Index notation in Lean 4

Joseph Tooby-Smith
Reykjavik University, Reykjavik, Iceland

November 1, 2024

Abstract

The physics community relies on index notation to effectively manipulate tensors of specific types. 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. In the background, our implementation leverages a novel application of category theory.

The file you are currently reading is a draft. There will be typos, mistakes, sentences that don't make sense, notation that is not defined etc.

1. INTRODUCTION

In previous work, the author initiated the digitalization (or formalization) of high energy physics results using the interactive theorem prover Lean 4 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 using dependent-type theory for correctness. The HepLean project is driven by four primary motivations: to facilitate easier look-up of results through linear information storage; to support the creation and proof of new results using automated tactics and AI tools; to facilitate correctness checking of result in high energy physics; and to introduce new pedogocial 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, which aims to formalize mathematics. Indeed, HepLean is built downstream of Mathlib, meaning it has Mathlib as a dependency and uses many of the definitions and theorems thereof. Other projects include the ongoing effort led by Kevin Buzzard to formalize the proof of Fermat's Last Theorem into Lean. In the realm of the sciences, the paper [?] looks at absorption theory, thermodynamics, and kinematics in Lean. Whilst the package SciLean [?], is a push in the direction of scientific computing within Lean.

Physicists heavily rely 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 specfic types of tensors and operations between them, with tensors forming a backbone of morden physics.

Because of the importance of index notation, it is crucial to have a way to write index notation in Lean. This will make results from high energy physics easier to write and prove in Lean. Additionally, it will make the syntax more familiar to high energy physicists. Such an implementation is the subject of this paper. It is a challange because, not only do we need an implentation that is nice and flexiable to use,

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

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

we also need it to be formally well-defined and rigourous. We hope that the implementation presneted here will not only enhances usability of Lean but also promotes the adoption of formal methods in the physics community.

As a taster of what is to come, our implmentation allows to write results like the following in Lean:

```
{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 which the correct contraction of indices and permutation of indices between the sides of the expression.

Previous implmentations of index notation have been made in programming languages like Haskell [?]. However, these implementations 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 beleive is) a novel solution.

This paper is split into two main sections. Section 2 dicusses the implementation of index notation into Lean. Section js: ref gives two examples of theorems and proofs using index notation. A more minor section, section js: ref disucsses the future work related to this project.

2. IMPLEMENTATION OF INDEX NOTATION INTO LEAN 4

Our implementation of index notation in Lean can be broken down into three main components, illustrated in Figure 1. The first component is the *syntax for tensor expressions*, which is what users interact with when writing results in Lean. This syntax closely mirrors the notation familiar to physicists, making it intuitive and accessible. It appears directly in the Lean files and can be thought of as an informal string that represents the tensor expressions. The code snippet above, involving pauli matrices, is an example of this.

The second component involves transforming this syntax into a *tensor tree*. The tensor tree is a formal mathematical representation of the tensor expression. By parsing the syntax into a structured tree, we establish a rigorous foundation that captures the tensor expression. This formal representation allows us to easily manipulate tensor expressions and prove results related to them in a way that Lean accepts as formal.

The third and final component is the conversion of the tensor tree into an actual *tensor*. This process utilizes properties of the symmetric-monoidal category of representations to translate the tensor tree into a tensor.

Although Lean processes information from left to right in Figure 1, starting with the syntax and proceeding to the tensor tree, and then finally (when we ask it to) to the tensors, it is more effective to discuss the implementation from right to left. Starting with tensors is advantageous because they are the primary objects of interest. The left and middle parts of the diagram can be thought of as intermediate stages that facilitate the manipulation and understanding of these tensors and their expressions.

2.1.1 Building blocks of tensors and color

Tensors of a species, such as complex Lorentz tensors, are constructed from a set of building block representations of a group G over a field k.

For complex Lorentz tensors, the group G is $SL(2,\mathbb{C})$, the field k is the field of complex numbers and six building block representations. The six building block representations 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.

An example of how these representations are defined in Lean is given below for the representation of left-handed Weyl fermions, `Fermion.leftHanded`:

```
/-- 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 definiton 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 definiton 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 explainatary comments to this code, not seen in the actual Lean code, to give the reader an idea of what each part does.

