*JFP* **35**, e9, 42 pages, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.
doi:10.1017/S0956796825000048

# *Domain-specific tensor languages*

JEAN-PHILIPPE BERNARDY
*Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden*
(*e-mail:* `jean-philippe.bernardy@gu.se`)

PATRIK JANSSON
*Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden*
(*e-mail:* `patrikj@chalmers.se`)

## Abstract

The tensor notation used in several areas of mathematics is a useful one, but it is not widely available to the functional programming community. In a practical sense, the (embedded) domain-specific languages (DSLs) that are currently in use for tensor algebra are either 1. array-oriented languages that do not enforce or take advantage of tensor properties and algebraic structure or 2. follow the categorical structure of tensors but require the programmer to manipulate tensors in an unwieldy point-free notation. A deeper issue is that for tensor calculus, the dominant pedagogical paradigm assumes an audience which is either comfortable with notational liberties which programmers cannot afford, or focus on the applied mathematics of tensors, largely leaving their linguistic aspects (behaviour of variable binding, syntax and semantics, etc.) for the reader to figure out by themselves. This state of affairs is hardly surprising, because, as we highlight, several properties of standard tensor notation are somewhat exotic from the perspective of lambda calculi. We bridge the gap by defining a DSL, embedded in Haskell, whose syntax closely captures the index notation for tensors in wide use in the literature. The semantics of this EDSL is defined in terms of the algebraic structures which define tensors in their full generality. This way, we believe that our EDSL can be used both as a tool for scientific computing, but also as a vehicle to express and present the theory and applications of tensors.

## 1 Introduction and motivation

Tensor calculus is an essential tool in physics and applied mathematics. It was instrumental already a century ago in the formulation of Einstein's general relativity, and its usage has spread to many areas of science. At its heart lies linear algebra, which is defined as the study of linear maps between vector spaces. In applications, one commonly manipulates *representations* of linear algebraic objects: vectors as 1-dimensional arrays and linear maps as matrices (2-dimensional arrays). Indeed, assuming a given basis, the representations are equivalent to the algebraic objects. Likewise, tensors are often thought of as a higher-dimensional version of matrices: their algebraic formulation is as a category of linear maps between vector spaces.

Viewing the above situation through the lens of programming language theory, the algebraic formulation forms a set of combinators and the array-based representations is a possible semantics for them. Even though the praxis is to blur the distinction between algebraic objects and their coefficient representations, it is a source of confusion in the case of tensor calculus, which studies tensor fields, in the sense of tensor-valued functions defined over a manifold. (We provide some evidence in Section 8.1.) Notably, difficulties arise because the basis varies over the manifold. The first contribution of this paper is to provide a clear conceptual picture by highlighting the syntax-semantics distinction.

On the practical side, the situation is similar. One can find a plethora of languages and libraries purportedly geared towards tensor manipulation, but they inevitably focus on their multi-dimensional array representations. There is nearly no support for algebraic tensor field expressions. In this paper, we work towards bridging this gap, by applying programming-language methodology to the notations of tensor algebra and tensor calculus—thus viewing them as domain-specific languages. For the readership with a programming language background, we aim to provide a down-to-earth presentation of tensor notations. We capture all their important properties, in particular by making use of linear types. We also aim to attract a readership that already has a working knowledge of tensors. For them we aim to fully formalise the relationship between the representation-oriented notation for tensor fields and its linear-algebraic semantics. We do so by viewing this syntax as terms in a (linear-typed) lambda calculus. As usual with DSLs, this presentation comes with an executable semantics. This means we end up with a usable tool to manipulate tensor fields, which is the second contribution of this paper.

## *1.1 Overview*

To make the presentation more pedagogical, we delay the introduction of tensor fields over manifolds until Section 5. Until then, the reader can think of each tensor as "just" an element of a certain vector space. This allows us to present the core concepts in a simpler setting, even though they will apply unchanged in the more general context. As hinted above, we will use an algebraic semantics for tensors, following a categorical structure (Section 3). Together, the combinators forming this categorical structure form a point-free EDSL, which we refer to as ROGER in reference to Roger Penrose (see Section 8.5 as for why).

Every ROGER program can be evaluated to morphisms in any suitable tensor category. This includes matrices, but also string diagrams with the appropriate structure as well. ROGER is useful in its own right, but has all the downsides of a point-free language, and thus is not in wide use in the mathematics community, where the so-called *Einstein notation* is preferred. The Einstein notation mimics the usual notation to access components of matrices, but speaks about these components in a wholesale manner, that is, with index variables that range over all the dimensions. We formalise this notation in an index-based EDSL (Section 4). We refer to this EDSL as ALBERT in the rest of the paper. Expressions in ALBERT evaluate to morphisms in ROGER, and thus in any tensor category.

In sum, because the index-notation, diagram notation and matrices are instances of tensor categories, programs written in any of our EDSLs can be executed as tensor programs using the matrix instance or can generate index or diagram notation for the code in question. The relationships between these notations and EDSLs are depicted in Figure 1.

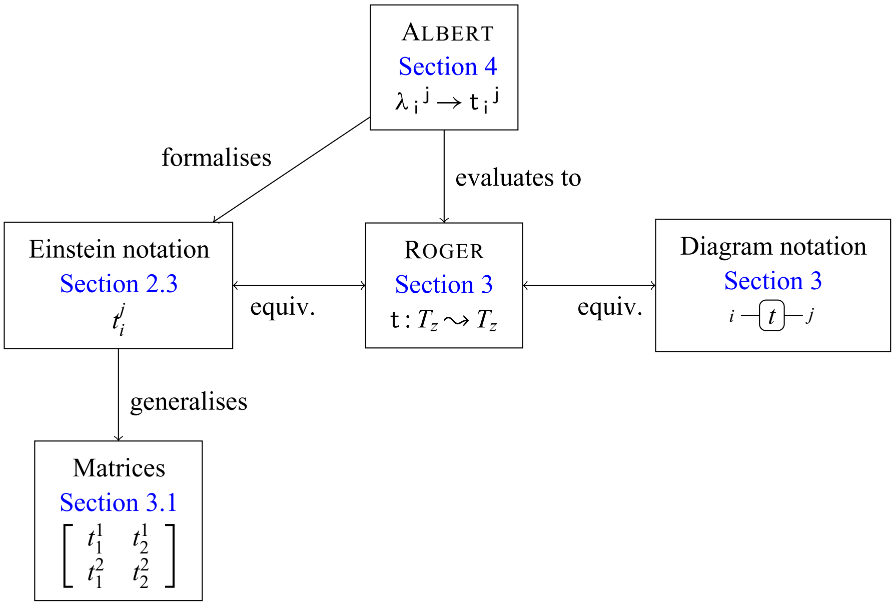
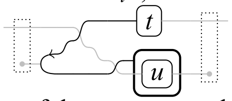


Fig. 1: Tensor notations, EDSLS and relationships between them. Even though the index notation, the morphism notation and the string diagram notation are all equivalent mathematically, in our implementation ROGER is coded as a (set of) type-classes, and the index and diagram notations are instances of it.

