Index notation in Lean 4

Joseph Tooby-Smith
Reykjavik University

October 31, 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.

1. INTRODUCTION

HepLean 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 an interactive theorem prover and programming language with syntax resembling traditional pen-and-paper mathematics. Lean allows users to write definitions, theorems, and proofs which are then automatically checked for correctness, using its foundation of dependent-type theory. HepLean has four main motivations: A linear storage of information makes look-up of results easier; Allows for the creation and proof of new results using automated tactics like AI; Makes it easier to check the correctness of results; And allows for new ways high-energy physics and computer science can be taught.

Other works in Lean 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 from there. 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, js: sorry

Notation in physics and index notation: Physicists heavily rely on specialized notation to express complex 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.

Challange: Formalizing index notation into an interactive theorem prover like Lean is chalanging due to the complexity and implicitness of the notation and the need for flexiable and rigour in it's formalization. This paper presents such an implementation of index notation within the HepLean project.

Motivation: The primary motivations for implementing index notation into Lean are:

1. To make easier to write and prove results from high energy physics into Lean.
2. To make the syntax of Lean more familiar to high energy physicists use to index notation.

We hope that the implementation of index notation into Lean not only enhances usability but also promotes the adoption of formal methods in the physics community.

Conclusion: The conclusion of the implentation of index notation into Lean 4 is that we can write results like the following:

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

and lean will correctly intpret this result as a formal mathematical object which proves can be applied to. js: Mention that we will return to this example later.

Other papers: Previous attempts to formalize 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.

Outline: This paper is split into two main sections. Section js: ref 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

The implementation of index notation in Lean can be broken down into three main components, illustrated in Figure js: ref. 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 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 informal 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 abstract tensor tree into a concrete tensor.

These three steps—syntax, tensor tree, and tensor—are illustrated in Figure js: ref. Although Lean processes information from left to right, 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. This reverse approach is advantageous because the tensors, which are the primary objects of interest, are located on the right side of the diagram. The left and middle parts of the diagram represent 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.

There are six building block representations for complex Lorentz tensors. These are

- `Fermion.leftHanded` is the representation of $SL(2,\mathbb{C})$ corresponding $v \mapsto Mv$, corresponding to Left-handed Weyl fermions.
- `Fermion.altLeftHanded` is the representation of $SL(2,\mathbb{C})$ corresponding $v \mapsto M^{-1T}v$, corresponding to alternative Left-handed Weyl fermions (as we will call them).
- `Fermion.rightHanded` is the representation of $SL(2,\mathbb{C})$ corresponding $v \mapsto M^{*}v$, corresponding Right-handed Weyl fermions.
- `Fermion.altRightHanded` is the representation of $SL(2,\mathbb{C})$ corresponding $v \mapsto M^{-1\dagger}v$, corresponding to alternative Right-handed Weyl fermions.
- `Lorentz.complexContr` is 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
- `Lorentz.complexCo` is 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 `Fermion.leftHanded` is defined in Lean as follows:

```
def leftHanded : Rep ℂ SL(2,ℂ) := Rep.of {
    toFun := fun M => {
      toFun := fun (ψ : LeftHandedModule) =>
        LeftHandedModule.toFin2ℂEquiv.symm (M.1 * ψ.toFin2ℂ),
      map_add' := by
        intro ψ ψ'
        simp [mulVec_add]
      map_smul' := by
        intro r ψ
        simp [mulVec_smul]}
    map_one' := by
      ext i
      simp
    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]}
```

js: Not exact definiton as could not type $*_v$ Here the outer `toFun` argument is defining a map from $SL(2,\mathbb{C})$ to linear maps from `LeftHandedModule` (equivalent to $\mathbb{C}^2$) to itself. The inner `toFun`, `map_add'`

and `map_smul'` are defining defining respectively, the underlying function of this linear map, and proving it it linear with respect to addition and scalar multiplication. The `map_one'` and `map_mul'` are proving that action of the identity is trivial, and that the action of the product of two elements is the product of the actions of the elements.