Moving on, to each of the building block representations, we assign a unique label, which we refer to as a *color*. We then get a type of colors for a given species of tensors, which in this paper we will denote C.

For complex Lorentz tensors we associate the color `upL` to the representation `Fermion.leftHanded`, `downL` to `Fermion.altLeftHanded`, `upR` to `Fermion.rightHanded`, `downR` to `Fermion.altRightHanded`, `up` to `Lorentz.complexContr` and `down` to `Lorentz.complexCo`. The type of colors is defined in Lean as an inductive type as follows:

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

This can roughly be thought of as the type with six elements. We will see a much more complicated form of inductive type when we come to define tensor trees.

The assignment of colors to their corresponding representations is done via a functor F_D from the set C to the category of k-representations of G, which we denote Rep k G. That, is a functor

$$F_D : C \Rightarrow \mathrm{Rep}\, k\, G\,. \tag{1}$$

Since the source category of this functor is just a set, we could have instead defined it as a function - however we will see later that we want to lift this functor to a symmetric monoidal functor with a more sophiscated source category. For this reason, it is useful to have a functor here.

However, been a functor from a set, means that defining it for specific cases in Lean is somewhat stright forard. For complex Lorentz tensors this functor is defined as follows:

```
FDiscrete := 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
```

This code snippet actually forms part of a larger structure called a tensor species, which we will get on to shortly.

2.1.2 General tensors

Given any type C, which we will think of as a type of colors, we can define the category $\mathrm{T}^{\times}_{/C}$ as follows. Objects are functions $f : X \to C$ for some type X. A morphism from $f : X \to C$ to $g : Y \to C$ is a bijection $\phi : X \to Y$ such that $f = g \circ \phi$. This category is equivalent to the core of the category of types sliced over C, which explains the origin of the notation. In Lean we denote this category as `OverColor C`, although we will use the shorter notation $\mathrm{T}^{\times}_{/C}$ here when we can.

The category $\mathrm{T}^{\times}_{/C}$ is not any old category, it carries a symmetric monoidal structure, which we will denote $\otimes$. The structure such that $f \otimes g$ for objects f and g is the map $X \oplus Y \to C$ where $\oplus$ denotes the disjoint union of types.

The functor F_D defined above, can be lifted to a symmetric monoidal functor F from $\mathrm{T}^{\times}_{/C}$ to $\mathrm{Rep}\, k\, G$. This functor takes $f : X \to C$ to the tensor product $\bigotimes_{x \in X} F_D(f(x))$ and morphisms to the linear maps of representations corresponding to reindexings of tensor products. This construction is general, and functorial. In otherwords, there is a functor

$$\mathrm{lift} : \mathrm{Fun}(C, \mathrm{Rep}kG) \Rightarrow \mathrm{SymmMonFun}(\mathrm{T}^{\times}_{/C}, \mathrm{Rep}kG) \tag{2}$$

In Lean we define this functor, along with the corresponding lift of F_D, F.

We can think of an object $f : X \to C$ of $\mathrm{T}^{\times}_{/C}$ as a type of indices X, and a specification of what repreesntation each index is associated to. The representation $F(f)$ is then the tensor product of each of these representations, so that vectors $v \in F(f)$ can be thought of as tensors with indices indexed by X of color C (e.g. 'up' or 'down').

With this in mind, we define a general tensor of a given species as a vector in the representation $F(f)$ for some f in $\mathrm{T}^{\times}_{/C}$.

In physics, we typically focus on objects $f : X \to C$ in $\mathrm{T}^{\times}_{/C}$ where X is a finite type of the form $\mathrm{Fin}\, n$ for some n. Here, $\mathrm{Fin}\, n$ represents the type of natural numbers less than n, i.e., $\{0, 1, \ldots, n-1\}$. When appropraite in what follows, we will restrict to these objects.

In Lean for a map of types `f : X \to C`, what we have been writing as $F(f)$ is written as `S.F.obj (OverColor.mk f)`. The object `S` is the tensor species, which we will define shortly. The `OverColor.mk` tells lean that the function `f` should be considered as an object of `OverColor C`, not just as an object of type `X -> C`. js: fix arrows here

2.1.3 Tensor operations

js: In rewrite - I got up to here.

Now we have discussed how tensors of a given species can be formally defined, we can define basic operations on these tensors.