This means that a function in ALBERT, say

$$\text{example}\ t\ u = \text{contract}\ (\lambda\,{}^i{}_i \to t\,{}^i \star \text{deriv}\,{}_i\ u)$$

will, depending on the type, either:

1. render itself in Einstein notation as $t^i \nabla_i u$;
2. render itself as the diagram  or
3. run on matrix representations of the tensors $t$ and $u$ and compute the result (a scalar field in this case, representing the directional derivative of $u$ in the direction of $t$).

Together, ALBERT and ROGER form a Haskell library for expressing tensors.[1] This library leverages linear types as implemented in GHC 9. This implementation defines an executable semantics of ALBERT, and is presented in Section 7. All the examples presented in this paper were prepared using our library. In particular, the diagrams are generated with it.

In Section 5, we move to deal with tensor fields proper. Essentially this means that every expression in either EDSL corresponds to a tensor field, and that we can manipulate derivatives of such fields. With this addition, the EDSLS can be used for symbolic calculations of tensor fields. We can, for example, apply covariant derivatives to tensor expressions, re-express them in terms of partial derivatives and Christoffel symbols,

[1] The code is available at `https://github.com/jyp/linear-smc/`.

and instantiate those to concrete coordinates systems. We demonstrate this workflow in Section 6, where we express Einstein's General Relativity equation for the curvature of space-time and verify that the Schwarzschild metric tensor is a solution.

We start in Section 2 with a summary of the notions of linear algebra and tensors.

## 2 Background: linear algebra and tensors

The goal of this section is both to provide the canonical presentation as reference and to expose its abstruse character. The summary does not replace a proper introduction to the topic, and we urge the reader to turn to an appropriate reference if necessary (see references in Section 8.1).

A typical definition of tensor that one might find is the following:

*An nth-rank*[2] *tensor in m-dimensional space is a mathematical object that has n indices and $m^n$ components and obeys certain transformation rules.* (Rowland & Weisstein, 2023)

(The transformations in question relate to change of basis, as we will see.) This kind of definition is heavily geared towards coordinate representations, rather than their algebraic definition. Why do pedagogical accounts widely refer to coordinate representations rather than semantics? One answer is that calculations are eventually always performed using coordinates. Another answer is that the kind of algebraic thinking required to grasp tensors may be too abstract to form an intuition. Our point of view is that it is indeed at the wrong abstraction level, and that the categorical structures are better suited to reasoning about tensors than the pure linear-algebraic ones. Nonetheless, we will have to refer to the algebraic definitions of tensors down the road, so we provide a minimal recap below.

### 2.1 Pure algebraic point of view

The main object of study are homomorphisms between vector spaces: linear transformations, also called linear maps. We will later see that tensors are such maps.

**Definition 1** (vector space). *A vector space (over a field S) is a commutative group* v *equipped with a compatible notion of scaling by elements of S.*

```
class Group v where
  (+)    :: v → v → v      class (Group v) ⇒ VectorSpace v where
  0      :: v                (◃) :: S → v → v
  negate :: v → v
```

*A vector space must additionally satisfy a number of laws, including that scaling is a linear operation:* $s \triangleleft (x+y) = s \triangleleft x + s \triangleleft y$.

[2] What is called here *rank* is referred to as *order* in our text. We choose this terminology because rank has another meaning in linear algebra. Namely, it is the minimum number of simple tensors that sum to it. A simple tensor is a tensor product of a number of non-zero vectors and co-vectors: $\mathsf{t} = \mathsf{v}_1 \otimes ... \otimes \mathsf{v}_n$. Most proper tensors are not simple. That is, their rank is more than one.

The exact nature of this field of scalars ($S$) has little bearing on the algebraic development[3], but we assume throughout that they are real numbers. Note that $S$ is itself a vector space, with scaling ($\triangleleft$) then being scalar multiplication.

**Definition 2** (linear map). *A function $f : V \longrightarrow W$ is a linear map iff. for all collections of scalars $c_i$ and vectors $\vec{v}_i$ we have*

$$f\left(\sum_i c_i \triangleleft \vec{v}_i\right) = \sum_i c_i \triangleleft (f(\vec{v}_i))$$

For a fixed domain and codomain, linear maps themselves form a vector space.

```
instance (Group w) ⇒ Group (v ⟶ w) where
  negate f = (λ v → negate (f v))
  f + f′ = (λ v → f v + f′ v)
  0 = (λ v → 0)
instance (VectorSpace v, VectorSpace w) ⇒ VectorSpace (v ⟶ w) where
  c ◃ f = (λ v → c ◃ f v)
```

The eager reader should be warned that, for now, indices are used to range of over arbitrary sets of vectors and scalars (and bound by $\sum$), in a usual way. Indices take a special meaning only when we get to coordinates and the Einstein notation (from Sections 2.2 and 2.3).

**Definition 3** (covector space). *Given a vector space $V$, the covector space $V^*$ is defined as the set of linear maps $V \longrightarrow S$.*

Since covector spaces are special cases of linear maps, they form vector spaces too. In a similar vein, the set of linear maps $f : S \longrightarrow W$ is isomorphic to $W$. (Indeed $f(s) = f(s \triangleleft 1) = s \triangleleft f(1)$, and thus the vector $f(1)$ in $W$ fully determines the linear function $f$.)

**Definition 4** (bilinear map). *A function $f : V \times W \longrightarrow U$ is a bilinear map iff. for all $c_i, d_j : S$, $\vec{v}_i : V$, and $\vec{w}_j : W$ we have*

$$f\left(\sum_i c_i \triangleleft \vec{v}_i, \sum_j d_j \triangleleft \vec{w}_j\right) = \sum_{i,j} c_i d_j \triangleleft (f(\vec{v}_i, \vec{w}_j))$$

**Definition 5** (Tensor product of vector spaces). *Given two vector spaces $V$ and $W$, their tensor product is a vector space, denoted by $V \otimes W$, together with a bilinear map $\phi : (V \times W) \longrightarrow (V \otimes W)$ with the following universal property. For every vector space $Z$ and every bilinear map $h : (V \times W) \longrightarrow Z$, there exists a unique linear map $h' : (V \otimes W) \longrightarrow Z$ such that $h = h' \circ \phi$. The output $\phi(v, w)$ is often denoted by $v \otimes w$,*

[3] Symmetrisation and antisymmetrisation (Section 6) require the field to have characteristic zero, which is true for real and complex fields.

*overloading the same symbol. (We let the reader check that the tensor product always exists.)*

**Examples:** Here is an attempt at providing an intuition for what is, and is not, a bilinear function. Consider the simplest case of the definition of bilinear map where there is just one vector $\vec{v}$ as the first argument and one vector $\vec{w}$ as the second argument to $f$. We then have $f(\vec{v}, 0) = f(1 \triangleleft \vec{v}, 0 \triangleleft \vec{w}) = (1 \times 0) \triangleleft f(\vec{v}, \vec{w}) = 0$. This means that vector addition is *not* bilinear because $\vec{v} + 0 = \vec{v} \neq 0$. Similarly, $f$ cannot be first or second projection, because they are also linear, not bilinear.

