# Index notation in Lean 4

Joseph Tooby-Smith

*Reykjavik University, Reykjavik, Iceland*

November 6, 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. 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 [1], 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 for correctness using dependent-type theory. The HepLean project is driven by four primary motivations: to facilitate easier look-up of results through a linear storage of information; 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 [2], which aims to formalize mathematics. Indeed, HepLean uses many results from Mathlib, and is built downstream thereof. Other projects in mathematics include the ongoing effort led by Kevin Buzzard to formalize the proof of Fermat's Last Theorem into Lean [3]. In the realm of the sciences, the paper [4] looks at absorption theory, thermodynamics, and kinematics in Lean, whilst the package SciLean [5], 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. As well as, making results from high energy physics easier to write and prove in Lean, it will make the syntax more familiar to high energy physicists. Such an implementation is the subject of this paper. It is a challenge because, we need an implementation that is nice and flexiable to use, but which is also formally well-defined and rigourous. We hope that the implementation presented here will

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

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, in our implementation the result regarding Pauli matrices that $\sigma^{\nu\alpha\dot{\beta}}\sigma_\nu{}_{\alpha'\dot{\beta}'} = 2\varepsilon^{\alpha\alpha'}\varepsilon^{\dot{\beta}\dot{\beta}'}$ 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 the sides of the expression.

Previous implementations of index notation have been made in programming languages like Haskell [6]. 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 theorems and proofs using index notation. The first of these examples involves a lemma regarding the contraction of indices of symmetric and anti-symmetric 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.

## 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. The code snippet above, for Pauli matrices is an example of syntax.

The second representation of a tensor expression is a *tensor tree*. This representaiton 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 intrested 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 the symmetric monoidal category of representations.

Before we discuss these processes in more detail we do some set-up by defining formally a 'tensor species'.

## 2.1. TENSOR SPECIES

A tensor species is a 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 [7, 8], from which our construction is inspired.

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

In Lean a tensor species is defined as follows:

```
/-- The sturcture of a type of tensors e.g. Lorentz tensors, Einstien 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 go through this definition piece by piece. Firstly `k` is a type, has the instance `k_commRing` of a commuative ring. It represents the ring (or field) that our tensors are defined over. For complex Lorentz tensors, this is naturally, the field of complex numbers.

The type `G` has the instance of a group `G_group`. It represents the group which acts on our tensors. For complex Lorentz tensors, this is the group $SL(2, \mathbb{C})$, whilst for real Lorentz tensors this is the Lorentz group.

The type `C` is a type of colors. For complex Lorentz tensors, there are six colors forming the type

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

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 functor `FD`, which 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, so in what follows we will abbreviate this to `FD c`. For complex Lorentz tensors, this is defined as follows:

```
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 defined 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, the representation of left-handed Weyl fermions `Fermion.leftHanded` is defined in Lean as follows:

```
/-- 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.

The map `repDim` assigns to each color a natural number, which corresponds 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 map `basis` gives a basis indexed by `Fin (repDim c)` (numbers from 0 to repDim $c - 1$) of each representation for each $c \in C$. We will use this basis in the definition of evaluation of tensor indices.

To each color the map $\tau$, which is an involution via $\tau$\_involution, defines a dual color. The dual of a color is that which it can be contracted with. So, for complex Lorentz tensors the map $\tau$ 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 is defined by `contr`, which is a natural transformation from the functor `FD _ ⊗ FD (τ _)`, to the functor `𝟙_ (Discrete C ⇒ Rep k G)` 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

$$
\begin{array}{ccc}
\texttt{FD c} \otimes \texttt{FD (τ c)} & \xrightarrow{\quad \beta_- \quad} & \texttt{FD (τ c)} \otimes \texttt{FD c} \\
\downarrow{\scriptstyle \texttt{contr c}} & & \downarrow{\scriptstyle \texttt{𝟙} \otimes \texttt{FD.map (e c)}} \\
\texttt{𝟙\_(Rep k G)} & \xleftarrow{\texttt{contr (τ c)}} & \texttt{FD (τ c)} \otimes \texttt{FD (τ (τ c))}
\end{array}
\tag{1}
$$

where $\beta_-$ is the braiding of the symmetric monoidal category, and `e` is the isomorphism in `Discrete S.C` between `c` and `τ (τ c)`, which exists since $\tau$ is an involution.

Along with the contraction we define 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

$$
\begin{array}{ccc}
\texttt{𝟙\_(Rep k G)} & \xrightarrow{\quad \texttt{unit c} \quad} & \texttt{FD (τ c)} \otimes \texttt{FD c} \\
\downarrow{\scriptstyle \texttt{unit (τ c)}} & & \uparrow{\scriptstyle \texttt{𝟙} \otimes \texttt{FD.map (e c)}} \\
\texttt{FD (τ (τ c))} \otimes \texttt{FD (τ c)} & \xrightarrow{\quad \beta_- \quad} & \texttt{FD (τ c)} \otimes \texttt{FD (τ (τ c))}
\end{array}
\tag{2}
$$

The unit is actually a unit in the sense that contraction with it does nothing. This is made formal with

the condition `contr_unit`, which is corresponds to the diagram

$$
\begin{array}{ccc}
\texttt{FD c} & \xrightarrow{(\rho\_)^{-1}} \texttt{FD c} \otimes \mathbb{1}\_\texttt{(Rep k G)} & \xrightarrow{\mathbb{1}\,\otimes\,\texttt{unit c}} \texttt{FD c} \otimes (\texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD c}) \\
\lambda\_\Big\uparrow & & \Big\downarrow (\alpha\_)^{-1} \\
\mathbb{1}\_\texttt{(Rep k G)} \otimes \texttt{FD c} & \xleftarrow[\texttt{contr c}\,\otimes\,\mathbb{1}]{} (\texttt{FD c} \otimes \texttt{FD (}\tau\texttt{ c))} \otimes \texttt{FD c}
\end{array}
\tag{3}
$$

where $\rho\_$ is the right-unitor, $\lambda\_$ is the left-unitor, and $\alpha\_$ is the associator.

The final part of the definition of a tensor species is the definition of the metric, `metric` and it's interaction with the contraction and unit `contr_metric`. The metric is a natural transformation from the functor $\mathbb{1}\_$ (`Discrete C` $\Rightarrow$ `Rep k G`) to the functor `FD _` $\otimes$ `FD _`. It thus represents the assignement to each color `c` an object of `FD c` $\otimes$ `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

$$
\begin{array}{c}
\mathbb{1}\_\texttt{(Rep k G)} \otimes \mathbb{1}\_\texttt{(Rep k G)} \xleftarrow{(\rho\_)^{-1}} \mathbb{1}\_\texttt{(Rep k G)} \xrightarrow{\texttt{unit c}} \texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD c} \\
\texttt{metric c}\,\otimes\,\texttt{metric (}\tau\texttt{ c)}\Big\downarrow \qquad\qquad\qquad\qquad\qquad\qquad \Big\uparrow\beta\_ \\
(\texttt{FD c} \otimes \texttt{FD c}) \otimes (\texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD (}\tau\texttt{ c))} \qquad\qquad \texttt{FD c} \otimes \texttt{FD (}\tau\texttt{ c)} \\
\alpha\_\Big\downarrow \qquad\qquad\qquad\qquad\qquad\qquad \Big\uparrow \mathbb{1}\,\otimes\,\lambda\_ \\
\texttt{FD c} \otimes (\texttt{FD c} \otimes (\texttt{FD (}\tau\texttt{ c)} \otimes \texttt{FD (}\tau\texttt{ c)))} \qquad \texttt{FD c} \otimes (\mathbb{1}\_\texttt{(Rep k G)} \otimes \texttt{FD (}\tau\texttt{ c))} \\
\mathbb{1}\,\otimes\,(\alpha\_)^{-1}\searrow \qquad\qquad\qquad \nearrow \mathbb{1}\,\otimes\,(\texttt{contr c}\,\otimes\,\mathbb{1}) \\
\texttt{FD c} \otimes ((\texttt{FD c} \otimes \texttt{FD (}\tau\texttt{ c))} \otimes \texttt{FD (}\tau\texttt{ c))}
\end{array}
\tag{4}
$$

## 2.2. TENSORS

Given any type `C`, we define the category `OverColor 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 $\varphi : X \to Y$ such that $f = g \circ \varphi$. This category is equivalent to the core of the category of types sliced over `C`.

The category `OverColor 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 induced map $X \oplus Y \to C$ 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 `FD` can be lifted to a functor to a symmetric monoidal functor from `OverColor S.C` to `Rep k G`. This functor takes $f : X \to S.C$ to the tensor product, $\otimes$ `[k] x`, `S.FD (f x)`, of all `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 ...
```