The simipalist of these operations are addition, scalar multiplication and group action. Addition and scalar multiplication are given to us for free from the vector space structure on $F(f)$. The action of the group G on tensors also comes for free from the fact that $F(f)$ lives in RepkG.

The next simplest operation is permutation of indices (or building block representations). Such permutations arise from applying F to morphisms in $\mathrm{T}^{\times}_{/C}$. Theses sorts of permutations are often implicit in the notation physicists use, but arise in tensor expressions such as $T_{\mu\nu} = T_{\nu\mu}$.

The next operation is the tensor product of two tensors. Given $f : X \to C$ and $g : Y \to C$, and tensors $t \in F(f)$ and $s \in F(g)$ we can form a vector in $F(f) \otimes F(g)$. Formally this can be done by taking the tensor product of the morphisms $\mathbb{1} \to F(f)$ and $\mathbb{1} \to F(g)$ in the category of vector spaces over k. We can then use the tensorate of F to get a tensor in $F(f \otimes g)$.

We now turn to the slightly more complicated operation of contraction of indices, and the related metric and unit. To define the contraction of introduce an involution $\tau : C \to C$. For complex Lorentz tensors this is defined as follows

```
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
```

We say that τ takes a color to it's dual. With τ and F_D we can define the functor $F_\tau : C \Rightarrow RepkG$ which takes c to $F(c) \otimes F(\tau c)$. Letting $\mathbb{1}$ be the constant functor from C landing on the, we basic data for contraction is contained in a natural transformation from F_τ to $\mathbb{1}$. That is, for each $c \in C$ an equivariant map from $F(c) \otimes F(\tau c)$ to the trivial representation.

To see how this is can be used to contract indices, consider a tensor $t \in F(f)$, for $f : \mathrm{Fin}\ n.succ.succ \to C$. Here Fin $n.succ.succ$ is the type of all number $0, 1, 2, \dots, n+1$ less than $n.succ.succ = n+2$. Choosing two distinct indices $i, j :$ Fin $n.succ.succ$ we can contract them using the following chain of maps

$$F f \equiv F_\tau(fi) \otimes F(f'i) \to \mathbb{1} \otimes F(f'i) \to F(f'i) \tag{3}$$

Here $f' : \mathrm{Fin}\ n \to C$ is the function defived from f by removing the indices i and j. The first equivalence is somewhat complicated to define formally, but essentially involves extracting i and j.

The metric and unit are defined in a similar way. The metric is a natural transformation from $\mathbb{1}$ to $F_D \otimes F_D$. That is, for each $c \in C$ an equivariant map from the trivial representation to $F_D(c) \otimes F_D(c)$. The unit is a natural transformation from $\mathbb{1}$ to F'_τ which takes c to $F_D(\tau c) \otimes F_D(c)$. That is, for each $c \in C$ an equivariant map from the trivial representation to $F_D(\tau c) \otimes F_D(c)$.

The choice of contraction, metric and unit are subject to a number of conditions.

- Contraction must be symmetric.

$$\begin{tikzcd}
F_D(c) \otimes F_D(\tau c) \arrow[r, "\beta"] \arrow[d, "\mathscr{C}\, c"'] & F_D(\tau c) \otimes F_D(c) \arrow[d, "\mathbb{1} \otimes F_D t"] \\
\mathbb{I} & F_D(\tau c) \otimes F_D(\tau\tau c) \arrow[l, "\mathscr{C}\, \tau(c)"]
\end{tikzcd} \tag{4}$$

- Contraction with the unit does nothing.

$$\begin{tikzcd}
F_D(c) \arrow[r] & F_D(c) \otimes \mathbb{I} \arrow[r] & F_D(c) \otimes (F_D(\tau c) \otimes F_D(c)) \arrow[d] \\
\mathbb{I} \otimes F_D(c) \arrow[u] & & (F_D(c) \otimes F_D(\tau c)) \otimes F_D(c) \arrow[ll]
\end{tikzcd} \tag{5}$$

- The unit is symmetric

$$\begin{tikzcd}
\mathbb{I} \arrow[r] \arrow[d] & F_D(\tau c) \otimes F_D(c) \\
F_D(\tau\tau c) \otimes F_D(\tau c) \arrow[r] & F_D(\tau c) \otimes F_D(\tau\tau c) \arrow[u]
\end{tikzcd} \tag{6}$$