We also have that we can "move constant factors" between $\vec{v}$ and $\vec{w}$: $f(c \triangleleft \vec{v}, 1 \triangleleft \vec{w}) = (c \times 1) \triangleleft f(\vec{v}, \vec{w}) = (1 \times c) \triangleleft f(\vec{v}, \vec{w}) = f(1 \triangleleft \vec{v}, c \triangleleft \vec{w})$. In connection with the tensor product this means that even though, for any two vectors $\vec{v} : V$ and $\vec{w} : W$, we can construct a tensor $u = \phi(\vec{v}, \vec{w}) : V \otimes W$ which looks like we have embedded a pair, we cannot extract $\vec{v}$ and $\vec{w}$ again—they are mixed up together (entangled).

What a bilinear function can (and must) do, as we can see from the definition, when given two linear combinations, is to compute a linear combination based on all pairwise products of the coefficients, without depending on the coefficients themselves.

**Order of a tensor** Often, tensors are used in a context where there is a single (atomic) underlying vector space $\mathsf{T}$ which is not just the scalars. Then the complexity of a vector space built from $\mathsf{T}$ can be measured by its order. The order of $\mathsf{T}$ is defined to be 1 and the order of the scalar space is 0. The order of a tensor space $\mathsf{V} \otimes \mathsf{W}$ is the sum of the order of spaces $\mathsf{V}$ and $\mathsf{W}$, and this way we can build spaces of arbitrarily large order. The order of a linear map can be defined either as the pair of the orders of its input and output spaces, or as their sum (depending on convention). For example, a linear operator on an atomic vector space has order (1,1) or 2 in the respective conventions. Morphisms of order three or more are properly called tensors. Conversely, tensors of any order (including 0, 1 and 2) are linear maps, of the appropriate domain and codomain. When there is more that one underlying vector space, the order is not enough to characterise a tensor space: the full type needs to be specified, as in Section 3. (Yet this level of complexity won't be exercised in this paper.)

### *2.2 Coordinate representations*

In practice, the algebraic definitions are not easy to manipulate for concrete problems, thus one most commonly works with coordinate representations instead. (Our goal will be to break free of those eventually.) As a reminder, given a basis $\vec{e}_i$, any vector $\vec{x} \in V$ can be uniquely expressed as $\vec{x} = \sum_i x^i \triangleleft \vec{e}_i$ where each $x^i$ coordinate is a scalar. In this way, given a basis $\vec{e}_i$, a vector space is isomorphic to its set of coordinate representations. Note that a superscript is used for the index of such coordinates. The general convention that governs whether one should write indices in low or high positions is explained in Section 2.3; for now, it is enough to know that they are indexing notations.

Like vectors, linear maps are also commonly manipulated as matrices of coefficients. For a linear map $f$ from a vector space with basis $\vec{d}_i$ to a space with basis $\vec{e}_j$, each column is given by the coefficients of $f(\vec{d}_i)$. Indeed, using $F_i{}^j$ to denote the coefficients, we have:

$$f(\vec{x}) = f\left(\sum_i x^i \triangleleft \vec{d}_i\right) = \sum_i x^i \triangleleft f(\vec{d}_i) = \sum_i x^i \triangleleft \left(\sum_j F_i{}^j \triangleleft \vec{e}_j\right) = \sum_j \left(\sum_i F_i{}^j x^i\right) \triangleleft \vec{e}_j$$

In general, the values of the matrix coefficients $F_i{}^j$ depend on the choice of bases $\vec{d}_i$ and $\vec{e}_j$, but to reduce the number of moving parts one usually works with a coherent set of bases.

**Coherent bases** Starting from an atomic vector space $T$, one can build a collection of more complicated tensor spaces using tensor product, dual, and the unit (the scalar field $S$). For coordinate representations each such space could, in general, have its own basis, but it is standard to work with a collection of coherent bases. Given a basis $\vec{e}_i$ for a finite-dimensional atomic vector space $T$, the coherent basis for $T^*$ is the set of covectors $\tilde{e}^j$ such that $\tilde{e}^j(\vec{e}_i) = \delta_i^j$. (It is usually called the dual basis.) Likewise, given two coherent bases $\vec{d}_i$ and $\vec{e}_i$ respectively for $V$ and $W$, the coherent basis for $V \otimes W$ is $b_{i,j} = \phi(\vec{d}_i, \vec{e}_j)$, where $\phi$ is given by Definition 5. Note that this basis is indexed by a pair. Accordingly, if the dimension of $V$ is $m$ and the dimension of $W$ is $n$, the dimension of $V \otimes W$ is $m \times n$. Additionally, re-associating tensor spaces do not change coherent bases ($\vec{e}_{(i,j),k}$ is the same as $\vec{d}_{i,(j,k)}$, up to applying the corresponding associator). Finally, the scalar vector space has dimension one, and thus has a single base vector, which is coherently chosen to be the unit of the scalar field (the number $1 : S$).

**Coordinate transformations** Assuming one basis $\vec{e}_j$ and another basis $\vec{d}_i$ for the same vector space $V$ such that $\vec{d}_i = \sum_j F_i{}^j \vec{e}_j$, then the coordinates in basis $\vec{d}_i$ for $\vec{x}$ are $\hat{x}^j = \sum_i F_i{}^j x^i$. We say that the matrix $F$ is the transformation matrix for $V$ given the choice of bases made above, and denote it $J(V)$. Then the transformation matrices for vector spaces built from an atomic space $T$ using the coherent set of bases defined above are given by the following structural rules:

| | |
|---|---|
| $J(V \otimes W) = J(V) \otimes J(W)$ | Kronecker product of matrices |
| $J(V^*) = J(V)^{-1}$ | matrix inverse |
| $J(S) = 1$ | scalar unit |

Furthermore, the matrix representation $G$ of a linear map $g : V \longrightarrow W$ is transformed to: $J(W) \cdot G \cdot J(V^*)$, where $(\cdot)$ is matrix multiplication. These are the "transformation rules" that Rowland & Weisstein (2023) allude to in the above quote.

## *2.3 Einstein notation*

The previous section showed how to deal with concrete matrices, using a concrete choice of bases. The next step is to manipulate symbolic expressions involving matrices. The language of such expressions (together with a couple of simple conventions) is colloquially referred to as Einstein notation.

In this notation, every index ranges over the dimensions of an atomic vector space.[4] Consequently, the total number of free (non repeated) indices indicates the order of a tensor expression in index notation. An index can be written as a subscript (and called a low index) or as a superscript (and called a high index).

The location (high or low) of the index is dictated by which coordinate transformation applies to it. That is, if a high index ranges over the dimensions of $V$, then $J(V)$ applies, whereas $J(V^*)$ applies for a low index. Additionally, every reference to a tensor is fully saturated, in the sense that a symbolic tensor is always applied to as many indices as its order. Thus, for instance, $x^i$ denote (the components of) a vector, and $y_j$ denote (the components of) a covector. The expression $t_i{}^j$ refers to (components of) a linear transformation of order (1,1). In the absence of contraction (see below), multiplication increases the order of tensors. For instance, $x^i\, y_j$ also has order (1,1). In general, if $t$ and $u$ are expressions denoting tensors of order $m$ and $n$, respectively, then their product $t\, u$ denotes a tensor of order $m + n$.

