure that no bump happens and that the original universe levels of the symmetric monoidal category on which we perform the construction are preserved.

5 Conclusion and future work

To summarize, the pseudofunctor from Theorem 1.1 is obtained as a composition described by the diagram

$$\mathrm{Pith}(\mathrm{Span}(\mathrm{Fin})) \xrightarrow{\Lambda} \mathcal{K}^{\mathrm{op}} \xrightarrow{\mathsf{C}_{\mathcal{K}}} \mathrm{Cat}$$

where $\mathcal{K}$ is the Kleisli bicategory for the relative pseudomonad that sends a set to the free symmetric monoidal category on that set. The pseudofunctor $\mathsf{C}_{\mathcal{K}}$ is the restriction to free symmetric monoidal categories of the (bicategorical) Yoneda embedding for the bicategory of symmetric monoidal categories, while the pseudofunctor Λ bundles the link between finite sets and free symmetric monoidal categories. The version of the coherence theorem for symmetric monoidal categories that we implement allows one to give first a better presentation of $\mathcal{K}$ in terms of symmetric lists, and the complete description of morphisms of symmetric lists given in Corollary 2.9 gives the full link between bijections of finite sets and symmetric lists.

As future work, we intend on formalizing some applications of the unbiasing that we constructed in this article. We expect that our work will let us formalize tensor powers of objects in a symmetric monoidal category. More precisely, we should obtain the assignment $x \mapsto x^{\otimes n}$ as a functor from C to Σ_n-equivariant objects in C, by first defining a functor

$$\mathrm{B}\Sigma_n \to \mathrm{End}_{\mathrm{Pith}(\mathrm{Span}(\mathrm{Fin}))}(\{*\}, \{*\})$$

that sends the unique object to the set $\{1, \ldots, n\}$ and then by letting $\mathsf{C}^{\otimes}$ act on it to yield a bifunctor $\mathrm{B}\Sigma_n \to \mathsf{C}^{\otimes}(\{*\}) \to \mathsf{C}^{\otimes}(\{*\})$ on which we can flip the order of the variables to produce the desired bifunctor.

Another possible corollary of our constructions is the ability to unbias commutative monoid objects internal to symmetric monoidal categories: by interpreting a commutative monoid object x internal to a symmetric monoidal category C as a lax symmetric monoidal functor $\{*\} \to \mathsf{C}$, our unbiasing of lax monoidal functors in Proposition 4.4 gives a lax natural transformation from (a pseudofunctor equivalent to) the terminal pseudofunctor to $\mathsf{C}_{\mathcal{K}}$. When whiskerings of lax natural transformations by pseudofunctors are defined in `Mathlib`, this will give the corresponding transformation of pseudofunctors out of $\mathrm{Pith}(\mathrm{Span}(\mathrm{Fin}))$, encoding operations of higher arities for the commutative monoid object x.

Future work also includes formalizing the fact that the functor Λ that we construct in this work is locally an equivalence of categories, providing extra evidence that $\mathrm{Pith}(\mathrm{Span}(\mathrm{Fin}))$

is indeed the "pseudoalgebraic theory" of symmetric monoidal ordinary categories, and fully formalizing the link between symmetric monoidal ordinary categories and Cranch's point of view [6]. This can be seen from the fact that the function/fibration correspondence realizes an equivalence between the core of spans of finite sets from I to J and functions $I \to \mathrm{Core}\left(\mathrm{Fin}_{/J}\right)$, the latter being equivalent to the category of 1-morphisms $I \rightsquigarrow J$ in $\mathcal{K}$ via the coherence theorem. Hence, one needs to formalize a natural isomorphism from the composition of the functor induced by Λ on 1-morphisms and that second equivalence to the function/fibration correspondence. We expect no major roadblock for this part.

The local equivalence mentioned in the previous paragraph will bring us one step closer to what we would consider the most definitive form of unbiasing: biequivalences of bicategories between the bicategory of pseudofunctors (with lax, resp. oplax, resp. strong natural transformations as 1-morphisms) $\mathrm{Pith}(\mathrm{Span}(\mathrm{Fin})) \to \mathrm{Cat}$ that preserve products, and the bicategory of symmetric monoidal categories with lax (resp. oplax, resp. strong) monoidal functors as 1-morphisms. To achieve this, missing basic bicategorical constructions such as biequivalences, products in bicategories, preservation of products by pseudofunctors, local characterization of biequivalences, etc., need to be formalized and added to `Mathlib`.

