Formally verified formalization of index notation in Lean 4

Joseph Tooby-Smith
Reykjavik University, Reykjavik, Iceland

November 7, 2024

Abstract

The physics community relies on index notation to effectively manipulate types of tensors. This paper introduces the first formally verified implementation of index notation in the interactive theorem prover Lean 4. By integrating index notation into Lean, we bridge the gap between traditional physics notation and formal verification tools, making it more accessible for physicists to write and prove results within Lean. We also open up a new avenue through which AI tools can be used to prove results related to tensors in physics. Behind-the-scenes our implementation leverages a novel application of category theory.

1. INTRODUCTION

In previous work [1], the author initiated the digitalization (or formalization) of high energy physics results using the interactive theorem prover Lean 4 [2] in a project called HepLean. Lean is a programming language with syntax resembling traditional pen-and-paper mathematics. Users can write definitions, theorems, and proofs in Lean, which are then automatically checked for correctness using dependent-type theory. The HepLean project is driven by four primary motivations: (1) to facilitate easier look-up of results through a linear storage of information; (2) to support the creation and proof of new results using automated tactics and AI tools; (3) to facilitate checking of result in high energy physics for correctness; and (4) to introduce new pedagogical methods for high-energy physics and computer science.

HepLean is part of a broader movement of projects to formalize parts, or all, of mathematics and science. The largest of these projects is Mathlib [3], which aims to formalize mathematics. Indeed, HepLean uses many results from Mathlib, and has Mathlib as an import. Other projects in mathematics include the ongoing effort led by Kevin Buzzard to formalize the proof of Fermat's Last Theorem into Lean [4]. In the realm of the sciences, the paper [5] looks at absorption theory, thermodynamics, and kinematics in Lean, whilst the package SciLean [6], is a push in the direction of scientific computing within Lean.

Physicists rely heavily on specialized notation to express mathematical concepts succinctly. Among these index notation is particularly prevalent, as it provides a compact and readable way to represent specific types of tensors and operations between them. Such tensors form a backbone of modern physics.

Having a way to use index notation in Lean is crucial for the digitalisation of results from high energy physics. In addition to making results from high energy physics easier to write and prove in Lean, it will make the syntax more familiar to high energy physicists. However, there are challenges in implementing index notation in Lean, namely, the need for a formal and rigorous implementation that is also easy and nice to use. Such an implementation can now be found as part of HepLean:

https://heplean.github.io/HepLean/

$$\begin{tikzcd} \text{Syntax} \arrow[r, dashed, "\text{Elab}"] & \boxed{\text{Tensor Tree}} \arrow[r, "\text{Cat theory}"] & \boxed{\text{Tensor}} \end{tikzcd}$$

Figure 1: Overview of the implementation of index notation in Lean. The solid lines represent formally verified parts of the implementation.

and is the subject of this paper. We hope that the implementation presented here will not only enhance usability of Lean but also promotes the adoption of formal methods in the physics community.

To give an example of the up-shot of our implementation, the result regarding Pauli matrices that $\sigma^{\nu\alpha\dot{\beta}}\sigma_{\nu}^{\alpha'\dot{\beta}'} = 2\varepsilon^{\alpha\alpha'}\varepsilon^{\beta\beta'}$ is written in Lean as follows

```
{pauliCo | ν α β ⊗ pauliContr | ν α' β' = 2 •ₜ εL | α α' ⊗ εR | β β'}ᵀ
```

Lean will correctly interpret this result as a tensor expression with the correct contraction of indices and permutation of indices between each side of the expression. Our implementation can handle different species of tensors, for example real Lorentz tensors, complex Lorentz tensors and Einstein tenors (although as of writing the most involved, complex Lorentz tensors, have been implemented).

Previous implementations of index notation have been made in programming languages like Haskell [7]. However, the programs they appear in do not provide the formal verification capabilities inherent in Lean. The formal verification requirement of Lean introduces unique challenges in implementing index notation, necessitating (what we believe is) a novel solution.

In Section 2 of this paper, we will discuss the details of our implementation. In Section 3 we will give two examples of definitions, theorems and proofs in Lean using index notation to give the reader an idea of how our implementation works in practice. The first of these examples involves a lemma regarding the contraction of indices of symmetric and antisymmetric tensors. The second involves examples related to the Pauli matrices and bispinors. We finish this paper in Section 4 by discussing potential future work related to this project.

NOTATION

In this paper, we will follow Lean's notation for types and terms. For example if `C` is a type (which can be thought of as similar to a set), then `c : C` is an element of the type `C`. In addition, instead of having two streams of notation for mathematical objects, one from the Lean code and one in LaTeX, we will use Lean code throughout represent mathematical objects. Throughout Section 2, we will assume a basic knowledge of the theory of symmetric monoidal categories.

2. IMPLEMENTATION OF INDEX NOTATION INTO LEAN 4

Our implementation of index notation can be thought of as three different representations of tensor expressions and maps between them. This is illustrated in Figure 1.

The first representation is *syntax*. This can (roughly) be thought of as the informal string that represents the tensor expression. It is what the user interacts with when writing results in Lean, and what appears in raw Lean files. An example of syntax is given in the the code snippet we gave above for Pauli matrices.[1]

[1] In practice there is a representation before syntax, which is as a sequence of tokens in file which is parsed by Lean into a more structured syntax. For simplicity, we think of these representations as the same here.

The second representation of a tensor expression is a *tensor tree*. This representation is mathematically formal, but easy to use and manipulate. It is a structured tree which has different types of nodes for each of the main operations that one can perform on tensors.

The process of going from syntax to a tensor tree is done via an elaborator which follows a number of (informally defined) rules.

The third and final representation is a bona-fide *tensor*. This representation is the mathematical object that we are actually interested in. However, in going to this representation we lose the structure of the tensor expression itself, making it difficult to work with.

The process of going from a tensor tree to a tensor involves properties from a symmetric monoidal category of representations.

Throughout our discussion we will need the notion of a 'tensor species'. Thus, we first give formal definition of this, before discussing each of the above representations and the processes between them in more detail.

2.1. TENSOR SPECIES

A tensor species is a novel formalization of the data needed to define e.g., complex Lorentz tensors, real Lorentz tensors or Einstein tensors. The word 'species' is a nod to 'graphical species' defined in [8, 9], from which our construction is inspired.

We will start by giving the complete definition of a tensor species in Lean, and then dissect this definition, discussing each of the components in turn.