**Contraction** In Einstein notation, the convention is that, within a term, a repeated index is implicitly summed over. (In terms familiar to this journal: such indices are implicitly bound by a summation operator.) Because summation is a linear operator, within a term all the well-scoped locations of the summation operator are equivalent—so it makes a lot of sense to omit them. Additionally, when an index is repeated, it must be repeated exactly twice; once as a high index and once as a low index. Mentioning an index twice is called *contraction*. Viewing tensors as higher dimensional matrices of coefficients, contraction consists in summing coefficients along a diagonal. Therefore, a contraction reduces the order of the tensor by two.[5] The indices which are contracted are sometimes called "dummy" and those that are not contracted are called "live" (In terms familiar to the functional programming community, dummies are bound variables and live indices are free variables.) To be well-scoped, every term in a sum must use the same live indices. For instance, the expression $t_l{}^j u_m{}^k v_i{}^{lm} + v_i{}^{jk}$ denotes a tensor of order (1,2). Its live indices are ${}_i$, ${}^j$, ${}^k$, and the indices $m$ and $l$ are dummies.

At this stage, one can see the Einstein notation as a convenient way to notate expressions which manipulate coordinates of tensors. The high/low index convention makes it clear which transformations apply. Yet it may be mysterious why indices must be repeated exactly twice, and why (live) indices cannot be omitted from a term. The answer lies in the following observation. Even though the Einstein notation may originate as a convenient way to express coordinates, it really is intended to describe algebraic objects. The physicists Thorne & Blandford (2015) put it this way:

> *[we suggest to] momentarily think of [Einstein notation] as a relationship between components of tensors in a specific basis; then do a quick mind-flip and regard it quite differently, as a relationship between geometric, basis-independent tensors with the indices playing the roles of slot names.*

[4] That is, the components of tensor spaces are always indexed separately; and the tensor unit (scalar) is never indexed. Previously, indices were used to range over arbitrary sets of vectors and scalars—without regard for the dimensions of the spaces which they inhabit.

[5] As such, it is the generalisation of a trace operation. We come back to the connection between contraction and trace in Section 3.

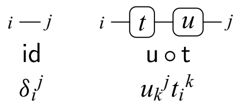


Fig. 2: Diagram, categorical and index notations for identity and composition.

(A "slot" is a component of input or output tensor space.) The key of this "mind flip" is that live indices correspond to inputs (or outputs) of linear functions, and contraction corresponds to connecting inputs to outputs. The main contribution of this paper is to work out this connection in full as a pair of two EDSLs.

## 3 Categorical structures

The key concepts needed to understand the essence of Einstein notation are the categorical structures that tensors inhabit. Besides, these structures will be instrumental in our design: we will let the user of ALBERT write expressions which are (close to) Einstein notation, but they will be evaluated to morphisms in the appropriate category. The underlying category can then be specialised according to the application at hand.

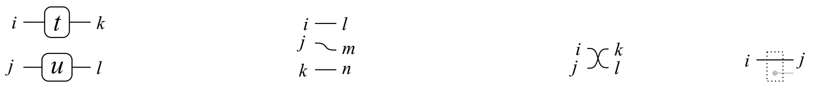
The categorical approach consists in raising the abstraction level, and focusing on the ways that linear maps are combined to construct more complex ones. The first step is to view linear maps as morphisms of a category whose objects are vector spaces. We render the type of morphisms from a to b as $\mathsf{a} \overset{z}{\leadsto} \mathsf{b}$, corresponding to $z$ a b in Haskell code.

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Category}\ z\ \textbf{where} \\
\quad \mathsf{id}\ \ :: \mathsf{a} \overset{z}{\leadsto} \mathsf{a} \\
\quad (\circ) :: (\mathsf{b} \overset{z}{\leadsto} \mathsf{c}) \to (\mathsf{a} \overset{z}{\leadsto} \mathsf{b}) \to (\mathsf{a} \overset{z}{\leadsto} \mathsf{c})
\end{array}
$$

Vector spaces form a commutative monoid under tensor product. Hence, linear maps form a symmetric monoidal category, or SMC, whose combinators are as follows.

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Category}\ z \Rightarrow \mathsf{SymmetricMonoidal}\ z\ \textbf{where} \\
\quad (\otimes) :: (\mathsf{a} \overset{z}{\leadsto} \mathsf{b}) \to (\mathsf{c} \overset{z}{\leadsto} \mathsf{d}) \to (\mathsf{a} \otimes \mathsf{c}) \overset{z}{\leadsto} (\mathsf{b} \otimes \mathsf{d}) \\
\quad \sigma \quad :: (\mathsf{a} \otimes \mathsf{b}) \overset{z}{\leadsto} (\mathsf{b} \otimes \mathsf{a}) \\
\quad \alpha \quad :: ((\mathsf{a} \otimes \mathsf{b}) \otimes \mathsf{c}) \overset{z}{\leadsto} (\mathsf{a} \otimes (\mathsf{b} \otimes \mathsf{c})) \\
\quad \bar{\alpha} \quad :: (\mathsf{a} \otimes (\mathsf{b} \otimes \mathsf{c})) \overset{z}{\leadsto} ((\mathsf{a} \otimes \mathsf{b}) \otimes \mathsf{c}) \\
\quad \rho \quad :: \mathsf{a} \overset{z}{\leadsto} (\mathsf{a} \otimes \mathbf{1}) \\
\quad \bar{\rho} \quad :: (\mathsf{a} \otimes \mathbf{1}) \overset{z}{\leadsto} \mathsf{a}
\end{array}
$$

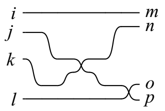
In the above SMC class definition, we follow the usual convention of using the same symbol ($\otimes$) both for the product of objects and the parallel composition of morphisms. In fact, this morphism operator is also called a tensor product in the literature. An SMC comes with a number of laws, which are both unsurprising and extensively documented elsewhere (Barr



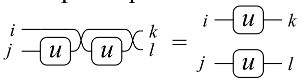
| | | | |
|---|---|---|---|
| $\mathsf{t} \otimes \mathsf{u}$ | $\alpha$ | $\sigma$ | $\rho$ |
| $t_i{}^k u_j{}^l$ | $\delta_i{}^l \delta_j{}^m \delta_k{}^n$ | $\delta_i{}^l \delta_j{}^k$ | $\delta_i{}^j$ |
| $\mathsf{T} \otimes \mathsf{T} \overset{z}{\rightsquigarrow} \mathsf{T} \otimes \mathsf{T}$ | $(\mathsf{T} \otimes \mathsf{T}) \otimes \mathsf{T} \overset{z}{\rightsquigarrow} \mathsf{T} \otimes (\mathsf{T} \otimes \mathsf{T})$ | $\mathsf{T} \otimes \mathsf{T} \overset{z}{\rightsquigarrow} \mathsf{T} \otimes \mathsf{T}$ | $\mathsf{T} \overset{z}{\rightsquigarrow} \mathsf{T} \otimes \mathbf{1}$ |

Fig. 3: Diagram, categorical, and Einstein notations for morphisms of symmetric monoidal categories. They are in general polymorphic, but we display them here as acting on an atomic vector space $T$, or the simplest allowable combination thereof (see the last row in the figure for the monomorphic type of the respective morphisms). The morphisms $\bar{\alpha}$ and $\bar{\rho}$ are not shown, but are drawn symmetrically to $\alpha$ and $\rho$, respectively.