We expect to contribute the results from this work to `Mathlib` over time, as we fully believe that unbiased symmetric monoidal categories are an essential step towards enabling more complex manipulations of objects in symmetric monoidal categories (such as the aforementioned symmetric tensor powers in general monoidal categories) as well as a necessity to bridge ordinary category theory with higher category theory once the latter develops in `Mathlib`. Similarly, we expect to contribute the missing bicategorical infrastructure mentioned in the previous paragraph to `Mathlib`.

References

[1] Nathanael Arkor, Philip Saville, and Andrew Slattery. *Bicategories of algebras for relative pseudomonads*. 2025. arXiv: 2501.12510 [math.CT]. URL: https://arxiv.org/abs/2501.12510.

[2] Tom Bachmann and Marc Hoyois. "Norms in motivic homotopy theory". In: *Astérisque* 425 (Sept. 2021). ISSN: 2492-5926. DOI: 10.24033/ast.1147. URL: http://dx.doi.org/10.24033/ast.1147.

[3] Ilya Beylin and Peter Dybjer. "Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids". In: *Types for Proofs and Programs*. Ed. by Stefano Berardi and Mario Coppo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 47–61. ISBN: 978-3-540-70722-6.

[4] Anders Björner and Francesco Brenti. *Combinatorics of Coxeter groups*. Vol. 231. Graduate Texts in Mathematics. Springer, New York, 2005, pp. xiv+363. ISBN: 978-3540-442387; 3-540-44238-3.

[5] Francis Borceux. *Handbook of Categorical Algebra*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994.

[6] James Cranch. "Algebraic Theories and $(\infty, 1)$-Categories". In: (2009). arXiv: 1011.3243 [math.AT]. URL: https://arxiv.org/abs/1011.3243.

[7] B. J. Day. "Linear monads". In: *Bulletin of the Australian Mathematical Society* 17.2 (1977), pp. 177–192. DOI: 10.1017/S0004972700010418.

[8] M. Fiore, N. Gambino, M. Hyland, and G. Winskel. "Relative pseudomonads, Kleisli bicategories, and substitution monoidal structures". In: *Selecta Mathematica* 24.3 (Nov. 2017), pp. 2791–2830. ISSN: 1420-9020. DOI: 10.1007/s00029-017-0361-3. URL: http://dx.doi.org/10.1007/s00029-017-0361-3.

[9] Claudio Hermida. "Representable Multicategories". In: *Advances in Mathematics* 151.2 (2000), pp. 164–225. ISSN: 0001-8708. DOI: https://doi.org/10.1006/aima.1999.1877. URL: https://www.sciencedirect.com/science/article/pii/S0001870899918777.

[10] Alexander E. Hoffnung. "Spans in 2-Categories: A monoidal tricategory". In: (2013). arXiv: 1112.0560 [math.CT]. URL: https://arxiv.org/abs/1112.0560.

[11] J.F. Jardine. "Supercoherence". In: *Journal of Pure and Applied Algebra* 75.2 (1991), pp. 103–194. ISSN: 0022-4049. DOI: https://doi.org/10.1016/0022-4049(91)90122-I. URL: https://www.sciencedirect.com/science/article/pii/002240499190122I.

[12] leanprover-community. *mathlib4*. https://github.com/leanprover-community/mathlib4. Version v4.28.0-rc1. 2026.

[13] Jacob Lurie. *Higher Algebra*. URL: https://www.math.ias.edu/~lurie/papers/HA.pdf.

[14] Jacob Lurie. *Kerodon*. URL: https://kerodon.net/.

[15] Saunders Maclane. "Natural Associativity and Commutativity". In: *Rice Institute Pamphlet - Rice University Studies* 49 (1963), pp. 28–46.

[16] Ieke Moerdijk and Ittay Weiss. "Dendroidal sets". In: *Algebraic & Geometric Topology* 7.3 (Nov. 2007), pp. 1441–1470. ISSN: 1472-2747. DOI: 10.2140/agt.2007.7.1441. URL: http://dx.doi.org/10.2140/agt.2007.7.1441.

[17] Stefano Piceghello. "Coherence for Monoidal and Symmetric Monoidal Groupoids in Homotopy Type Theory". PhD thesis. Universitetet i Bergen, 2021. URL: https://hdl.handle.net/11250/2830640.

[18] Christopher J. Schommer-Pries. *The Classification of Two-Dimensional Extended Topological Field Theories*. 2014. arXiv: 1112.1000 [math.AT]. URL: https://arxiv.org/abs/1112.1000.

[19] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: ACM, Jan. 2020. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.