In Lean a tensor species is defined as follows:


```
/-- The structure of a type of tensors e.g. Lorentz tensors, Einstein tensors,
complex Lorentz tensors. -/
structure TensorSpecies where
  /-- The commutative ring  over which we want to consider the tensors to live in,
    usually ‘ℝ‘ or ‘ℂ‘. -/
  k : Type
  /-- An instance of ‘k‘ as a commutative ring. -/
  k_commRing : CommRing k
  /-- The symmetry group acting on these tensor e.g. the Lorentz group or SL(2,ℂ).
    -/
  G : Type
  /-- An instance of ‘G‘ as a group. -/
  G_group : Group G
  /-- The colors of indices e.g. up or down. -/
  C : Type
  /-- A functor from ‘C‘ to ‘Rep k G‘ giving our building block representations.
    Equivalently a function ‘C → Re k G‘. -/
  FD : Discrete C ⥤ Rep k G
  /-- A specification of the dimension of each color in C. This will be used for
    explicit
    evaluation of tensors. -/
  repDim : C → ℕ
  /-- repDim is not zero for any color. This allows casting of ‘ℕ‘ to ‘Fin
    (S.repDim c)‘. -/
  repDim_neZero (c : C) : NeZero (repDim c)
```

```
/-- A basis for each Module, determined by the evaluation map. -/
basis : (c : C) → Basis (Fin (repDim c)) k (FD.obj (Discrete.mk c)).V
/-- A map from ‘C‘ to ‘C‘. An involution. -/
τ : C → C
/-- The condition that ‘τ‘ is an involution. -/
τ_involution : Function.Involutive τ
/-- The natural transformation describing contraction. -/
contr : OverColor.Discrete.pairτ FD τ ⟶ 𝟙_ (Discrete C ⥤ Rep k G)
/-- Contraction is symmetric with respect to duals. -/
contr_tmul_symm (c : C) (x : FD.obj (Discrete.mk c))
    (y : FD.obj (Discrete.mk (τ c))) :
  (contr.app (Discrete.mk c)).hom (x ⊗ₜ[k] y) = (contr.app (Discrete.mk (τ
  c))).hom
  (y ⊗ₜ (FD.map (Discrete.eqToHom (τ_involution c).symm)).hom x)
/-- The natural transformation describing the unit. -/
unit : 𝟙_ (Discrete C ⥤ Rep k G) ⟶ OverColor.Discrete.τPair FD τ
/-- The unit is symmetric. -/
unit_symm (c : C) :
  ((unit.app (Discrete.mk c)).hom (1 : k)) =
  ((FD.obj (Discrete.mk (τ (c)))) ◁
    (FD.map (Discrete.eqToHom (τ_involution c)))).hom
  ((β_ (FD.obj (Discrete.mk (τ (τ c)))) (FD.obj (Discrete.mk (τ (c))))).hom.hom
  ((unit.app (Discrete.mk (τ c))).hom (1 : k)))
/-- Contraction with unit leaves invariant. -/
contr_unit (c : C) (x : FD.obj (Discrete.mk (c))) :
  (λ_ (FD.obj (Discrete.mk (c)))).hom.hom
  (((contr.app (Discrete.mk c)) ▷ (FD.obj (Discrete.mk (c)))).hom
  ((α_ _ _ _ (FD.obj (Discrete.mk (c)))).inv.hom
  (x ⊗ₜ[k] (unit.app (Discrete.mk c)).hom (1 : k)))) = x
/-- The natural transformation describing the metric. -/
metric : 𝟙_ (Discrete C ⥤ Rep k G) ⟶ OverColor.Discrete.pair FD
/-- On contracting metrics we get back the unit. -/
contr_metric (c : C) :
  (β_ (FD.obj (Discrete.mk c)) (FD.obj (Discrete.mk (τ c)))).hom.hom
  (((FD.obj (Discrete.mk c)) ◁ (λ_ (FD.obj (Discrete.mk (τ c)))).hom).hom
  (((FD.obj (Discrete.mk c)) ◁ ((contr.app (Discrete.mk c)) ▷
  (FD.obj (Discrete.mk (τ c))))).hom
  (((FD.obj (Discrete.mk c)) ◁ (α_ (FD.obj (Discrete.mk (c)))
    (FD.obj (Discrete.mk (τ c))) (FD.obj (Discrete.mk (τ c)))).inv).hom
  ((α_ (FD.obj (Discrete.mk (c))) (FD.obj (Discrete.mk (c)))
    (FD.obj (Discrete.mk (τ c)) ⊗ FD.obj (Discrete.mk (τ c)))).hom.hom
  ((metric.app (Discrete.mk c)).hom (1 : k) ⊗ₜ[k]
    (metric.app (Discrete.mk (τ c))).hom (1 : k))))))
  = (unit.app (Discrete.mk c)).hom (1 : k)
```

Let us work through through this definition piece by piece. The first part of the definition of a tensor species is a type `k` which is a commutative ring via `k_commRing`. For a given `S : TensorSpecies`, we retrieve `k` and all of the components of `TensorSpecies` via `S.k` etc. For the tensor species of complex Lorentz tensors, denoted `complexLorentzTensor`, the ring `complexLorentzTensor.k` is the ring of complex numbers.

The second part of the definition of a tensor species is a type `G` which is a group via `G_group`. For complex Lorentz tensors, `complexLorentzTensor.G` is the group $SL(2,\mathbb{C})$.

The next part of the definition is a type `C` which we call the type of colors. For complex Lorentz tensors, `complexLorentzTensor.C` is equal to the type

```
inductive Color
  | upL : Color
  | downL : Color
  | upR : Color
  | downR : Color
  | up : Color
  | down : Color
```

which contains six colors. Colors can be thought of as labels for each of the building block representations making up the tensor species. This is made formal by the next part of the definition of a tensor species, a functor `FD` from the discrete category formed by `C` to the category of representations of `G` over `k`, `Rep k G`. The category `Rep k G` and its properties are defined in Mathlib, along with the necessary category theory we use in this paper. The functor `FD` assigns to each color a representation of `G` over `k`. Note that for `c : C` to apply `FD` we have to write `FD.obj (Discrete.mk c)` in Lean. This is somewhat cumbersome, hence in what follows we will abbreviate this to `FD c`. For complex Lorentz tensors, the functor `complexLorentzTensor.FD` is defined as:

```
FD := Discrete.functor fun c =>
  match c with
  | Color.upL => Fermion.leftHanded
  | Color.downL => Fermion.altLeftHanded
  | Color.upR => Fermion.rightHanded
  | Color.downR => Fermion.altRightHanded
  | Color.up => Lorentz.complexContr
  | Color.down => Lorentz.complexCo
```

The representations appearing here are:

- The representation of left-handed Weyl fermions, denoted in Lean as `Fermion.leftHanded`, and corresponding to the representation of $SL(2,\mathbb{C})$ taking $v \mapsto Mv$ for $M \in SL(2,\mathbb{C})$.
- The representation of 'alternative' left-handed Weyl fermions, denoted in Lean as `Fermion.altLeftHanded`, and corresponding to the representation of $SL(2,\mathbb{C})$ taking $v \mapsto M^{-1T} v$ for $M \in SL(2,\mathbb{C})$.
- The representation of right-handed Weyl fermions, denoted in Lean as `Fermion.rightHanded`, and corresponding to the representation of $SL(2,\mathbb{C})$ taking $v \mapsto M^{\star} v$ for $M \in SL(2,\mathbb{C})$.
- The representation of 'alternative' right-handed Weyl fermions, denoted in Lean as `Fermion.altRightHanded`, and corresponding to the representation of $SL(2,\mathbb{C})$ taking $v \mapsto M^{-1\dagger} v$ for $M \in SL(2,\mathbb{C})$.

- The representation of contravariant Lorentz tensors, denoted in Lean as `Lorentz.complexContr`, and corresponding to the representation of $SL(2,\mathbb{C})$ induced by the homomorphism of $SL(2,\mathbb{C})$ into the Lorentz group and the contravariant action of the Lorentz group on four-vectors.

- The representation of covariant Lorentz tensors, denoted in Lean as `Lorentz.complexCo`, and corresponding to the representation of $SL(2,\mathbb{C})$ induced by the homomorphism of $SL(2,\mathbb{C})$ into the Lorentz group and the covariant action of the Lorentz group on four-vectors.

As an example of how these are defined in Lean, the representation of left-handed Weyl fermions `Fermion.leftHanded` is given by:

```
/-- The vector space ℂ^2 carrying the fundamental representation of SL(2,C).
  In index notation corresponds to a Weyl fermion with indices ψ^a. -/
def leftHanded : Rep ℂ SL(2,ℂ) := Rep.of {
  /- The function from SL(2,ℂ) to endomorphisms of LeftHandedModule
    (which corresponds to the vector space ℂ^2). -/
  toFun := fun M => {
    /- Start of the definition of the linear map. -/
    /- The function underlying the linear map. Defined as the dot product. -/
    toFun := fun (ψ : LeftHandedModule) =>
      LeftHandedModule.toFin2ℂEquiv.symm (M.1 *ᵥ ψ.toFin2ℂ),
    /- Proof that the function is linear with respect to addition. -/
    map_add' := by
      intro ψ ψ'
      simp [mulVec_add]
    /- Proof that the function is linear with respect to scalar multiplication. -/
    map_smul' := by
      intro r ψ
      simp [mulVec_smul]
    /- End of the definition of the linear map. -/}
  /- Proof that (the outer) toFun gives the identity map on the identity of
   SL(2,ℂ). -/
  map_one' := by
    ext i
    simp
  /- Proof that the action of the product of two elements is
    the product of the actions of the elements. -/
  map_mul' := fun M N => by
    simp only [SpecialLinearGroup.coe_mul]
    ext1 x
    simp only [LinearMap.coe_mk, AddHom.coe_mk, LinearMap.mul_apply,
   LinearEquiv.apply_symm_apply,
      mulVec_mulVec]}
```

We have added some explanatory comments to this code, not seen in the actual Lean code, to give the reader an idea of what each part does. Note that the `Fermion.` part of the name of `Fermion.leftHanded` is dropped in this definition, as it is inherited from the Lean namespace in which the definition is made.