- Contraction with the metric its dual returns the unit

$$\begin{tikzcd}
\mathbb{I} \arrow[r] \arrow[d] & F_D(\tau c) \otimes F_D(c) \\
(F_D c \otimes F_d c) \otimes (F_D(\tau c) \otimes F_D(\tau c)) \arrow[d] & F_D(c) \otimes F_D(\tau c) \arrow[u] \\
F_D c \otimes (F_d c \otimes (F_D(\tau c) \otimes F_D(\tau c))) \arrow[r] & F_D c \otimes ((F_d c \otimes F_D(\tau c)) \otimes F_D(\tau c)) \arrow[u]
\end{tikzcd} \tag{7}$$

In Lean we quite write these conditions in this way, instead we give them in a simpiler equivalent form in which we apply the maps in the above diagram to pure tensors. We will see how they are written explicitly in Lean when we come to formally define the tensor species in the next section.

The last operation we want to talk about is evaluation. Where one specifies the exact value of one of the indices of a tensor e.g. η_{0i}. All of the operations disucsed thus far have related to the category RepkG. As such, they have respect the group action. Evaluation, on the other hand, does not respect the group action. For example, taking the 0th component of a four-vector is not Lorentz invariant. Nevertheless, we can define evaluation in the categor of vector spaces. To do this, we need a basis for each $F_D(c)$, the coordinates of a vector in $F_D(c)$ in that basis is discrbed by a linear map of vector spaces from $F_D(c)$ to k. This can be used to evaluate the index of a tensor in the following way:

$$Ff \equiv F_D(fi) \otimes F(f'i) \to \mathbb{1} \otimes F(f'i) \to F(f'i) \tag{8}$$

where f' is f with the ith index removed, and again the first equivalence is somewhat complicated to define but involves extracting the ith index.

2.1.4 Tensor Species

The data of the field k, the group G, the functor F_D (from which F can be derived), the involution τ, the natural transformations for contraction, metric, and unit, and basis needed for evaluation, form formally what we have lossely been calling a Tensor species.

That is, the difference between complex Lorentz tensors, Einstien tensors, and real Lorentz tensors is down to this data. In Lean it useful to work with general tensor species where possible, so important results have to be defined only ones. Thus we make the following definition

```
/-- The sturcture of a type of tensors e.g. Lorentz tensors, Einstien tensors,
  complex Lorentz tensors. -/
structure TensorSpecies where
  /-- The colors of indices e.g. up or down. -/
  C : Type
  /-- 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 field 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
  /-- A ‘MonoidalFunctor‘ from ‘OverColor C‘ giving the rep corresponding to a map
    of colors
    ‘X → C‘. -/
  FDiscrete : Discrete C ⥤ Rep k G
  /-- 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 FDiscrete τ ⟶ 𝟙_ (Discrete C ⥤ Rep k G)
  /-- The natural transformation describing the metric. -/
  metric : 𝟙_ (Discrete C ⥤ Rep k G) ⟶ OverColor.Discrete.pair FDiscrete
  /-- The natural transformation describing the unit. -/
  unit : 𝟙_ (Discrete C ⥤ Rep k G) ⟶ OverColor.Discrete.τPair FDiscrete τ
  /-- 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 (FDiscrete.obj (Discrete.mk c)).V
  /-- Contraction is symmetric with respect to duals. -/
  contr_tmul_symm (c : C) (x : FDiscrete.obj (Discrete.mk c))
      (y : FDiscrete.obj (Discrete.mk (τ c))) :
```

```
  (contr.app (Discrete.mk c)).hom (x ⊗ₜ[k] y) = (contr.app (Discrete.mk (τ c))).hom
    (y ⊗ₜ (FDiscrete.map (Discrete.eqToHom (τ_involution c).symm)).hom x)
/-- Contraction with unit leaves invariant. -/
contr_unit (c : C) (x : FDiscrete.obj (Discrete.mk (c))) :
  (λ_ (FDiscrete.obj (Discrete.mk (c)))).hom.hom
  (((contr.app (Discrete.mk c)) ▷ (FDiscrete.obj (Discrete.mk (c)))).hom
  ((α_ _ _ (FDiscrete.obj (Discrete.mk (c)))).inv.hom
  (x ⊗ₜ[k] (unit.app (Discrete.mk c)).hom (1 : k)))) = x
/-- The unit is symmetric. -/
unit_symm (c : C) :
  ((unit.app (Discrete.mk c)).hom (1 : k)) =
  ((FDiscrete.obj (Discrete.mk (τ (c)))) ◁
    (FDiscrete.map (Discrete.eqToHom (τ_involution c)))).hom
  ((β_ (FDiscrete.obj (Discrete.mk (τ (τ c)))) (FDiscrete.obj (Discrete.mk (τ (c))))).hom.hom
  ((unit.app (Discrete.mk (τ c))).hom (1 : k)))
/-- On contracting metrics we get back the unit. -/
contr_metric (c : C) :
  (β_ (FDiscrete.obj (Discrete.mk c)) (FDiscrete.obj (Discrete.mk (τ c)))).hom.hom
  (((FDiscrete.obj (Discrete.mk c)) ◁ (λ_ (FDiscrete.obj (Discrete.mk (τ c)))).hom).hom
  (((FDiscrete.obj (Discrete.mk c)) ◁ ((contr.app (Discrete.mk c)) ▷
  (FDiscrete.obj (Discrete.mk (τ c))))).hom
  (((FDiscrete.obj (Discrete.mk c)) ◁ (α_ (FDiscrete.obj (Discrete.mk (c)))
    (FDiscrete.obj (Discrete.mk (τ c))) (FDiscrete.obj (Discrete.mk (τ c)))).inv).hom
  ((α_ (FDiscrete.obj (Discrete.mk (c))) (FDiscrete.obj (Discrete.mk (c)))
    (FDiscrete.obj (Discrete.mk (τ c)) ⊗ FDiscrete.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)
```