where Lean calls symmetric monoidal functors `BraidedFunctor`.

We denote the lift of `S.FD` by `F`. Namely, we have

```
def F : 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. We can apply the functor `S.F` to `f : X → S.C` as follows `S.F.obj (OverColor.mk f)`, which we will abbreviate to `S.F f`. This representaiton, `S.F f`, is the tensor product of each of these representations, so that vectors `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`.

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`. When appropriate in 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 convience.

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 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 addition of two tensors. -/
  | add {n : ℕ} {c : Fin n → S.C} : TensorTree S c → TensorTree S c → TensorTree
    S c
  /-- 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 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 a 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 dicuss this before dicussing 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.obj (OverColor.mk c)`, which is defined recursively as follows:

```
/-- 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
  | 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 dicuss each of the constructors in turn, after which we will give some examples tensor trees.

**tensorNode:** The constructor `tensorNode` based on `c` creates a tensor tree from a tensor `t` in `S.F.obj` `(OverColor.mk 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. Diagramatically, such a tree is



Naturally, the tensor associated with this node is exactly the tensor provided during construction.

**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. Diagramatically, such a tree is
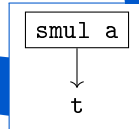


The tensor associated with this new tree is obtained by multiplying the tensor associated with `t` by the scalar `a`.

**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. Diagramatically, we have
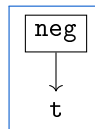


The tensor associated with this new tree is obtained by negating the tensor associated with `t`.

**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. Diagramatically, this corresponds to
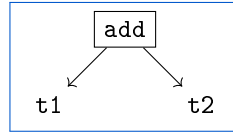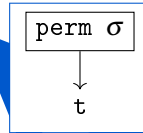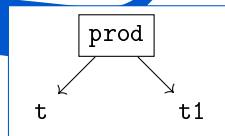
The tensor associated with this new tree is obtained by adding the tensors associated with `t1` and `t2`.

**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. Diagramatically,
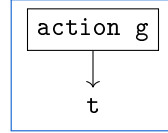


The tensor associated with this new tree is obtained by acting on the tensor associated with `t` with the group element `g`.

**perm:** The constructor `perm` takes a morphism $\sigma$ 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` $\sigma$, with the tensor tree `t` as its child. Diagramatically, this is given by



The tensor associated with this new tree is obtained by applying the image of the morphism $\sigma$ 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 $i = 0,\ldots,n-1$ and via `c1 (i - n)` on $i = n,\ldots,n+n1-1$. This new tree has a root node labeled `prod`, with the tensor trees `t` and `t1` as its children. Diagramatically, this is given by



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.obj (OverColor.mk c) ⊗ S.F.obj (OverColor.mk c1)`, using the tensorate 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.obj (OverColor.mk (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 succesor 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 that 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. Diagramatically,

```
┌───────────┐
│ contr i j │
│     ↓     │
│     t     │
└───────────┘
```