The next part of the definition of a tensor species is the map `repDim`, which assigns to each color a natural number corresponding to the dimension of the representation associated to that color. The condition is placed on the representations that they are non-empty, i.e., that the dimension is not equal to zero `repDim_neZero`. The `basis` part of the definition of a tensor species gives a basis indexed by `Fin (repDim c)` (numbers from `0` to `repDim c - 1`) of each representation for each `c : C`. We will use this basis in the definition of evaluation of tensor indices. For complex Lorentz tensors, these are the standard basis for Lorentz vectors and Weyl-fermions.

Next in the definition of a tensor species is the map `τ`, which is an involution via `τ_involution`. This assigns to each color its 'dual' corresponding to the color it can be contracted with. So, for complex Lorentz tensors the map `τ` is given by:

```
τ := fun c =>
  match c with
  | Color.upL => Color.downL
  | Color.downL => Color.upL
  | Color.upR => Color.downR
  | Color.downR => Color.upR
  | Color.up => Color.down
  | Color.down => Color.up
```

The contraction itself is defined in a tensor species by `contr`, which is a natural transformation from the functor `FD _ ⊗ FD (τ _)`, to the constant functor `𝟙_ (Discrete C ⇒ Rep k G)` which takes every object in `C` to the trivial representation `𝟙_ (Rep k G)`. This natural transformation is simply the assignment to each color `c` a linear-map from `FD c ⊗ FD (τ c)` to `k` which is equivariant with respect to the group action. This contraction cannot be defined arbitrarily, but must satisfy the symmetry condition `contr_tmul_symm`, which corresponds to the commutative diagram

$$\begin{tikzcd} \texttt{FD c} \otimes \texttt{FD (}\tau\texttt{ c)} \arrow[r, "\beta\_"] \arrow[d, "\texttt{contr c}"'] & \texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD c} \arrow[d, "\mathbb{1} \otimes \texttt{FD.map (e c)}"] \\ \mathbb{1}\texttt{\_(Rep k G)} & \texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD (}\tau\texttt{ (}\tau\texttt{ c))} \arrow[l, "\texttt{contr (}\tau\texttt{ c)}"] \end{tikzcd} \tag{1}$$

where `β_` is the braiding of the symmetric monoidal category, and `e c` is shorthand for the isomorphism in `Discrete S.C` between `c` and `τ (τ c)`, which exists since `τ` is an involution.

Along with the contraction, the definition of a tensor species includes the `unit`, along with its own symmetry condition `unit_symm`. The unit is a natural transformation from the functor `𝟙_ (Discrete C ⇒ Rep k G)` to `FD (τ _) ⊗ FD _`. This is the assignment to each color `c` an object of `FD (τ c) ⊗ FD c` which is invariant with respect to the group action. The symmetry condition `unit_symm` is represented by the commutative diagram

$$\begin{tikzcd} \mathbb{1}\texttt{\_(Rep k G)} \arrow[r, "\texttt{unit c}"] \arrow[d, "\texttt{unit (}\tau\texttt{ c)}"'] & \texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD c} \\ \texttt{FD (}\tau\texttt{ (}\tau\texttt{ c))} \otimes \texttt{FD (}\tau\texttt{ c)} \arrow[r, "\beta\_"'] & \texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD (}\tau\texttt{ (}\tau\texttt{ c))} \arrow[u, "\mathbb{1} \otimes \texttt{FD.map (e c)}"'] \end{tikzcd} \tag{2}$$

The next condition, `contr_unit`, makes formal the statement that contraction with the unit does nothing. It corresponds to the diagram

$$\begin{tikzcd}
\mathtt{FD\ c} \arrow[r, "(\rho_)^{-1}"] & \mathtt{FD\ c} \otimes \mathbb{1}\mathtt{_(Rep\ k\ G)} \arrow[r, "\mathbb{1} \otimes \mathtt{unit\ c}"] & \mathtt{FD\ c} \otimes \mathtt{(FD\ (}\tau\mathtt{\ c)} \otimes \mathtt{FD\ c)} \arrow[d, "(\alpha_)^{-1}"] \\
\mathbb{1}\mathtt{_(Rep\ k\ G)} \otimes \mathtt{FD\ c} \arrow[u, "\lambda_"] & & \mathtt{(FD\ c} \otimes \mathtt{FD\ (}\tau\mathtt{\ c))} \otimes \mathtt{FD\ c} \arrow[ll, "\mathtt{contr\ c} \otimes \mathbb{1}"']
\end{tikzcd} \tag{3}$$

where $\rho\_$ is the right-unitor, $\lambda\_$ is the left-unitor, and $\alpha\_$ is the associator in the category `Rep k G`.

The final part of the definition of a tensor species is the metric, `metric`, and it's interaction, `contr_metric`, with the contraction and unit. The metric is a natural transformation from the functor $\mathbb{1}$`_ (Discrete C ⇒ Rep k G)` to the functor `FD _ ⊗ FD _`. It thus represents the assignment to each color `c` an object of `FD c ⊗ FD c` which is invariant with respect to the group action. The metric can be used to change an index into a dual index. The condition `contr_metric` corresponds to the diagram

$$\begin{tikzcd}
\mathbb{1}\mathtt{_(Rep\ k\ G)} \otimes \mathbb{1}\mathtt{_(Rep\ k\ G)} \arrow[d, "\mathtt{metric\ c} \otimes \mathtt{metric\ (}\tau\mathtt{\ c)}"'] & \mathbb{1}\mathtt{_(Rep\ k\ G)} \arrow[l, "(\rho_)^{-1}"'] \arrow[r, "\mathtt{unit\ c}"] & \mathtt{FD\ (}\tau\mathtt{\ c)} \otimes \mathtt{FD\ c} \\
\mathtt{(FD\ c} \otimes \mathtt{FD\ c)} \otimes \mathtt{(FD\ (}\tau\mathtt{\ c)} \otimes \mathtt{FD\ (}\tau\mathtt{\ c))} \arrow[d, "\alpha_"] & & \mathtt{FD\ c} \otimes \mathtt{FD\ (}\tau\mathtt{\ c)} \arrow[u, "\beta_"'] \\
\mathtt{FD\ c} \otimes \mathtt{(FD\ c} \otimes \mathtt{(FD\ (}\tau\mathtt{\ c)} \otimes \mathtt{FD\ (}\tau\mathtt{\ c)))} \arrow[dr, "\mathbb{1} \otimes (\alpha_)^{-1}"] & & \mathtt{FD\ c} \otimes \mathtt{(}\mathbb{1}\mathtt{_(Rep\ k\ G)} \otimes \mathtt{FD\ (}\tau\mathtt{\ c))} \arrow[u, "\mathbb{1} \otimes \lambda_"'] \\
& \mathtt{FD\ c} \otimes \mathtt{((FD\ c} \otimes \mathtt{FD\ (}\tau\mathtt{\ c))} \otimes \mathtt{FD\ (}\tau\mathtt{\ c))} \arrow[ur, "\mathbb{1} \otimes \mathtt{(contr\ c} \otimes \mathbb{1}\mathtt{)}"'] &
\end{tikzcd} \tag{4}$$

For complex Lorentz tensors, the contraction is defined through the dot product, e.g., the contraction of ψ^{μ} and ϕ_{μ} is via dot product of the underlying vectors. The metric is defined through the Minkowski metric and the metric tensors e.g. $\epsilon^{\alpha\alpha}$ for Weyl fermions. Lastly,the units are defined through identity matrices.

2.2. TENSORS

Given any type `C`, we define the category `OverColor C` as follows. Objects are functions `f : X → C` for some type `X`. A morphism from `f : X → C` to `g : Y → C` is a bijection φ `: X → Y` such that `f = g ∘` φ. This category is equivalent to the core of the category of types sliced over `C`.