We can then let e.g. `S : TensorSpecies` and recover the functor F_D by `S.FDiscrete`. The functor F is defined seperatrly can can be recovered by `S.F`. If `f : X \to S.C` is map then a tensor is an element of `S.F.obj (OverColor.mk f)`. The `OverColor.mk` tells Lean to consider f as a function an element of the category $\mathrm{T}^{\times}_{/C}$, or as it is written in Lean `OverColor S.C`.

A function `f : Fin 2 → S.C`, for instance landding on $c1, c2 \in C$ can be written in Lean as `![c1, c2]`, thus we can write `S.F.obj (OverColor.mk ![c1, c2])`.

We will see instances of this for Complex Lorentz tensors. As an example, a Lorentz tensor T^{μ}_{ν} can be written as

```
T : complexLorentzTensor.F.obj (OverColor.mk ![Color.up, Color.down])
```

2.2. TENSOR TREES

A tensor expression consists of a series of tensors and operations between them. For example

$$\eta_{\mu\nu} P^{\nu}_{\sigma} + V_{\mu\sigma}, \tag{9}$$

consists of a product of tensors, a contraction and a addition. Such an expression is a tensor in its own right, and we could just use the operations discussed above to define it.

However, it is useful for bridging the gap between syntax and a tensor to have a more structured way of representing such expressions. This is where tensor trees come in. Tensor trees will also be useful when it comes to proving results about tensors.

A tensor tree is essentially a tree with nodes representing tensors or operations on tensors. For example the tensor tree for the expression above is:

Since we really only care about tensors with $X = Finn$, tensor trees in Lean are implemented only for these.

In Lean we define a tensor tree as follows:

```
/-- A syntax tree for tensor expressions. -/
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 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 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 correpsonding 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 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 correpsonding 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 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)
```

Let us give an example of what this notation means. For each `n : ℕ` and map `c : Fin n → S.C` we get type `TensorTree S c`. This is the type of tensor trees corresponding to tensors in `S.F.obj (OverColor.mk c)`.

The first constructor `tensorNode` generators a tensor tree of type `TensorTree S c` from a tensor of type `S.F.obj (OverColor.mk c)`. The other constructors are slightly more complicated. The constructor `contr` says given an index `i` and `j`, and a tensor tree `t` of type `TensorTree S c`, we can get a tensor tree of type `TensorTree S (c ∘ Fin.succAbove i ∘ Fin.succAbove j)`.