The tensor associated with the new tensor tree is construced as follows. We start with a tensor in `S.F.obj (OverColor.mk c)`. This is mapped into a vector in `(S.FD.obj (c i) ⊗ S.FD.obj (S.τ (c i))) ⊗ S.F.obj (OverColor.mk (c ∘ Fin.succAbove i ∘ Fin.succAbove j))` via an equivalence from `S.F.obj (OverColor.mk c)`. The equivalence is constructed using an equivalence in `OverColor C` to extract `i` and `i.succAbove j` from `c`, then using the tensorate 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.obj (OverColor.mk (c ∘ Fin.succAbove i ∘ Fin.succAbove j))` which is then mapped to a tensor in `S.F.obj (OverColor.mk (c ∘ Fin.succAbove i ∘ Fin.succAbove j))` using the left-unitor of `Rep S.G S.k`.   js: Relate this back to the tensor construction.

**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. Diagramatically, this corresponds to

```
┌───────────┐
│  eval i x │
│     ↓     │
│     t     │
└───────────┘
```
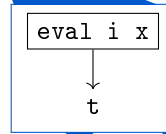
The tensor associated with the new tensor tree is constructed as follows. We start with a tensor in `S.F.obj (OverColor.mk c)`. This is mapped into a vector in `S.FD.obj (c i) ⊗ S.F.obj (OverColor.mk (c ∘ Fin.succAbove i))` via an equivalence from `S.F.obj (OverColor.mk c)`. The equivalence is constructed using an equivalence in `OverColor C` to extract `i` from `c`, and then using the tensorate 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.obj (OverColor.mk (c ∘ Fin.succAbove i))`. This mapping should be thought of as occuring in the category of modules over `S.k`, rather then in `Rep S.G S.k`, since the evaluation is not invariant under the group action. We then fininally map into `S.F.obj (OverColor.mk (c ∘ Fin.succAbove i))` using the left-unitor.

The main reason tensor trees are nice to work with is that they have the following property: 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 manifests in a series of lemmas. For instance, 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 the certain places in tensor trees and replace sub-trees with other sub-trees. We will see this used exensively in the examples.

## 2.4.   SYNTAX AND THEIR MAP TO TENSOR TREES

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

| | |
|---|---|
| $\{T \mid \mu\ \nu\}^{\mathrm{T}}$ | `tensorNode T` |

for a tensor node. Here the $\mu$ and $\nu$ 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 $\{\_\}^{\mathrm{T}}$ notation is used to tell Lean that the syntax is to be treated as a tensor expression.

We can write e.g.

| | |
|---|---|
| $\{T \mid \mu\ \nu\}^{\mathrm{T}}$`.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 $\mu$ or $\nu$ above, for example

| | |
|---|---|
| $\{T \mid 1\ \nu\}^{\mathrm{T}}$ | `eval 0 1 (tensorNode T)` |

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

| | |
|---|---|
| $\{T \mid \mu\ \nu\}^{\mathrm{T}}$ | `neg (tensorNode T)` |

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

| | |
|---|---|
| $\{a \cdot_t T \mid \mu\ \nu\}^{\mathrm{T}}$ | `smul a (tensorNode T)` |

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

| | |
|---|---|
| $\{g \cdot_a T \mid \mu\ \nu\}^{\mathrm{T}}$ | `action g (tensorNode T)` |

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

| | |
|---|---|
| `{T | μ ν ⊗ T2 | σ}`$^\mathrm{T}$ | `prod (tensorNode T) (tensorNode T2)` |

The syntax for contraction is as one expect,

| | |
|---|---|
| `{T | μ ν ⊗ T2 | ν σ}`$^\mathrm{T}$ | `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 $\nu$ 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}`$^\mathrm{T}$ | `(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}`$^\mathrm{T}$ and also `{T | μ ν + T2 | ν μ}`$^\mathrm{T}$, 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}`$^\mathrm{T}$ | `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}`$^\mathrm{T}$ | `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}`$^\mathrm{T}$ | `(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
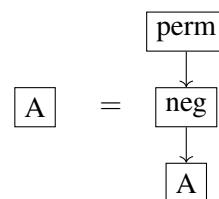