The category `OverColor C` carries a symmetric monoidal structure, which we will denote $\otimes$. The structure such that `f ⊗ g` for objects `f` and `g` is the induced map `X ⊕ Y → C` where $\oplus$ denotes the disjoint union of types. In Lean this is denoted `Sum.elim f g`.

For a given tensor species `S`, the functor `S.FD` can be lifted to a functor to a symmetric monoidal functor from `OverColor S.C` to `Rep k G`. Here the monoidal structure on `Rep k G` is the tensor product over `k`. This functor takes `f : X → S.C` to the tensor product over `k` of all `FD (f x)`, `⨂[k] x, S.FD (f x)`. This construction is general and functorial, allowing us to define the functor

```
def OverColor.lift : (Discrete S.C ⥤ Rep S.k S.G) ⥤ BraidedFunctor (OverColor
    S.C) (Rep S.k S.G) where ...
```

from functors from `Discrete S.C` to `Rep S.k S.G` to symmetric monoidal functors (or braided functors) from `OverColor S.C` to `Rep S.k S.G`.

We denote the lift of `S.FD` by `S.F`, which is defined through

```
def F  (S : TensorSpecies) : BraidedFunctor (OverColor S.C) (Rep S.k S.G) :=
  (OverColor.lift).obj S.FD
```

We can think of an object `f : X → S.C` of `OverColor S.C` as a type of indices `X`, and a specification of what color or representation each index is associated to. For example, for the tensor $\phi^{\mu}{}_{\nu}$ would have `X` as the type of indices which, since there are two of them, is `Fin 2`, and `f` as the function which assigns to each index the color of the index, so `f 0` would be `Color.up`, and `f 1` would be `Color.down`. We can apply the functor `S.F` to `f : X → S.C` in Lean as follows `S.F.obj (OverColor.mk f)`, which we will abbreviate to `S.F f`. This representation, `S.F f`, is the tensor product of each of representations `S.FD (f x)` for `x : X`. In our example this is equivalent to `Lorentz.complexContr ⊕ Lorentz.complexCo`. Vectors of the form `v : S.F f` can be thought of as tensors with indices indexed by `X` of color `C`.

With this in mind, we define a general tensor of a species `S` as a vector in a representation `S.F f` for some `f : OverColor S.C`.

In physics, we typically focus on objects `f : X → S.C` of `OverColor S.C` where `X` is a finite type of the form `Fin n` for some `n : ℕ`. In most of what follows, we will restrict to these objects.

2.3. TENSOR TREES AND THEIR MAP TO TENSORS

Tensor trees are tress with a node for each of the basic operations one can perform on a tensor. Namely, tensor trees have nodes for addition of tensors, permutation of tensor indices, negation of tensors, scalar multiplication of tensors, group action on a tensor, tensor product of tensors, contraction of tensor indices, and evaluation of tensor indices. They also have nodes for tensors themselves.

Given a species `S`, we have a type of tensor tree for each map of the form `c : Fin n → S.C` in `OverColor S.C`. This restriction to `Fin n` is done for convenience.

Tensor trees are defined inductively through a number of constructors:

```
inductive TensorTree (S : TensorSpecies) : {n : ℕ} → (Fin n → S.C) → Type where
  /-- A general tensor node. -/
  | tensorNode {n : ℕ} {c : Fin n → S.C} (T : S.F.obj (OverColor.mk c)) :
    TensorTree S c
  /-- A node corresponding to the scalar multiple of a tensor by a element of the
    field. -/
  | smul {n : ℕ} {c : Fin n → S.C} : S.k → TensorTree S c → TensorTree S c
  /-- A node corresponding to negation of a tensor. -/
  | neg {n : ℕ} {c : Fin n → S.C} : TensorTree S c → TensorTree S c
```

```
/-- A node corresponding to the addition of two tensors. -/
| add {n : ℕ} {c : Fin n → S.C} : TensorTree S c → TensorTree S c → TensorTree
  S c
/-- A node corresponding to the action of a group element on a tensor. -/
| action {n : ℕ} {c : Fin n → S.C} : S.G → TensorTree S c → TensorTree S c
/-- A node corresponding to the permutation of indices of a tensor. -/
| perm {n m : ℕ} {c : Fin n → S.C} {c1 : Fin m → S.C}
    (σ : (OverColor.mk c) → (OverColor.mk c1)) (t : TensorTree S c) :
  TensorTree S c1
/-- A node corresponding to the product of two tensors. -/
| prod {n m : ℕ} {c : Fin n → S.C} {c1 : Fin m → S.C}
  (t : TensorTree S c) (t1 : TensorTree S c1) : TensorTree S (Sum.elim c c1 ∘
  finSumFinEquiv.symm)
/-- A node corresponding to the contraction of indices of a tensor. -/
| contr {n : ℕ} {c : Fin n.succ.succ → S.C} : (i : Fin n.succ.succ) →
  (j : Fin n.succ) → (h : c (i.succAbove j) = S.τ (c i)) → TensorTree S c →
  TensorTree S (c ∘ Fin.succAbove i ∘ Fin.succAbove j)
/-- A node corresponding to the evaluation of an index of a tensor. -/
| eval {n : ℕ} {c : Fin n.succ → S.C} : (i : Fin n.succ) → (x : ℕ) →
  TensorTree S c →
  TensorTree S (c ∘ Fin.succAbove i)
```

Each constructor here, e.g. `tensorNode`, `smul`, `neg`, etc., can be thought of as forming a different type of node in a tensor tree.

Since the interpretation of each of the constructors is down to how we turn them into a tensor, we discuss this before outlining each of the constructors in turn. The process of going from a tensor tree to a tensor is proscribed by a function `TensorTree S c → S.F c`, which is defined recursively as follows:

```
/-- The underlying tensor a tensor tree corresponds to. -/
def TensorTree.tensor {n : ℕ} {c : Fin n → S.C} : TensorTree S c → S.F.obj
    (OverColor.mk c) := fun
  | tensorNode t => t
  | smul a t => a • t.tensor
  | neg t => - t.tensor
  | add t1 t2 => t1.tensor + t2.tensor
  | action g t => (S.F.obj (OverColor.mk _)).ρ g t.tensor
  | perm σ t => (S.F.map σ).hom t.tensor
  | prod t1 t2 => (S.F.map (OverColor.equivToIso finSumFinEquiv).hom).hom
    ((S.F.μ _ _).hom (t1.tensor ⊗ₜ t2.tensor))
  | contr i j h t => (S.contrMap _ i j h).hom t.tensor
  | eval i e t => (S.evalMap i (Fin.ofNat' _ e)) t.tensor
```

Let us now discuss each of the constructors in turn.

tensorNode: The constructor `tensorNode` creates a tensor tree based on `c` from a tensor `t` in `S.F c`. This tensor tree consists of a single node that directly represents the tensor. Since all other tensor tree constructors require an existing tensor tree as input, `tensorNode` serves as the foundational base case for building more complex trees. As a diagram such a tree is:

$$\boxed{\texttt{t}}$$

Naturally, the tensor associated with this node is exactly the tensor provided during construction, as can be seen in `TensorTree.tensor`.

smul: The constructor smul takes a scalar `a` and an existing tensor tree `t` based on `c` and constructs a new tensor tree also based on `c`. Conceptually, this new tree has a root node labeled `smul a`, with the tensor tree `t` as its child. As a diagram such a tree is:

$$\begin{tikzcd} \boxed{\texttt{smul a}} \arrow[d] \\ \texttt{t} \end{tikzcd}$$

The tensor associated with this new tree is obtained by multiplying the tensor associated with `t` by the scalar `a`.

neg: The constructor `neg` takes an existing tensor tree `t` based on `c` and constructs a new tensor tree also based on `c`. This new tree has a root node labeled `neg`, with the tensor tree `t` as its child. As a diagram, we have:

$$\begin{tikzcd} \boxed{\texttt{neg}} \arrow[d] \\ \texttt{t} \end{tikzcd}$$

The tensor associated with this new tree is obtained by negating the tensor associated with `t`.