So in Lean the expression above could be written as follows: js: ...

This definition of a tensor tree does not rely the actual nature of the operations invovled. From a tensor tree we can define a tensor itself using a recursively defined map

```
/-- The underlying tensor a tensor tree corresponds to. -/
def tensor : ∀ {n : ℕ} {c : Fin n → S.C}, TensorTree S c → S.F.obj (OverColor.mk
    c) := fun
  | tensorNode t => t
  | add t1 t2 => t1.tensor + t2.tensor
  | perm σ t => (S.F.map σ).hom t.tensor
  | neg t => - t.tensor
  | smul a t => a • 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 Fin.size_pos')) t.tensor
  | action g t => (S.F.obj (OverColor.mk _)).ρ g t.tensor
```

Note that this definition is recursive, Lean can automatically determine that this will terminate (with a little help about the number of nodes in a tensor tree). This association of a tensor tree with a tensor is not one-to-one (but it is onto). Many tensor trees can represent the same tensor, in the same way that many tensor expressions represent the same tensor.

2.2.1 Using Tensor trees in proofs

Tensor trees can be used in proofs for the following reason. Define a sub-tree of a tensor trees to be a node and all child nodes of that node. If T is a tensor tree and S a sub-tree 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.

In Lean this property is encoded in a number of lemmas, for example: js: lemma

We will this at play in the example section of this paper.

2.3. ELABORATION

We now dicuss how we make the Lean code look similar to what we would use on pen-and-paper physics.

This is done using a two step process. Firstly, we define syntax for tensor expressions. Then we write code to turn this syntax into a tensor tree. This process is not formally defined or verified, Lean takes the outputed tensor tree as the formal object to work with.

Instead of deleving into the nitty-gritty details of how this process works under the hood, we give some examples to see how it works.

In what follows we will assume that T, and T_i etc are tensors defined as $S.F.obj\_$ for some tensor species S. We will also assume that these tensors are defined correctly for the expressions below to 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.

We can write e.g.

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

to get the underlying tensor.

Note that we have not lowered or risen the indices, as one would expect from pen-and-paper notation. There is one primary reason for this, whether an index is lower or risen does not carry any information, since this information comes from the tensor itself. Also, for something like complex Lorentz tensors, there are at least three different types of upper-index.

We can extract the tensor from a tensor tree using the following syntax 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)`

For the product of two tensors is also fairly simple, we have

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

The syntax for contraction is as one expect,

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

On the RHS here the first argument (`1`) of `contr` is the index of the first ν on the LHS, 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 then two indices are traying 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 becuase the inner contraction is done 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 of course that the indices are of the correct color (which Lean will check). The elabor 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 here the `_` is a placeholder for the permutation, and in this case will be trivial, but for

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

it will be the permutation for the two indentities.

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

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

Note the use of the `.tensor` to extract the tensor from the tensor tree, it does not really mean much to ask for equalitiy of the tensor trees themselves.

3. EXAMPLES

We give two examples in this section. The first example is a simple theorem involving index notation and tensor trees. We will show here, in rather explicit detail, how we can manipulate tensor trees to solve such theorems. The second example we shall give will show a number of definitions in HepLean concerning index notation. Here we will not give so much detail, the point rather being to show the reader the broad use of our construction.

3.1. EXAMPLE 1: SYMMETRIC AND ANTI-SYMMETRIC TENSOR

If $A^{\mu\nu}$ is an anti-symmetric 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 lemma are written as follows:

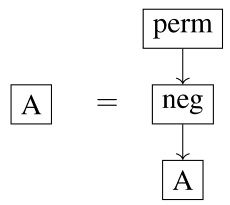
```
lemma antiSymm_contr_symm
    {A : complexLorentzTensor.F.obj (OverColor.mk ![Color.up, Color.up])}
    {S : complexLorentzTensor.F.obj (OverColor.mk ![Color.down, Color.down])}
    (hA : {A | μ ν = - (A | ν μ)}ᵀ) (hs : {S | μ ν = S | ν μ}ᵀ) :
    {A | μ ν ⊗ S | μ ν = - A | μ ν ⊗ S | μ ν}ᵀ := by
  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

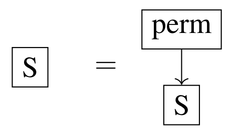
```
{A : complexLorentzTensor.F.obj (OverColor.mk ![Color.up, Color.up])}
{S : complexLorentzTensor.F.obj (OverColor.mk ![Color.down, Color.down])}
```

are simply defining A and S to be tensors of type $A^{\mu\nu}$ and $S_{\mu\nu}$ respectively. This agrees with the notation set out in §?? js: ref.

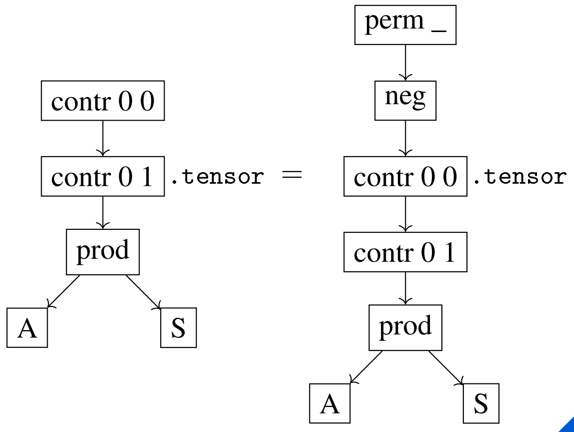
The parameter `hA` is stating that A is anti-symmetric. Expanded in terms of tree diagrams we have

`hA : {A	μ ν = - (A | ν μ)}ᵀ` Description: The tensor A is anti-symmetric.	

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

`hS : {S	μ ν = S	ν μ}ᵀ` Description: The tensor `S` is symmetric.	

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

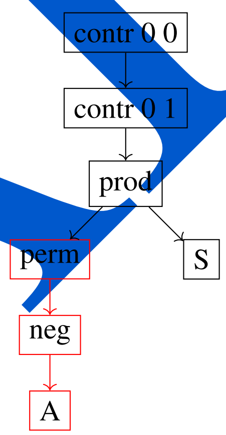


The perm here actually does nothing, but is included by Lean.

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 go through each step in turn.

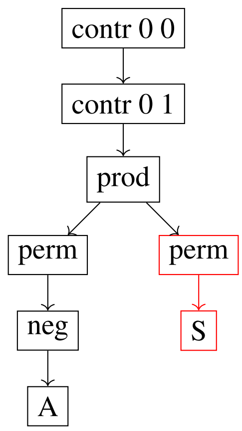
```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_tensor_eq_fst <| hA]
```

Description: Replace the node `A` with the RHS of `hA`.



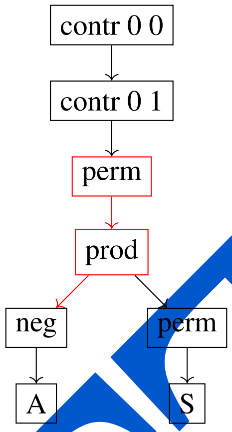

```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_tensor_eq_snd <| hS]
```

Description: Replace the node `S` with the RHS of `hS`.



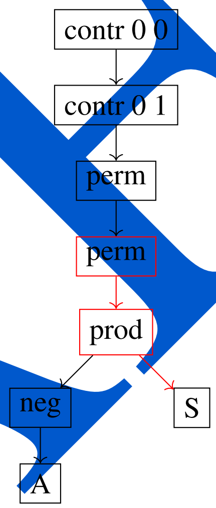
```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_perm_left _ _ _ _]
```

Description: Move the left permutation through the product.



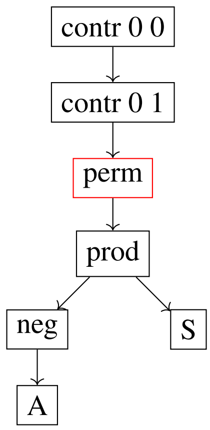

```
rw [contr_tensor_eq <| contr_tensor_eq
<| perm_tensor_eq <| prod_perm_right _ _
_ _]
```

Description: Move the right permutation through the product.



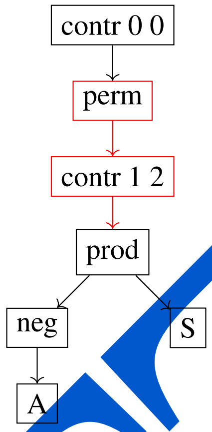