Similarly, the parameter `hs` is stating that $S$ is symmetric. Expanded in terms of tree diagrams
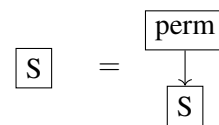


The line `{A | μ ν ⊗ S | μ ν = - A | μ ν ⊗ S | μ ν}`ᵀ is the statment we are trying to prove. In terms of tree diagrams it says that

```
              perm _
                |
               neg
                |
contr 0 0     contr 0 0
   |     .tensor  =  |     .tensor
contr 0 1     contr 0 1
   |             |
  prod          prod
  / \           / \
 A   S         A   S
```
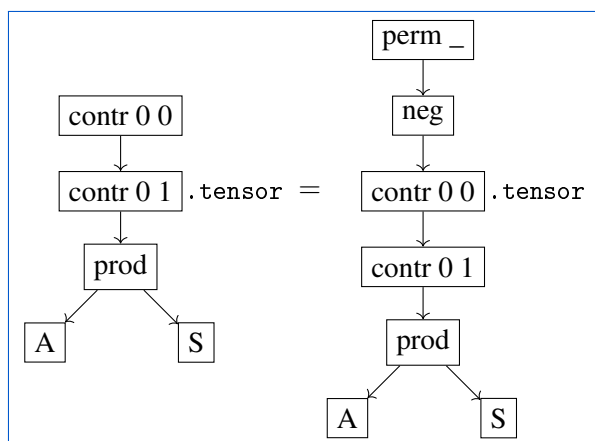
The perm here actually does nothing, but is included by Lean.

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`.

```
contr 0 0
    |
contr 0 1
    |
  prod
  /    \
perm    S
  |
 neg
  |
  A
```
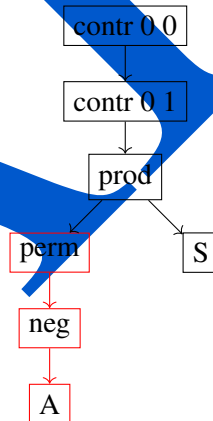
```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_tensor_eq_snd <| hs]
```
―――――――――――――――――

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

```
contr 0 0
    |
contr 0 1
    |
  prod
  /    \
perm   perm
  |      |
 neg     S
  |
  A
```
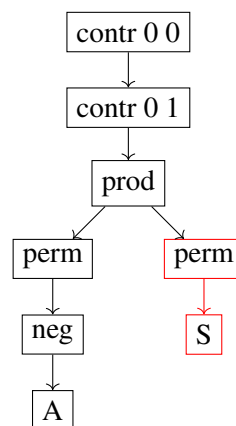
```
rw [contr_tensor_eq <| contr_tensor_eq
<| prod_perm_left _ _ _ _]
```

Description: Move the left permutation through
the product.

```
contr 0 0
   │
contr 0 1
   │
 perm
   │
 prod
  ╱ ╲
neg   perm
 │     │
 A     S
```
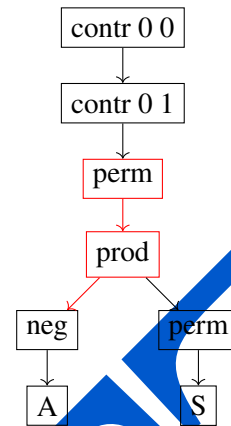
```
rw [contr_tensor_eq <| contr_tensor_eq
<| perm_tensor_eq <| prod_perm_right _ _
_ _]
```

Description: Move the right permutation through
the product.

```
contr 0 0
   │
contr 0 1
   │
 perm
   │
 perm
   │
 prod
  ╱ ╲
neg   S
 │
 A
```
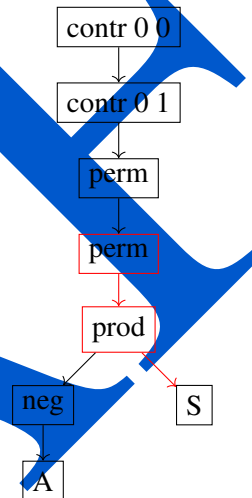
```
rw [contr_tensor_eq <| contr_tensor_eq
<| perm_perm _ _ _]
```