We assign a unique label, which we refer to as a *color*, to each of the building block representations. We denote the type of colors for a given species of tensors as C. For complex Lorentz tensors C is defined as

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

Here `Color` is the name of our type and `upL`, `downL`, etc. are the colors.

To formally associate each color with its corresponding representation, we define a discrete functor from the set C to the category of k-representations of G, denoted RepkG, that is, a functor

$$F_D : C \Rightarrow \text{Rep}kG. \tag{1}$$

For complex Lorentz tensors this functor in Lean 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
```

The reason why we have used a functor here, rather than just a function, will become apparent in what follows.

2.1.2 A general tensor

For a given tensor species, given the symmetric monoidal structure on RepkG, we can take the tensor product of the building block representations. Elements of such tensor products form the general notion of a tensor for that given species.

To formalize this in Lean, we consider the category $\mathscr{S}^{\times}_{/C}$. The objects of $\mathscr{S}^{\times}_{/C}$ 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 ($\int$) over C, hence our notation.

Any functor $H : C \Rightarrow \text{Rep}kG$ can be lifted to a symmetric monoidal functor $\mathscr{S}^{\times}_{/C} \Rightarrow \text{Rep}kG$ which takes f to the tensor product $\bigotimes_{x \in X} H(f(x))$ and morphisms to the linear maps of representations corresponding to reindexings of tensor products.

This construction is itself functorial, giving a functor:

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

In the previous subsection we defined the functor F_D which associates to each color its corresponding representation. We can then define a functor F by the image of F_D under the lift functor.

A tensor of a given species can then be thought of as a vector in the representations in the of F. Defining a tensor in this way allows us to utalize the structur of monodial categories and functors in a useful way.

In HepLean we formally define lift and F.

For most purposes in physics X will be a type Fin$n$, corresponding to the type (set) of natural numbers (including 0) less n. Later on we will restrict to these types.

2.1.3 Basic operations

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 $\mathscr{S}^{\times}_{/C}$. 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 $f : \text{Fin}\, n.succ.succ \to C$. Here Fin $n.succ.succ$ is the type of all number $0, 1, 2, \ldots, n+1$ less than $n.succ.succ = n+2$. Choosing

two distinct indices $i, j : \mathrm{Fin}\ n.succ.succ$ 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.

$$\begin{array}{ccc} F_D(c) \otimes F(\tau c) & \longrightarrow & F_D(\tau c) \otimes F_D(c) \\ \downarrow & & \downarrow \\ \mathbb{I} & \longrightarrow & F_D(\tau c) \otimes F_D(\tau\tau c) \end{array} \tag{4}$$

- Contraction with the unit does nothing.

$$\begin{array}{ccccc} F_D(c) & \longrightarrow & F_D(c) \otimes \mathbb{I} & \longrightarrow & F_D(c) \otimes (F_D(\tau c) \otimes F_D(c)) \\ \uparrow & & & & \downarrow \\ \mathbb{I} \otimes F_D(c) & & \longleftarrow & & (F_D(c) \otimes F_D(\tau c)) \otimes F_D(c) \end{array} \tag{5}$$

- The unit is symmetric

$$\begin{array}{ccc} \mathbb{I} & \longrightarrow & F_D(\tau c) \otimes F_D(c) \\ \downarrow & & \uparrow \\ F_D(\tau\tau c) \otimes F_D(\tau c) & \longrightarrow & F_D(\tau c) \otimes F_D(\tau\tau c) \end{array} \tag{6}$$

- Contraction with the metric its dual returns the unit

$$\begin{array}{ccccc} \mathbb{I} & & \longrightarrow & & F_D(\tau c) \otimes F_D(c) \\ \downarrow & & & & \uparrow \\ (F_D c \otimes F_d c) \otimes (F_D(\tau c) \otimes F_D(\tau c)) & & & & F_D(c) \otimes F_D(\tau c) \\ \downarrow & & & & \uparrow \\ F_D c \otimes (F_d c \otimes (F_D(\tau c) \otimes F_D(\tau c))) & & \longrightarrow & & F_D c \otimes ((F_d c \otimes F_D(\tau c)) \otimes F_D(\tau c)) \end{array} \tag{7}$$

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. As an example, the condition is written in Lean as

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:

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

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

Then for, e.g. complex Lorentz tensors we define a specific element of this type as follows js: defn of complex Lorentz tensors

js: Discuss here the notation for defining a tensor.

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 briding 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.obj (OverColor.mk _)).ρ g t.tensor
```