& Wells, 1999). We omit them here. The operations $\sigma$ (swap), $\alpha$, $\bar{\alpha}$ (associators) witness the commutative monoidal structure which tensor products possess. The unit of the tensor product, written $\mathbf{1}$, is the scalar vector space (S), which is witnessed by the isomorphisms $\rho$ and $\bar{\rho}$, called unitors.

As an example, take the morphism $\mathsf{ex} = (\mathsf{id} \otimes \sigma) \circ \bar{\alpha} \circ (\mathsf{id} \otimes \alpha \circ (\sigma \otimes \mathsf{id}) \circ \bar{\alpha}) \circ \alpha \circ \alpha$. It is polymorphic, but has in particular type $((\mathsf{T} \otimes \mathsf{T}) \otimes \mathsf{T}) \otimes \mathsf{T} \overset{z}{\rightsquigarrow} (\mathsf{T} \otimes \mathsf{T}) \otimes (\mathsf{T} \otimes \mathsf{T})$. Its input and output orders are both 4, for a total order of (4,4) or 8. It is written $\delta_i{}^m \delta_j{}^p \delta_k{}^n \delta_l{}^o$ in Einstein notation, which makes more explicit the connection between inputs and output. An even more explicit notation is its rendering as a diagram:  .

This diagram notation can be generalised to all morphisms in a SMC and is known as *string diagrams*. It is a two-dimensional instance of the abstract categorical structure. It is also fully abstract, in the sense that every diagram can be mapped to a unique morphism in the underlying SMC. Figures 2 and 3 show several of the atomic diagrams which make up SMCs. The guide for this notation is that each morphism is represented by a network of wires. Wires are drawn in a way that makes it clear which inputs are connected to which outputs. Because unit objects can be added and dropped at will (using $\rho$ and $\bar{\rho}$), under some conventions the corresponding wires are not drawn at all. Here we choose to draw them as grey lines.

The diagram notation is defined in such a way that morphisms that are equal under the category laws have topologically equivalent diagram representations (Selinger, 2011). That is, if we can deform one diagram to another without cutting wires, then they are equivalent. We can illustrate this kind of topological reasoning with the following simple example. Assuming an abstract tensor $\mathsf{u} : \mathsf{T} \overset{z}{\rightsquigarrow} \mathsf{T}$, one can check that $\sigma \circ (\mathsf{id} \otimes \mathsf{u}) \circ \sigma \circ (\mathsf{id} \otimes \mathsf{u})$ is equivalent to $\mathsf{u} \otimes \mathsf{u}$ by applying a number of algebraic laws, but this is an error-prone process. If we first convert the morphisms to diagram form, we need to check , which is a matter of repositioning the second box.[6]

[6] The property that equivalent morphisms have the same representation is also satisfied by the Einstein notation (up to $\alpha$-renaming). Indeed, a connection in the diagrammatic notation is represented by marking the two

At this point, a reader familiar with programming languages might be tempted to assume that $\mathsf{V} \otimes \mathsf{W}$ is like a pair of V and W. That is, that tensors would not only form a category, but even a Cartesian category. This is not the case: tensors are not equipped with projections nor duplications. This observation justifies the fact that contraction in Einstein notation must involve exactly two indices. Indeed, contraction corresponds to connecting loose wires in the diagram notation, and because we do not have a Cartesian category, only two loose wires can be connected (to make a new continuous wire).

**Addition and scaling** As we saw, tensors of the same type (same domain and codomain) form themselves a vector space, and as such can be scaled and added together. The corresponding categorical structure is called an additive category. Thus, every tensor category $z$ will satisfy the Additive constraint:

$$\textbf{type}\ \mathsf{Additive}\, z = \forall\, \mathsf{a}\, \mathsf{b}.\, \mathsf{VectorSpace}\, (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b})$$

Recalling the definition of VectorSpace from Section 2.1, Additive implies that we have the following two operations for every a and b:

$$(+) :: (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b})$$
$$(\triangleleft) :: \mathsf{S} \rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b})$$

An additive category requires that composition ($\circ$) and tensor products ($\otimes$) are bilinear. In full:

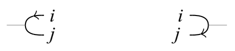
$$\begin{aligned} \mathsf{t} \circ (\mathsf{u} + \mathsf{v}) &= (\mathsf{t} \circ \mathsf{u}) + (\mathsf{t} \circ \mathsf{v}) & \mathsf{t} \otimes (\mathsf{u} + \mathsf{v}) &= (\mathsf{t} \otimes \mathsf{u}) + (\mathsf{t} \otimes \mathsf{v}) \\ (\mathsf{t} + \mathsf{u}) \circ \mathsf{v} &= (\mathsf{t} \circ \mathsf{v}) + (\mathsf{u} \circ \mathsf{v}) & (\mathsf{t} + \mathsf{u}) \otimes \mathsf{v} &= (\mathsf{t} \otimes \mathsf{v}) + (\mathsf{u} \otimes \mathsf{v}) \\ (\alpha \triangleleft \mathsf{t}) \circ \mathsf{u} &= \alpha \triangleleft (\mathsf{t} \circ \mathsf{u}) & (\alpha \triangleleft \mathsf{t}) \otimes \mathsf{u} &= \alpha \triangleleft (\mathsf{t} \otimes \mathsf{u}) \\ \mathsf{t} \circ (\alpha \triangleleft \mathsf{u}) &= \alpha \triangleleft (\mathsf{t} \circ \mathsf{u}) & \mathsf{t} \otimes (\alpha \triangleleft \mathsf{u}) &= \alpha \triangleleft (\mathsf{t} \otimes \mathsf{u}) \end{aligned}$$

We note in passing that there is no obviously good way to represent addition using diagrams. If diagrams should be added together we write them side by side with a plus sign in between.

**Compact Closed Category** There remains to capture the relationship between a vector space $V$ and its associated covector space $V^*$. This is done abstractly using a compact closed category structure (Selinger, 2011). In a compact closed category, every object has a dual, and duals generalise the notion of co-vector space.

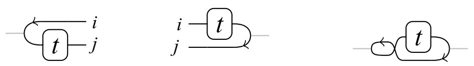
$$\begin{aligned} &\textbf{class}\ (\mathsf{SymmetricMonoidal}\ z) \Rightarrow \mathsf{CompactClosed}\ z\ \textbf{where} \\ &\quad \eta :: \mathbf{1} \overset{z}{\rightsquigarrow} (\mathsf{a}^* \otimes \mathsf{a}) \\ &\quad \epsilon :: (\mathsf{a} \otimes \mathsf{a}^*) \overset{z}{\rightsquigarrow} \mathbf{1} \end{aligned}$$

ends of the connection by the same index name (in sub- or super-script position). Besides, rearranging the inputs or outputs of a morphism is implemented by a renaming of indices. For instance, the right identity law, $\mathsf{t} \circ \mathsf{id} = \mathsf{t}$ becomes $\delta_i^k t_k^j = t_i^j$. (That is, multiplication by $\delta$ can act like a variable substitution operator for indices.) The associativity law $\mathsf{t} \circ (\mathsf{u} \circ \mathsf{v}) = (\mathsf{t} \circ \mathsf{u}) \circ \mathsf{v}$ becomes $(v_i^l u_l^k) t_k^j = v_i^k (u_k^l t_l^j)$: a combination of $\alpha$-equivalence and associativity of multiplication.