Description:  Combine  the  two  permutations
(using functoriality).

```
contr 0 0
   │
contr 0 1
   │
 perm
   │
 prod
  ╱ ╲
neg   S
 │
 A
```
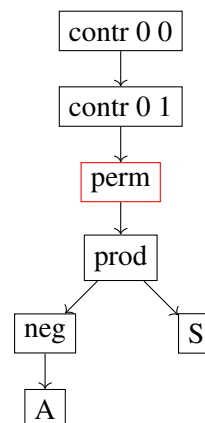
```
rw [contr_tensor_eq <| perm_contr_congr
1 2]
```

---

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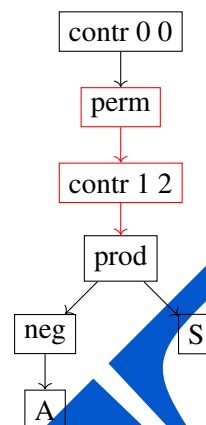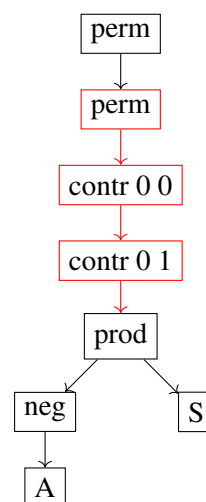


```
rw [perm_contr_congr 0 0]
```

---

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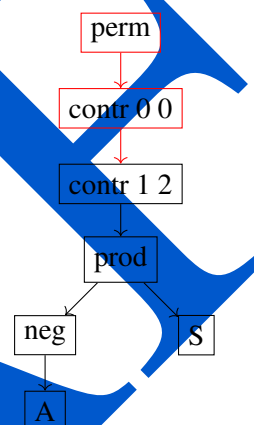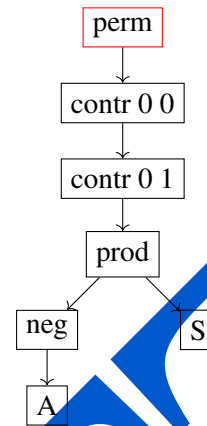


```
rw [perm_tensor_eq <| contr_contr _ _ _]
```

---

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

`rw [perm_perm]`
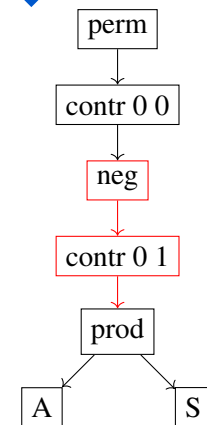
---

Description: Combine the two permutations.



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

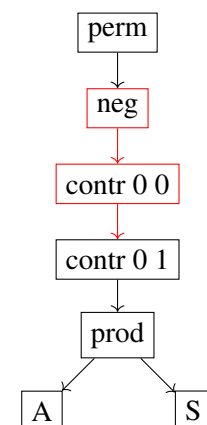---

Description: Move the negation through the product.



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

---
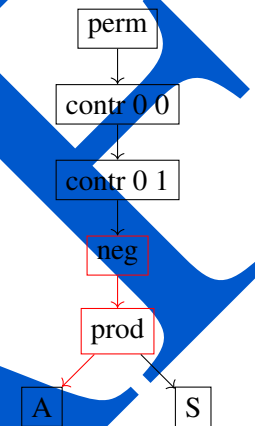
Description: Move the negation through the first contraction.



`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 $\eta$' is the metric usually written as $\eta_{\mu\nu}$, <span style="color:magenta">js: etc.</span>

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_{a}' 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

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

[2]  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:10.1145/3372885.3373824`.

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

[4] 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/D3DD00077J`.

[5] Tomáš Skřivan. Scilean: Scientific computing assistant. *GitHub repository*. URL: `https://github.com/lecopivo/SciLean`.

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

[7] André 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//S1571066111000260`, `doi:10.1016/j.entcs.2011.01.025`.

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