add: The constructor `add` takes two existing tensor trees `t1` and `t2`, based on the same `c`, and constructs a new tensor tree. This new tree has a root node labeled `add`, with the tensor trees `t1` and `t2` as its children. As a diagram, this corresponds to:

$$\begin{tikzcd} & \boxed{\texttt{add}} \arrow[dl] \arrow[dr] & \\ \texttt{t1} & & \texttt{t2} \end{tikzcd}$$

The tensor associated with this new tree is obtained by adding the tensors associated with `t1` and `t2`.

action: The constructor `action` takes a group element `g` of `S.G`, an existing tensor tree `t` based on `c`, and constructs a new tensor tree also based on `c`. This new tree has a root node labeled `action g`, with the tensor tree `t` as its child. As a diagram:

$$\begin{tikzcd} \boxed{\texttt{action g}} \arrow[d] \\ \texttt{t} \end{tikzcd}$$

The tensor associated with this new tree is obtained by acting on the tensor associated with `t` with the group element `g`.

perm: The constructor `perm` takes a morphism σ from `c` to `c1` in `OverColor S.C` and an existing tensor tree `t` based on `c`, and constructs a new tensor tree based on `c1`. This new tree has a root node labeled `perm` σ, with the tensor tree `t` as its child. As a diagram, this is given by:

$$\begin{tikzcd} \boxed{\texttt{perm}\ \sigma} \arrow[d] \\ \texttt{t} \end{tikzcd}$$

The tensor associated with this new tree is obtained by applying the image of the morphism σ under the functor `S.F` to the tensor associated with `t`.

prod: The constructor `prod` takes two existing tensor trees `t` based on `c` and `t1` based on `c1` and constructs a new tensor tree based on `Sum.elim c c1 ∘ finSumFinEquiv.symm` which is the map from `Fin (n + n1)` acting via `c i` on `0 ≤ i ≤ n -1` and via `c1 (i - n)` on `n ≤ i ≤ (n + n1) - 1`. This new tree has a root node labeled `prod`, with the tensor trees `t` and `t1` as its children. As a diagram, this is given by:

$$\begin{tikzcd} & \boxed{\texttt{prod}} \arrow[dl] \arrow[dr] & \\ \texttt{t} & & \texttt{t1} \end{tikzcd}$$

The tensor associated with this new tree is obtained by taking the tensor product of the tensors associated with `t` and `t1`, giving a vector in `S.F c ⊗ S.F c1`, using the tensorator of `S.F` to map this vector into a vector in `S.F.obj (OverColor.mk c ⊗ OverColor.mk c1)`, and finally using an isomorphism between `OverColor.mk c ⊗ OverColor.mk c1` and `OverColor.mk (Sum.elim c c1 ∘ finSumFinEquiv.symm)` to map this vector into `S.F (Sum.elim c c1 ∘ finSumFinEquiv.symm)`.

contr: The constructor `contr` firstly, takes an existing tensor tree `t` based on a `c : Fin n.succ.succ -> S.C`. Here `n.succ.succ` is $n+1+1$ with `succ` meaning the successor of a natural number. It also takes, an `i` of type `Fin n.succ.succ`, `j` of type `Fin n.succ` and a proof that `c (i.succAbove j) = S.τ (c i)`, where `i.succAbove` is the map from `Fin n.succ` to `Fin n.succ.succ` with a hole at `i`. The proof says that the color of the index `i.succAbove j` is the dual of the color of the index `i`, and thus these two indices can be contracted. Note that we use a `j` in `Fin n.succ` and `i.succAbove j` as the index to be contracted, instead of another index in `Fin n.succ.succ` to ensure the two indices to be contracted are not the same. The constructor outputs a new tensor tree based on `c ∘ i.succAbove ∘ j.succAbove`. This new tree has a root node labeled `contr i j`, with the tensor tree `t` as its child. As a diagram:

$$\begin{tikzcd} \boxed{\texttt{contr i j}} \arrow[d] \\ \texttt{t} \end{tikzcd}$$

The tensor associated with the new tensor tree is constructed as follows. We start with a tensor in `S.F c`. This is mapped into a vector in `(S.FD (c i) ⊗ S.FD (S.τ (c i))) ⊗ S.F (c ∘ Fin.succAbove i ∘ Fin.succAbove j)` via an equivalence from `S.F c`. The equivalence is constructed using an equivalence in `OverColor C` to extract `i` and `i.succAbove j` from `c`, then using the tensorator of `S.F`, and the fact that `c (i.succAbove j) = S.τ (c i)`. Using the contraction for `c i`, we then get a vector in `𝟙 ⊗ S.F (c ∘ Fin.succAbove i ∘ Fin.succAbove j)` which is then mapped to a tensor in `S.F (c ∘ Fin.succAbove i ∘ Fin.succAbove j)` using the left-unitor of `Rep S.G S.k`. This is all contained within the `S.contrMap` appearing in the definition of function `TensorTree.tensor`.

eval: The final constructor `eval` takes an existing tensor tree `t` based on a `c : Fin n.succ -> S.C`, an `i` of type `Fin n.succ`, and a natural number x. The constructor outputs a new tensor tree based on `c ∘ Fin.succAbove i`. This new tree has a root node labeled `eval i x`, with the tensor tree `t` as its child. As a diagram, this corresponds to:

$$\begin{tikzcd} \boxed{\texttt{eval i x}} \arrow[d] \\ \texttt{t} \end{tikzcd}$$

The tensor associated with the new tensor tree is constructed as follows. We start with a tensor in `S.F c`. This is mapped into a vector in `S.FD (c i) ⊗ S.F (c ∘ Fin.succAbove i)` via an equivalence from `S.F c`. The equivalence is constructed using an equivalence in `OverColor C` to extract `i` from `c`, and then using the tensorator of `S.F`. Using the evaluation for `c i` at the basis element indicated by `x` (if `x` is too big a natural number for the number of basis elements it defaults to `0`), we then get a vector in `𝟙 ⊗ S.F (c ∘ Fin.succAbove i)`. This mapping should be thought of as occurring in the category of modules over `S.k`, rather than in `Rep S.G S.k`, since the evaluation is not invariant under the group action. We then finally map into `S.F (c ∘ Fin.succAbove i)` using the left-unitor. This is all contained within the `S.evalMap` appearing in the definition of function `TensorTree.tensor`.

The main reason tensor trees are easy to work with is the following. Define a subtree of a tensor trees to be a node and all child nodes of that node. If `t` is a tensor tree and `s` a subtree of `t`, we can replace `s` in `t` with another tensor tree `s'` to get a new overall tensor-tree `t'`. If `s` and `s'` have the same underlying tensor, then `t` and `t'` will also.

In Lean this property manifests in a series of lemmas. For instance, for the `contr` constructor we have the lemma:

```
lemma contr_tensor_eq {n : ℕ} {c : Fin n.succ.succ → S.C} {T1 T2 : TensorTree S c}
    (h : T1.tensor = T2.tensor) {i : Fin n.succ.succ} {j : Fin n.succ}
    {h' : c (i.succAbove j) = S.τ (c i)} :
    (contr i j h' T1).tensor = (contr i j h' T2).tensor := by
  simp only [Nat.succ_eq_add_one, contr_tensor]
  rw [h]
```

These lemmas allow us to navigate to certain places in tensor trees and replace subtrees with other subtrees. We will see this used extensively in the examples in Section 3.

2.4. SYNTAX AND THEIR MAP TO TENSOR TREES

Syntax allows index notation in Lean code to look similar to pen-and-paper index notation. The syntax is turned into a tensor tree through a process called elaboration. Although, elaboration is not formally-verified in Lean, the tensor tree it outputs is.

Instead of delving into the finer details of this process, we give illustrative examples.

In what follows we will assume that `T`, `T1`, etc are tensors defined as elements of `S.F c`, `S.F c1`, etc for some tensor species `S` and some `c : Fin n → S.C`, `c1 : Fin n1 → S.C`, etc for which the expressions below make sense.

The syntax allows us to write the following

`{T	μ ν}ᵀ`	`tensorNode T`

for a tensor node. Here the `μ` and `ν` are free variables and it does not matter what we call them - Lean will elaborate the expression in the same way. The elaborator also knows how many indices to expect for a tensor `T` and will raise an error if the wrong number are given. The `{_}ᵀ` notation is used to tell Lean that the syntax is to be treated as a tensor expression. Throughout this section we will use the two-sided boxes given above, which denote the syntax on the left and the expression it is elaborated to on the right.