| | |
|---|---|
| $\eta$ | $\epsilon$ |
| $\delta_j{}^i$ | $\delta_i{}^j$ |
| $\mathbf{1} \overset{z}{\leadsto} \mathsf{T}^* \otimes \mathsf{T}$ | $\mathsf{T} \otimes \mathsf{T}^* \overset{z}{\leadsto} \mathbf{1}$ |

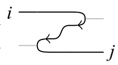
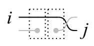
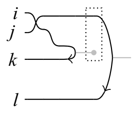
(a) Unit and counit.



| | | |
|---|---|---|
| $(\mathsf{id} \otimes \mathsf{t}) \circ \eta$ | $\epsilon \circ (\mathsf{t} \otimes \mathsf{id})$ | $\epsilon \circ (\mathsf{t} \otimes \mathsf{id}) \circ \sigma \circ \eta$ |
| $t_i{}^j$ | $t_i{}^j$ | $t_i{}^i$ |
| $\mathbf{1} \overset{z}{\leadsto} \mathsf{T}^* \otimes \mathsf{T}$ | $\mathsf{T} \otimes \mathsf{T}^* \overset{z}{\leadsto} \mathbf{1}$ | $\mathbf{1} \overset{z}{\leadsto} \mathbf{1}$ |

(b) Bending input and output connections.

Fig. 4: Illustration of compact closed categories in various notations. Note that Einstein notation does not change when bending connections using $\eta$ or $\epsilon$, though in the third example, the new connection is notated by repeated use of the index.

In the tensor instance, $\eta$ and $\epsilon$ produce and consume correlated products of vectors and covectors. While the algebraic view is very abstract and can be hard to grasp (it is for the authors), the diagrams help. Compact closed categories are required to satisfy the so-called snake laws: one is $(\epsilon \otimes \mathsf{id}) \circ \bar{\alpha} \circ (\mathsf{id} \otimes \eta) = \sigma$, or (  =  ), and the other is symmetrical. These laws ensure that the object $\mathsf{a}^*$ is just like the object $\mathsf{a}$, but travelling backwards (input and output roles are exchanged). To reflect this, in the diagrammatic representation of $\eta$ and $\epsilon$, we indicate the $\mathsf{a}^*$ object with a left-pointing arrow, as shown in Figure 4(a). Indeed, there is no difference between an input vector space $\mathsf{a}$ and an output vector space $\mathsf{a}^*$. Accordingly, in the Einstein notation, no difference is being made between inputs and outputs. Instead only co- or contra-variance is reflected notationally. Consequently neither $\eta$ nor $\epsilon$ are visible in the Einstein notation, except perhaps as a Kronecker $\delta$ (see Figure 4(a)). For instance, the morphism $\epsilon \circ (\bar{\rho} \circ (\mathsf{id} \otimes \epsilon) \otimes \mathsf{id}) \circ (\alpha \otimes \mathsf{id}) \circ ((\sigma \otimes \mathsf{id}) \otimes \mathsf{id})$, or  in diagram notation, is written $\delta_i{}^k \delta_j{}^l$ in Einstein notation. Figure 4(b) shows how an input object $\mathsf{a}$ (of any morphism) can be converted to an output $\mathsf{a}^*$, and *vice versa*. One can even combine both ideas and connect the output of a morphism $\mathsf{t}$ back to its input. By doing so, one constructs the *trace* of $\mathsf{t}$.[7]

We now have a complete description of the tensor combinators—the core of ROGER. Unfortunately, in practice, it is inconvenient to use as such. Indeed, most of the tensor expressions encountered in practice consists of building a network of connections between atomic blocks. Unfortunately, using the categorical combinators for this purpose is tedious. For instance, contracting two input indices is particularly tedious in the point-free notation, because it is realised as a composition with $\eta$ or $\epsilon$ with a large number of SMC combinators to select the appropriate dimensions to contract. It is akin to programming with SKI combinators instead of using the lambda calculus. Using variable names for indices, like in the Einstein notation, would be much more convenient. We will get there in Section 4.

[7] When the contracted index corresponds to a vector space of dimension $\mathsf{n}$ and letting $\mathsf{t}$ be the identity, we find that $\epsilon \circ \sigma \circ \eta = \mathsf{trace}\, \delta = \mathsf{n}$.

### *3.1 Matrix instances*

An important instance of the compact closed category structure is the category of matrices of coefficients, which we encountered in Section 2.2. In our host functional language, we define them as a function from (both input and output) indices to coefficients (of type S):[8]

```
newtype M_e a b = Tab (a → b → S)
```

To emphasise the dependency on the basis, we use a subscript when referring to a specific matrix category morphism, as in $M_b$ where b is a reference to the choice of basis. In the Haskell implementation, this basis is represented by a *phantom* type parameter.

The identity morphism is the identity matrix, and composition is matrix multiplication:

```
instance Category M_e where
  id = Tab δ
  Tab g ∘ Tab f = Tab (λ i j → summation (λ k → f i k ∗ g k j))
δ i j = if i == j then 1 else 0
```

In this instance, the objects are identified with sets that index the bases of the vector spaces that they stand for. These sets are assumed to be enumerable and bounded (so we have access to their inhabitants) and we can compare indices for equality.[9] The instance of the SMC structure for matrix representations in coherent bases is then:

```
instance SymmetricMonoidal M_e where
  (⊗) = kroneckerProduct
  ρ = Tab (λ x (y, ()) → δ x y)
  ρ̄ = Tab (λ (y, ()) x → δ x y)
  α = Tab (λ ((x, y), z) (x′, (y′, z′)) → δ ((x, y), z) ((x′, y′), z′))
  ᾱ = Tab (λ (x′, (y′, z′)) ((x, y), z) → δ ((x, y), z) ((x′, y′), z′))
  σ = Tab (λ (x, y) (y′, x′) → δ (x, y) (x′, y′))
kroneckerProduct (Tab f) (Tab g) = Tab (λ (i, k) (j, l) → f i j ∗ g k l)
```

Because objects index the bases of the corresponding vector spaces, tensor products are represented as usual pairs. In the above definition, the right-hand sides are Haskell code. This means that the asterisk operator $(*)$ denotes multiplication between scalars *as components of matrices*. In contrast, the operator $(\star)$ defined in Section 4 denotes multiplication between abstract scalar (order-0) tensor expressions (independent of the chosen tensor representation).

With the coherent choice of bases, $\eta$ and $\epsilon$ are simply realised as the identity.

```
instance CompactClosed M_e where
  η = Tab (λ () (Dual x, y) → δ x y)
  ϵ = Tab (λ (x, Dual y) () → δ x y)
```

[8] This functional representation is for expository purposes. It must be tabulated to avoid bad runtime behaviour.
[9] In Haskell, the constraints on indices (Enum, Bounded, Eq) are tracked using an *associated class constraint* on objects, which adds significant verbosity but is a well-understood technique (Orchard & Schrijvers, 2010; Sculthorpe *et al.*, 2013). For concision, we omit object constraints entirely in this presentation.