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 $\cdot_t$ {T | μ ν}^{T}	smul a (tensorNode T)

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

g $\cdot_a$ {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}	prod (tensorNode T) (tensorNode T2)

js: some missing T's

The syntax for contraction is as one expect,

{T | μ ν ⊗ T2 | ν σ}	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}	(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\} and also \{T | μ ν + T2 | ν μ\}, 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}	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}	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}`	`(tensorNode T).tensor = (perm _ (tensorNode T2)).tensor`

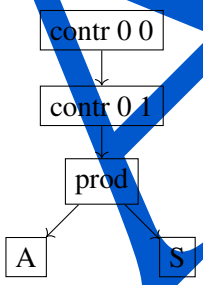
Note the use of the `.tensor` to extract the tensor from the tensor tree, it does not really mean much to ask for equalitiy of the tensor trees themselves.

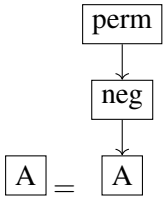
3. EXAMPLES

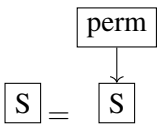
- We give two examples of theorems and proves related to index notation.
- Our first example is related to symmetric and anti-symmetric tensors.
- For this example we will go into explicit detail.
- Our second example is a more detailed one, where we prove that ...
- For this example we will only give a sketch of the prove, and discuss how things are done.

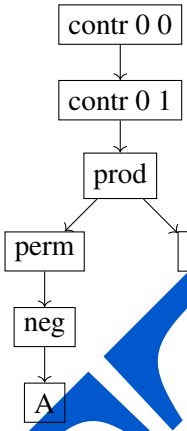
3.1. EXAMPLE 1: SYMMETRIC AND ANTI-SYMMETRIC TENSOR

On the left-hand side we start with the tree:



`hA : {A | μ ν = - (A | ν μ)}`^{T} Description:	

`hS : {S | μ ν = S | ν μ}`^{T} Description:	

`rw [contr_tensor_eq <| contr_tensor_eq <| prod_tensor_eq_fst <| hA]` Description: Replaces the subtree `A` with that specified by the RHS of `hA`.	

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 ‘σμα{dot β}‘. -/
def pauliContr := {PauliMatrix.asConsTensor | ν α β}ᵀ.tensor

/-- The Pauli matrices as the complex Lorentz tensor ‘σ_μα{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 ‘σμ_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 'pμ'. -/
def contrBispinorUp (p : complexContr) :=
  {pauliCo | μ α β ⊗ p | μ}^T.tensor

/-- A bispinor 'p_{aa}' created from a lorentz vector 'pμ'. -/
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
```

- Definition of Pauli matrices.
- Normally would explicity calculate this relation.
- Way we do it is write the pauli matrices in a basis.
- One can define tensor tree relations involving basis.
- For example ....
- One can then use the sorts of thensor manipulations in Section ...
- To prove the relation.

3.3. EXAMPLE 3: DEFINITIONS OF BISPINORS

- We have defined bispinors using this formalism.
- And can use them to prove results.

4. FUTURE WORK

- Informal lemmas and definitions.
  - The shear scale of the task of formalising all results index notation and tensors is too large for any one person.
  - Thus impsired by Lean blueprints we have created a list of informal-definitions and reslults which we hope someone (or something) will formalise in the future.
- Improvement of tactics.
  - As can be seen in the example of ..., there are lots of tedious calculations involved in manipulating the tensor trees.
  - In the future we would like to see a lot of these manipulations automated with sutiable tactics. We hope the tensor tree structure will lend itself nicely to this.
- There are two main ways we can take the concepts here further.
- We could define spinor-hleicity formalism, or extend things to tensor fields and derivatives thereof.
- We do not beleive there to be any unsumountable chanangle in these tasks.

REFERENCES