We can write e.g.,

`{T	μ ν}ᵀ.tensor`	`(tensorNode T).tensor`

to get the underlying tensor. We get this notation from the way `TensorTree.tensor` is defined with the prefix `TensorTree.` .

Note that we do not have indices which are upper or lower as one would expect from pen-and-paper notation (e.g., $\eta^{\mu}{}_{\nu}$). There is one primary reason for this; whether an index is upper or lower does not carry any information, since this information comes from the tensor itself. Also, for something like complex Lorentz tensors, there are three different types of upper-index, so such notation would be complicated.

If we want to evaluate an index we can put an explicit index in place of μ or ν above, for example:

`{T	1 ν}ᵀ`	`eval 0 1 (tensorNode T)`

The syntax and elaboration for negation, scalar multiplication and the group action are fairly similar. For negation we have:

`{T	μ ν}ᵀ`	`neg (tensorNode T)`

For scalar multiplication by $a \in k$ we have:

`{a •ₜ T	μ ν}ᵀ`	`smul a (tensorNode T)`

For the group action of $g \in G$ on a tensor T we have:

`{g •ₐ T	μ ν}ᵀ`	`action g (tensorNode T)`

The product of two tensors is also fairly similar, with us having:

`{T	μ ν ⊗ T2	σ}ᵀ`	`prod (tensorNode T) (tensorNode T2)`

The syntax for contraction is:

`{T	μ ν ⊗ T2	ν σ}ᵀ`	`contr 1 1 rfl (prod (tensorNode T) (tensorNode T2))`

On the right-hand side the first argument (`1`) of `contr` is the index of the first `ν` on the left-hand side, the second argument (also `1`) is the second index. The `rfl` is a proof that the colors of the two contracted indices are actually dual to one another. If they are not, this proof will fail and the elaborator will complain. It will also complain if more than two indices are trying to be contracted, although this depends on where exactly the indices sit in the expression, for example

`{T	μ ν ⊗ T2	ν ν}ᵀ`	`(prod (tensorNode T) (contr 0 0 rfl (tensorNode T2)))`

works fine because the inner contraction is computed before the product.

We now turn to addition. Our syntax allows for e.g., `{T | μ ν + T2 | μ ν}ᵀ` and also `{T | μ ν + T2 | ν μ}ᵀ`, provided that the indices are of the correct color (which Lean will check). The elaborator handles both these cases and generalizations thereof by adding a permutation node. Thus we have

`{T	μ ν + T2	μ ν}ᵀ`	`add (tensorNode T) (perm _ (tensorNode T2))`

where the `_` is a placeholder for the permutation, something we will use frequently in what follows. For the case above the permutation will be the identity, but for

`{T	μ ν + T2	ν μ}ᵀ`	`add (tensorNode T) (perm _ (tensorNode T2))`

it will be the permutation for the two indices.

Despite not forming part of a node in our tensor tree, we also give syntax for equality. This is done in a very similar way to addition, with the addition of a permutation node to account for e.g., expressions like $T_{\mu\nu} = T_{\nu\mu}$.

`{T	μ ν = T2	ν μ}ᵀ`	`(tensorNode T).tensor = (perm _ (tensorNode T2)).tensor`

Note that in the elaborated expression we ask for equality of tensors through `.tensor`. Tensors are, after all, the objects we care about.

With this syntax we can write complicated tensor expressions in a way close to pen-and-paper index notation. We will see examples of this in the next section.

3. EXAMPLES

We give two examples in this section. The first example is a simple theorem involving index notation and tensor trees. We will demonstrate, in rather explicit detail, how we can manipulate tensor trees to solve such theorems. The second example we give will show a number of definitions related to Pauli matrices and bispinors in HepLean concerning index notation. Here we won't give as much detail, the point being to show the reader the broad use of our construction.

3.1. EXAMPLE 1: SYMMETRIC AND ANTISYMMETRIC TENSOR

If $A^{\mu\nu}$ is an antisymmetric tensor and $S_{\mu\nu}$ and S is a symmetric tensor, then it is true that $A^{\mu\nu}S_{\mu\nu} = -A^{\mu\nu}S_{\mu\nu}$. In Lean this result, and it's proof are written as follows:

```
lemma antiSymm_contr_symm
    {A : complexLorentzTensor.F.obj (OverColor.mk ![Color.up, Color.up])}
    {S : complexLorentzTensor.F.obj (OverColor.mk ![Color.down, Color.down])}
    (hA : {A | μ ν = - (A | ν μ)}ᵀ) (hs : {S | μ ν = S | ν μ}ᵀ) :
    {A | μ ν ⊗ S | μ ν = - A | μ ν ⊗ S | μ ν}ᵀ := by
  conv =>
    lhs
    rw [contr_tensor_eq <| contr_tensor_eq <| prod_tensor_eq_fst <| hA]
    rw [contr_tensor_eq <| contr_tensor_eq <| prod_tensor_eq_snd <| hs]
    rw [contr_tensor_eq <| contr_tensor_eq <| prod_perm_left _ _ _ _]
    rw [contr_tensor_eq <| contr_tensor_eq <| perm_tensor_eq <| prod_perm_right _ _
    _ _]
    rw [contr_tensor_eq <| contr_tensor_eq <| perm_perm _ _ _]
    rw [contr_tensor_eq <| perm_contr_congr 1 2]
    rw [perm_contr_congr 0 0]
    rw [perm_tensor_eq <| contr_contr _ _ _]
    rw [perm_perm]
    rw [perm_tensor_eq <| contr_tensor_eq <| contr_tensor_eq <| neg_fst_prod _ _]
    rw [perm_tensor_eq <| contr_tensor_eq <| neg_contr _]
    rw [perm_tensor_eq <| neg_contr _]
  apply perm_congr _ rfl
  decide
```

Let us break this down. The statements

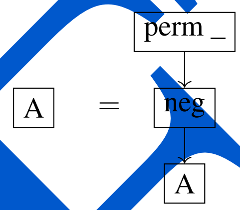
```
{A : complexLorentzTensor.F.obj (OverColor.mk ![Color.up, Color.up])}
{S : complexLorentzTensor.F.obj (OverColor.mk ![Color.down, Color.down])}
```

are simply defining A and S to be tensors of type $A^{\mu\nu}$ and $S_{\mu\nu}$ respectively. Here `![Color.up, Color.up]` is shorthand for the map `Fin 2 → complexLorentzTensor.C` that sends 0 to `Color.up` and 1 to `Color.up`.

The parameter `hA` is stating that A is antisymmetric. Expanded in terms of tree diagrams we have

`hA : {A | μ ν = - (A | ν μ)}`$^{\mathrm{T}}$

Description: The tensor A is antisymmetric.

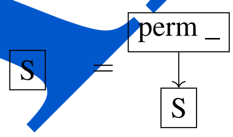


In the tensor-tree diagram on the right-hand side we implicitly mean equality of the underlying tensors of the given trees. This will be left implicit throughout.

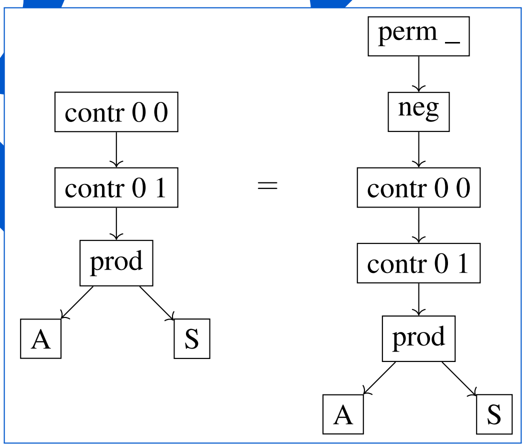
Similarly, the parameter `hs` is stating that S is symmetric. Expanded in terms of tree diagrams

`hS : {S | μ ν = S | ν μ}`$^{\mathrm{T}}$

Description: The tensor `S` is symmetric.