The object $\mathsf{a}^*$ has the same dimensionality as $\mathsf{a}$, so in our Haskell encoding we use **newtype** for it, **newtype** $\mathsf{DualObject\ a = Dual\ a}$. For concision (and following tradition), we use an asterisk as a shorthand, so $\mathsf{a}^*$ stands for the type $\mathsf{DualObject\ a}$.

**Coordinate representation functors.** It is worth mentioning that the transformation functions between linear maps $\mathsf{L}$ and their representations $M_{\mathsf{e}}$ in a given basis $\mathsf{e}$ are a pair of functors which are the identity on objects and just change the morphisms:

$$\begin{aligned}
&\mathsf{toCoordinates}_e &&:: (\mathsf{a} \overset{\mathsf{L}}{\rightsquigarrow} \mathsf{b}) \to (\mathsf{a} \overset{M_{\mathsf{e}}}{\rightsquigarrow} \mathsf{b})\\
&\mathsf{fromCoordinates}_d &&:: (\mathsf{a} \overset{M_{\mathsf{d}}}{\rightsquigarrow} \mathsf{b}) \to (\mathsf{a} \overset{\mathsf{L}}{\rightsquigarrow} \mathsf{b})
\end{aligned}$$

Furthermore, this pair defines an isomorphism between the respective compact-closed categories. Therefore, even though different representations form different categories, one can always transform one to another. The transformation between systems of coordinates, usually presented using transformation matrices (see Section 2.2) can be understood as the composition of $\mathsf{fromCoordinates}$ in the source basis, and $\mathsf{toCoordinates}$ in the target basis:

$$\begin{aligned}
&\mathsf{transform} :: (\mathsf{a} \overset{M_{\mathsf{d}}}{\rightsquigarrow} \mathsf{b}) \to (\mathsf{a} \overset{M_{\mathsf{e}}}{\rightsquigarrow} \mathsf{b})\\
&\mathsf{transform} = \mathsf{toCoordinates}_e \circ \mathsf{fromCoordinates}_d
\end{aligned}$$

## 4 Design of Albert

In this section, we provide the design of ALBERT. We intend our design to match Einstein notation as closely as possible. This aim is achieved except for the following two differences:

1. Indices can range over the dimensions of any space (not just atomic vector spaces[10])
2. Indices are always explicitly bound

The first difference is motivated by polymorphism considerations: we make many functions polymorphic over the vector space that they manipulate, and as a consequence, the corresponding indices can range over the dimensions of tensor or scalar vector spaces. For instance, when we write $\mathsf{delta}_{\mathsf{i}}{}^{\mathsf{j}}$, indices $({}_{\mathsf{i}}, {}^{\mathsf{j}})$ may range over order-2 tensor spaces, in which case the corresponding Einstein notation would be the product of two Kronecker deltas.

The second difference is motivated by the need to follow the functional programming conventions, which is required to embed the DSL in Haskell, or any functional language without macros. Besides, to avoid confusion, in ALBERT we choose names in all letters for combinators. For instance, when the conventional notation is the Greek letter $\delta$, we write delta in ALBERT.

The principles and most of the interface of ALBERT are presented in this section. The tensor-field specific functions are discussed in Section 5. The complete interface is summarised in Figures 9 and 10.

[10] Einstein notation can be easily extended to make indices range over the dimensions of any space, even though this is rarely found in the literature.

**Types** All types are parameterised by $z$, the category which tensors inhabit. The type of an index ranging over the dimensions of vector space $\mathsf{a}$ is $\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}$ (think of $\mathsf{P}$ as "port" or "end of a wire carrying $\mathsf{a}$" in the diagrams), where the variable $\mathsf{r}$ is a technical (scoping) device (made precise in Section 7). For the purpose of using ALBERT, it suffices to know that this variable $\mathsf{r}$ should be consistent throughout any given expression.

The type of expressions is $\mathsf{R}\,z\,\mathsf{r}$. Expressions of this type closely match expressions in Einstein notation. In particular, expressions with several free index variables correspond to higher-order tensors. For example, assuming two free index variables $_\mathsf{i} :: \mathsf{P}\,z\,\mathsf{r}\,\mathsf{T}$ and $^\mathsf{j} :: \mathsf{P}\,z\,\mathsf{r}\,\mathsf{T}^*$, then $\mathsf{w}_\mathsf{i} :: \mathsf{R}\,z\,\mathsf{r}$ represents a covector over $\mathsf{T}$; $\mathsf{v}^\mathsf{j} :: \mathsf{R}\,z\,\mathsf{r}$ represents a vector in $\mathsf{T}$; and $\mathsf{t}_\mathsf{i}{}^\mathsf{j} :: \mathsf{R}\,z\,\mathsf{r}$ represents a linear map from $\mathsf{T}$ to $\mathsf{T}$. (Why this is so will become clear when we present the semantics of tensors, which we will do before the end of the section.)

In sum, exactly as in Einstein notation, our tensor expressions define and manipulate tensors as (abstract) scalar-valued expressions depending on indices. Likewise, the order of the underlying tensor is the sum of the order of the free index variables occurring in it. Even though we present index variables as either super- or subscripts, they are just regular variable names.

Because the underlying category $z$ is not Cartesian, every input must be connected to a single output, and *vice-versa*. Hence, index variables occur exactly once in each term. We enforce this restriction by using (and binding) index variables linearly.[11] Accordingly, the types of the variables $\mathsf{v}, \mathsf{w}, \mathsf{t}$ mentioned above involve (type-)linear functions. For instance $\mathsf{w} :: \mathsf{P}\,z\,\mathsf{r}\,\mathsf{T} \multimap \mathsf{R}\,z\,\mathsf{r}$ is a covector over $\mathsf{T}$; $\mathsf{v} :: \mathsf{P}\,z\,\mathsf{r}\,\mathsf{T}^* \multimap \mathsf{R}\,z\,\mathsf{r}$ is a vector in $\mathsf{T}$; and $\mathsf{t} :: \mathsf{P}\,z\,\mathsf{r}\,\mathsf{T} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{T}^* \multimap \mathsf{R}\,z\,\mathsf{r}$ is a linear map over $\mathsf{T}$. This means that ALBERT uses a higher-order abstract syntax, that is, the abstraction mechanism of the host language provides us with the means to abstract over index variables. The order of a tensor variable is given by taking the sum of the order of the index parameters in its type. So, for instance, delta has order $2m$ if its type argument $\mathsf{a}$ stands for a vector space of order $m$:

$$\mathsf{delta} :: \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}^* \multimap \mathsf{R}\,z\,\mathsf{r}$$

**Tensor embedding, evaluation, and index manipulation** Next, we turn our attention to embedding ROGER into ALBERT. This is done by means of the following combinators:

$$\begin{array}{lll}\mathsf{tensorEmbed} & :: \mathsf{CompactClosed}\,z & \Rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \rightarrow (\forall\, \mathsf{r}.\, \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{b}^* \multimap \mathsf{R}\,z\,\mathsf{r}) \\ \mathsf{tensorEmbed}_1 & :: \mathsf{SymmetricMonoidal}\,z & \Rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathbf{1}) \rightarrow (\forall\, \mathsf{r}.\, \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{R}\,z\,\mathsf{r})\end{array}$$

The special case of a vector, where the target object is the unit, is common enough that it deserves a function of its own. In the general case, we take advantage of the compact-closed structure, and turn the output object ($\mathsf{b}$) of the morphism into an input of an index over the dual object ($\mathsf{b}^*$).