```
rw [contr_tensor_eq <| contr_tensor_eq
<| perm_perm _ _ _]
```

Description: Combine the two permutations (using functoriality).



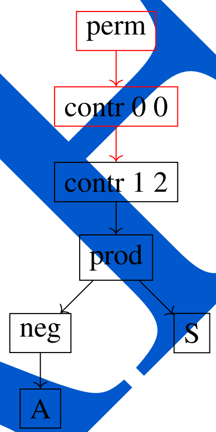
```
rw [contr_tensor_eq <| perm_contr_congr
1 2]
```

Description: Move the permutation through the contraction. And simplify the contraction indices to `1` and `2` (Lean will check if this is correct).



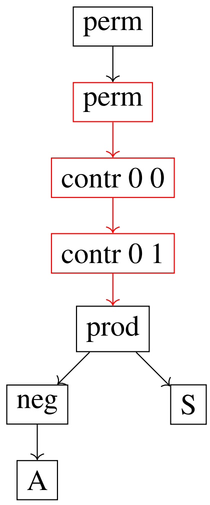

```
rw [perm_contr_congr 0 0]
```

Description: Move the permutation through the contraction. And simplify the contraction indices to `0` and `0` (Lean will check if this is correct).



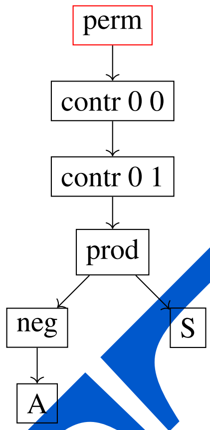

```
rw [perm_tensor_eq <| contr_contr _ _ _]
```

Description: Swap the two contractions. This introduces a permutation.



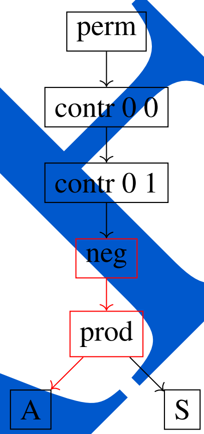
```
rw [perm_perm]
```

Description: Combine the two permutations.



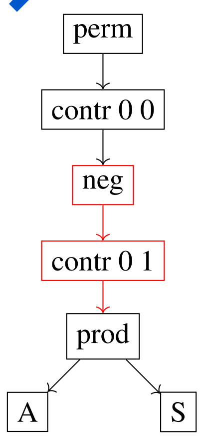

```
rw [perm_tensor_eq <| contr_tensor_eq <|
contr_tensor_eq <| neg_fst_prod _ _]
```

Description: Move the negation through the product.



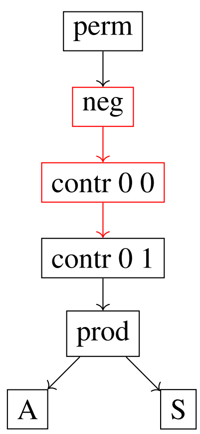

```
rw [perm_tensor_eq <| contr_tensor_eq <|
neg_contr _]
```

Description: Move the negation through the first contraction.




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

Description: Move the negation through the second contraction.



The remainder of the proof `apply perm_congr _ rfl` and `decide` that the tensor trees on the LHS and RHS are actually equal.

3.2. EXAMPLE 2: PAULI MATRICES AND BISPINORS

Using the formlism 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 as using an explcit basis expansion.

In these expressions we have the appearence of metrics `η'` is the metric usually written as $\eta_{\mu\nu}$, js: etc.

With these we can 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 `complexContr` and `complexCo` are complex contravariant and covariant Lorentz vectors. Lean knows to treat these as tensors when they appear in tensor expressions.

Using these definitions we can start to prove results about the pauli matrices and bispinors. These proofs reyl on essentially 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.

Examples of things we have proven range

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

the proof of which is an application of associativity of the tensor product, and appropratly shuffling around of the contractions.

To more complicated results such as

```
/-- 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 •ₜ εL | α α’ ⊗ εR | β β’}ᵀ := by
```

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]
```

The lemmas resource that we hope others—or even automated systems—will formalize in the future.

As demonstrated in our earlier examples, manipulating tensor expressions can involve tedious calculations, especially when dealing with directly with tensor trees. In the 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.

REFERENCES