The line `{A | μ ν ⊗ S | μ ν = - A | μ ν ⊗ S | μ ν}`$^{\mathrm{T}}$ is the statement we are trying to prove. In terms of tree diagrams it says that:

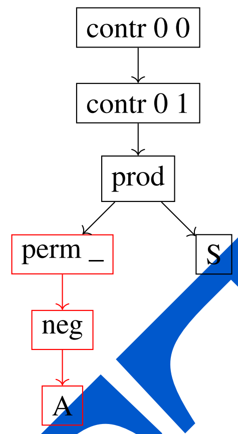


The perm here actually does nothing, but is included by Lean.

The lines of the proof in the `conv` block are manipulations of the tensor tree on the LHS of the equation. The `rw` tactic is used to rewrite the tensor tree using the various lemmas. We work through each step in turn.

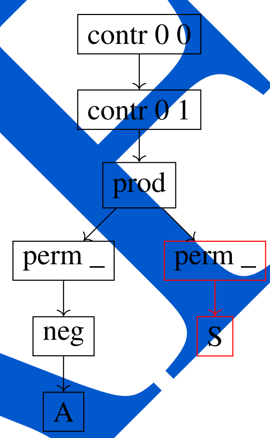
```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_tensor_eq_fst <| hA]
```

Description: Here `contr_tensor_eq` and `prod_tensor_eq_fst` navigate to the correct place in the tensor tree, whilst `hA` replaces the node `A` with the RHS of `hA`.



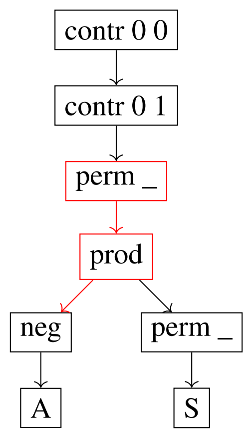

```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_tensor_eq_snd <| hs]
```

Description: Here `contr_tensor_eq` and `prod_tensor_eq_fst` navigate to the correct place in the tensor tree, whilst `hS` replaces the node `S` with the RHS of `hS`.



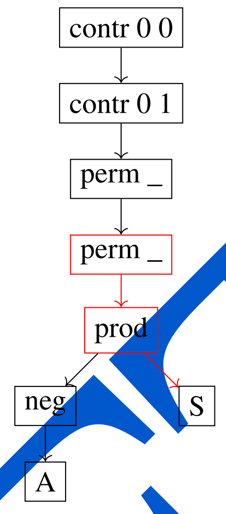

```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_perm_left _ _ _ _]
```

Description: Here `contr_tensor_eq` navigates to the correct place in the tensor tree, whilst `prod_perm_left` moves the permutation on the left through the product.



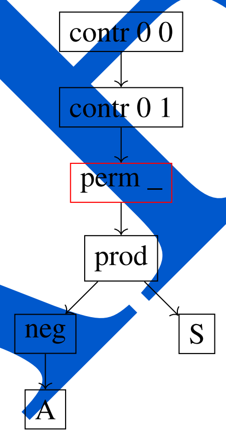
```
rw [contr_tensor_eq <| contr_tensor_eq
<| perm_tensor_eq <| prod_perm_right _ _
_ _]
```

Description: Here `contr_tensor_eq` and `perm_tensor_eq` navigate to the correct place in the tensor tree, whilst `prod_perm_right` moves the permutation on the right through the product.



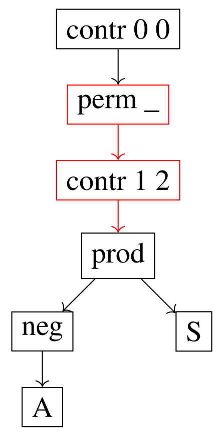

```
rw [contr_tensor_eq <| contr_tensor_eq
<| perm_perm _ _ _]
```

Description: Here `contr_tensor_eq` navigates to the correct place in the tensor tree, whilst `perm_perm` uses functoriality to combine the two permutations.



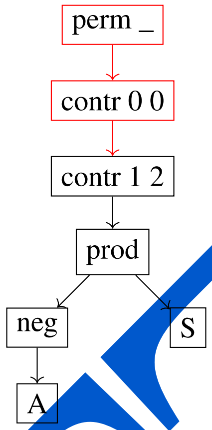

```
rw [contr_tensor_eq <| perm_contr_congr
1 2]
```

Description: Here `contr_tensor_eq` navigates to the correct place in the tensor tree, whilst `perm_contr_congr` moves the permutation through the contraction, and simplifies the contraction indices to `1` and `2` (Lean will check if this is correct).



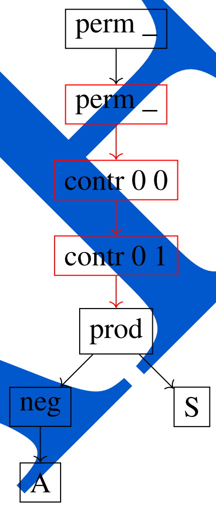
```
rw [perm_contr_congr 0 0]
```

Description: Here `perm_contr_congr` moves the permutation through the contraction, and simplifies the contraction indices to `0` and `0` (Lean will check if this is correct).



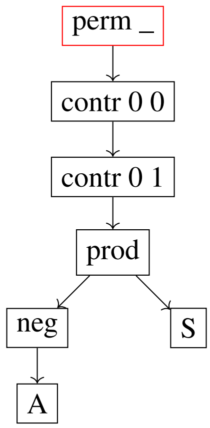

```
rw [perm_tensor_eq <| contr_contr _ _ _]
```

Description: Here `perm_tensor_eq` navigates to the correct place in the tensor tree, whilst `contr_contr` swaps the two contractions, in the meantime inducing a permutation.



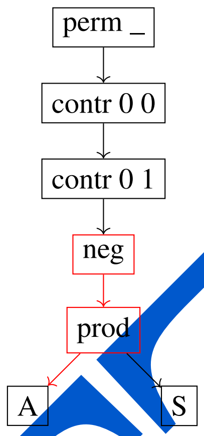

```
rw [perm_perm]
```

Description: Here `perm_perm` uses functoriality to combine the two permutations.



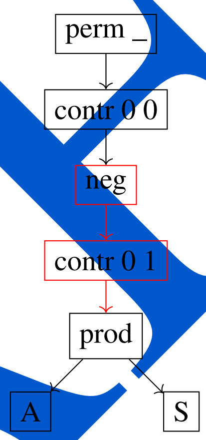
```
rw [perm_tensor_eq <| contr_tensor_eq <|
contr_tensor_eq <| neg_fst_prod _ _]
```

Description: Here `perm_tensor_eq` and `contr_tensor_eq` navigate to the correct place in the tensor tree, whilst `neg_fst_prod` moves the negation through the product.



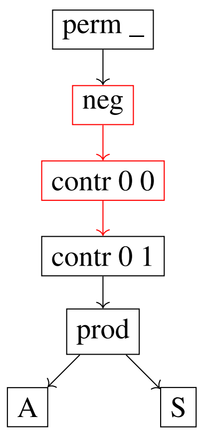

```
rw [perm_tensor_eq <| contr_tensor_eq <|
neg_contr _]
```

Description: Here `perm_tensor_eq` and `contr_tensor_eq` navigate to the correct place in the tensor tree, whilst `neg_contr` moves the negation through the first contraction.




```
rw [perm_tensor_eq <| neg_contr _]
```

Description: Here `perm_tensor_eq` navigates to the correct place in the tensor tree, whilst `neg_contr` moves the negation through the second contraction.



The remainder of the proof

```
apply perm_congr _ rfl
decide
```

helps Lean to understand that the two sides of the equation are equal.

3.2. EXAMPLE 2: PAULI MATRICES AND BISPINORS

Using the formalism we have set up thus far, it is possible to define Pauli matrices and bispinors as complex Lorentz tensors.

The Pauli matrices appear in HepLean as follows:

```
/-- The Pauli matrices as the complex Lorentz tensor ‘σ^μ^α^{dot β}‘. -/
def pauliContr := {PauliMatrix.asConsTensor | ν α β}ᵀ.tensor

/-- The Pauli matrices as the complex Lorentz tensor ‘σ_μ^α^{dot β}‘. -/
def pauliCo := {η' | μ ν ⊗ pauliContr | ν α β}ᵀ.tensor

/-- The Pauli matrices as the complex Lorentz tensor ‘σ_μ_α_{dot β}‘. -/
def pauliCoDown := {pauliCo | μ α β ⊗ εL' | α α' ⊗ εR' | β β'}ᵀ.tensor

/-- The Pauli matrices as the complex Lorentz tensor ‘σ^μ_α_{dot β}‘. -/
def pauliContrDown := {pauliContr | μ α β ⊗ εL' | α α' ⊗ εR' | β β'}ᵀ.tensor
```

The first of these definitions depends on `PauliMatrix.asConsTensor` which is defined using an explicit basis expansion as a map

```
𝟙_ (Rep ℂ SL(2,ℂ)) → complexContr ⊗ Fermion.leftHanded ⊗ Fermion.rightHanded
```

which Lean knows how to treat as a tensor. Here `complexContr` is the representation of complex Lorentz vectors under $SL(2,\mathbb{C})$ defined above as `Lorentz.complexContr`.

In these expressions we have the appearance of metrics. The metric `η'` is what is usually denoted $\eta_{\mu\nu}$, the metric `η` is what is usually denoted $\eta^{\mu\nu}$, the metric `εL'` is what is usually denoted $\varepsilon_{\alpha\alpha'}$, and the metric `εR'` is what is usually denoted $\varepsilon_{\dot\beta\dot\beta'}$.

With these we can also define bispinors:

```
/-- A bispinor ‘p^{aa}‘ created from a lorentz vector ‘p^μ‘. -/
def contrBispinorUp (p : complexContr) :=
  {pauliCo | μ α β ⊗ p | μ}ᵀ.tensor

/-- A bispinor ‘p_{aa}‘ created from a lorentz vector ‘p^μ‘. -/
def contrBispinorDown (p : complexContr) :=
  {εL' | α α' ⊗ εR' | β β' ⊗ contrBispinorUp p | α β}ᵀ.tensor

/-- A bispinor ‘p^{aa}‘ created from a lorentz vector ‘p_μ‘. -/
def coBispinorUp (p : complexCo) := {pauliContr | μ α β ⊗ p | μ}ᵀ.tensor

/-- A bispinor ‘p_{aa}‘ created from a lorentz vector ‘p_μ‘. -/
def coBispinorDown (p : complexCo) :=
  {εL' | α α' ⊗ εR' | β β' ⊗ coBispinorUp p | α β}ᵀ.tensor
```

Here `complexCo` corresponds to the representation of covariant complex Lorentz vectors, defined above as `Lorentz.complexCo`.

Using these definitions we can start to prove results about the pauli matrices and bispinors. These proofs essentially rely on the sorts of manipulations in the last section, although in some cases we expand tensors in terms of a basis and use rules about how the basis interacts with the operations in a tensor tree.

As an example, we prove the following lemmas

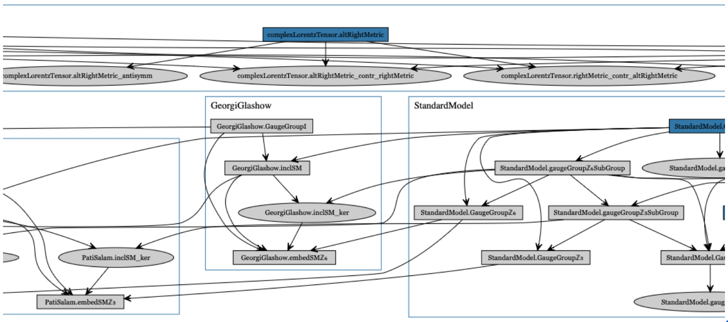


Figure 2: A screenshot of the informal dependency graph in HepLean. Gray nodes indicate informal results, whilst blue nodes indicate results already formalised.

```
lemma coBispinorDown_eq_pauliContrDown_contr (p : complexCo) :
  {coBispinorDown p | α β = pauliContrDown | μ α β ⊗ p | μ}ᵀ := by
```

and

```
/-- The statement that ‘η_{μν} σ^{μ α dot β} σ^{ν α’ dot β’} = 2 ε^{αα’} ε^{dot
    β dot β’}‘. -/
theorem pauliCo_contr_pauliContr :
    {pauliCo | ν α β ⊗ pauliContr | ν α’ β’ = 2 •ₗ εL | α α’ ⊗ εR | β β’}ᵀ := by
```

We do not give the proofs of these lemmas explicitly. The former, however, is a fairly simple application of associativity of the tensor product, and shuffling around of the contractions.

4. FUTURE WORK

The scale of formalizing all results regarding index notation is a task that surpasses the capacity of any single individual. Inspired by the Lean community's blueprint projects, we have added to HepLean informal lemmas related to index notation and tensors. An example of such is:

```
informal_lemma coBispinorUp_eq_metric_contr_coBispinorDown where
  math :≈ "{coBispinorUp p | α β = εL | α α’ ⊗ εR | β β’ ⊗ coBispinorDown p | α
    ’ β’ }ᵀ"
  proof :≈ "Expand ‘coBispinorDown‘ and use fact that metrics contract to the
    identity."
  deps :≈ [‘‘coBispinorUp, ‘‘coBispinorDown, ‘‘leftMetric, ‘‘rightMetric]
```

These informal lemmas are written in strings, and are not type checked. They also include dependencies indicating what definitions and lemmas we expect to be used in their statement or proof. They are intended to be a guide for future formalization efforts, either by humans or automated. All the informal lemmas and the related informal definitions which are in HepLean are given on the HepLean website in a dependency graph, part of which is shown in Figure 2.

As demonstrated in our earlier examples, manipulating tensor expressions can involve tedious calculations, especially when dealing directly with tensor trees. In future, we intend to automate many of these routine steps by developing suitable tactics within Lean. We are optimistic that the structured nature of tensor trees will lend itself well to such automation, thereby streamlining computations and enhancing the efficiency of formal proofs involving index notation and tensor species.

There are two primary directions in which we can extend the concepts presented in this work. First, we could incorporate the spinor-helicity formalism, which is used in the study of scattering amplitudes. Second, we could extend our approach to encompass tensor *fields*, their derivatives etc. We do not anticipate any insurmountable challenges in pursuing these extensions. They represent promising avenues for future research and have the potential to significantly enhance the utility of formal methods in physics.

ACKNOWLEDGMENTS

I thank Tarmo Uustalu for helpful discussions related to this project. I also thank Jan Idziak, Tomas Skrivan, and Thomas Murrills for dicussions related to index notation near the start of this project. I thank members of the Lean Zulip for answering my Lean related questions.

REFERENCES

[1] Joseph Tooby-Smith. HepLean: Digitalising high energy physics. 5 2024. arXiv:2405.08863.

[2] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015. doi:10.1007/[illegible]21401-6_26.

[3] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.[illegible]3372885.3373824.

[4] Kevin Buzzard and Richard Taylor. A lean proof of fermat's last theorem. 2024.

[5] Maxwell P Bobbin, Samiha Sharlin, Parivash Feyzishendi, An Hong Dang, Catherine M Wraback, and Tyler R Josephson. Formalizing chemical physics using the lean theorem prover. *Digital Discovery*, 3(2):264–280, 2024. doi:/10.1039/D[illegible]D00077J.

[6] Tomáš Skřivan. Scilean: Scientific computing assistant. *GitHub repository*. URL: https://github.[illegible]copivo/[illegible]Lean.

[7] Jean-Philippe Bernardy and Patrik Jansson. Domain-specific tensor languages. *arXiv preprint arXiv:2312.02664*, 2023.

[8] AndrÃ© Joyal and Joachim Kock. Feynman graphs, and nerve theorem for compact symmetric multicategories (extended abstract). *Electronic Notes in Theoretical Computer Science*, 270(2):105–113, 2011. Proceedings of the 6th International Workshop on Quantum Physics and Logic (QPL 2009). URL: https://www.sciencedirect.com/science/article/pii/S157106611[illegible]260, doi:[illegible]0.1016/j.entcs.2011.01.025.

[9] Sophie Raynor. Graphical combinatorics and a distributive law for modular operads. *Advances in Mathematics*, 392:108011, 2021.