The converse operation consists in evaluating a tensor expression into a morphism:

[11] There are two meanings to the word "linear", which we both use in this paper. The first meaning, exclusively used above refers to linear algebra (Definition 2). The second meaning is evoked by the phrase "linear type", and we use it to enforce usage restrictions of the indices appearing in the tensor expressions. When any ambiguity might exist, we write "type-linear" to refer to this second meaning.

$$\begin{aligned}
&\mathsf{tensorEval} :: (\mathsf{CompactClosed}\,z, \mathsf{Additive}\,z) \Rightarrow\\
&\qquad (\forall\,\mathsf{r}.\,\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{b}^* \multimap \mathsf{R}\,z\,\mathsf{r}) \to (\mathsf{a} \overset{z}{\leadsto} \mathsf{b})\\
&\mathsf{tensorEval}_1 :: (\mathsf{SymmetricMonoidal}\,z, \mathsf{Additive}\,z) \Rightarrow\\
&\qquad (\forall\,\mathsf{r}.\,\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{R}\,z\,\mathsf{r}) \to (\mathsf{a} \overset{z}{\leadsto} \mathbf{1})
\end{aligned}$$

The fact that we can move between these two DSLs freely (using embedding and evaluation) means we can combine their strengths. In both embedding and evaluation, neither a nor b need be atomic types. To match the convention of Einstein notation, the user of ALBERT can break down the corresponding indices into their components after embedding, or conversely combine components before evaluation. Likewise, unit objects might need to be introduced or discarded. The interface for performing such operations is provided in the form of the following four combinators:

$$\begin{aligned}
&\mathsf{unit} :: \mathsf{SymmetricMonoidal}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap (\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}, \mathsf{P}\,z\,\mathsf{r}\,\mathbf{1})\\
&\mathsf{unit}' :: \mathsf{SymmetricMonoidal}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,\mathbf{1} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}\\
&\mathsf{split} :: \mathsf{SymmetricMonoidal}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,(\mathsf{a} \otimes \mathsf{b}) \multimap (\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}, \mathsf{P}\,z\,\mathsf{r}\,\mathsf{b})\\
&\mathsf{merge} :: \mathsf{SymmetricMonoidal}\,z \Rightarrow (\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}, \mathsf{P}\,z\,\mathsf{r}\,\mathsf{b}) \multimap \mathsf{P}\,z\,\mathsf{r}\,(\mathsf{a} \otimes \mathsf{b})
\end{aligned}$$

To sum up, when $z$ is a SMC, the $\mathsf{P}\,z\,\mathsf{r}$ type transformer defines a homomorphism between the monoid of (linear) Haskell pairs and that of tensor products of the category $z$. As an illustration, a function t of type $\mathsf{P}\,z\,\mathsf{r}\,(\mathsf{a} \otimes \mathsf{b}^*) \multimap \mathsf{R}\,z\,\mathsf{r}$ can be curried to $\mathsf{t}' :: \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{b}^* \multimap \mathsf{R}\,z\,\mathsf{r}$. When using $\mathsf{t}'$, each index is its own variable, closely matching Einstein notation.

**Multiplication and contraction** Another pervasive operation in Einstein notation is multiplication. In ALBERT, we use a multiplication operator with a linear type:

$$(\star) :: \mathsf{R}\,z\,\mathsf{r} \multimap \mathsf{R}\,z\,\mathsf{r} \multimap \mathsf{R}\,z\,\mathsf{r}$$

According to the typing rules of the linear function types, the occurrences of variables are accumulated in a function call. This way, the order of the product $\mathsf{t} \star \mathsf{u}$ is the sum of the orders of t and u. Contraction is realised by the following combinator.

$$\mathsf{contract} :: \mathsf{CompactClosed}\,z \Rightarrow (\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}^* \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{R}\,z\,\mathsf{r}) \multimap \mathsf{R}\,z\,\mathsf{r}$$

There are a couple of contrasting points when compared to the Einstein notation. First, we *bind index variables explicitly*, and thus we use an explicit contraction combinator. Indeed, while in Einstein notation indices are not explicitly bound, this liberty cannot be taken in an EDSL based on a lambda calculus. Second, we consider the high and low indices involved in the contraction to be *separate variables*. Indeed, in Einstein notation each version of the index (high or low) must occur exactly once, and thus making them separate linearly bound variables is natural. One can think of it as the contraction creating a wire, with each end of the wire bound to a separate name. Nonetheless, the convention to use the same variable name in different positions is a convenient one. We recover it in this paper by a typographical trick: we use the same Latin letter for both indices and make the position as sub- or superscript integral to variable names. (This is purely a matter of convention and users of ALBERT are free to use whichever variable names they prefer.) Therefore, for instance, in ALBERT the composition of two linear transformations t and u, as shown in Figure 2, is realised as $\lambda\, {}_{\mathsf{i}}{}^{\mathsf{j}} \rightarrow \mathsf{contract}\,(\lambda\, {}^{\mathsf{k}}{}_{\mathsf{k}} \rightarrow \mathsf{t}\,{}_{\mathsf{i}}{}^{\mathsf{k}} \star \mathsf{u}\,{}_{\mathsf{k}}{}^{\mathsf{j}})$.

**Addition and zero** In Einstein notation, one can use the addition operator as if it were the point-wise addition of each of the components, for instance $t_i^j + u_i^j$. Note that the live indices are used in each of the operands of the sum, thus repeated in the whole expression. This means that the following linear type for the sum operator would be incorrect:

```
plus_wrong :: R z r ⊸ R z r ⊸ R z r
```

This is because in the expression $\mathsf{plus}_{wrong}\,(\mathsf{t}\,{}_{\mathsf{i}}{}^{\mathsf{j}})\,(\mathsf{u}\,{}_{\mathsf{i}}{}^{\mathsf{j}})$, both i and j occur twice, while the type would require indices to be split between the left and right operand. Thus, we must use another type. We settle on the following one:

```
plus :: (Bool → R z r) ⊸ R z r
```

This type allows to code $t_i^j + u_i^j$ as follows:[12]

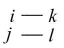
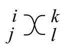
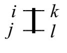
```
plus (λ c → case c of
  False → t i j
  True  → u i j)
```

The above is well typed. Indeed, 1. the argument of plus is type-linear, so any use of indices in its body is considered type-linear; and 2. only one branch of a **case** is considered to be run, and therefore the same indices can (and must) be used in all the branches. The fact that only one branch is run is counter-intuitive because the semantics depends on both of them. We explain our solution to this apparent contradiction in Section 7.2.

Conversely, there is a zero tensor of every possible order. Thus, we have a zero combinator with an index argument ranging over an arbitrary space:

```
zeroTensor :: (CompactClosed z, Additive z) ⇒ P z r a ⊸ R z r
```

The scaling operator ( ◃ ) underpins non-zero constants, with no particular difficulty.

With the primitives of additive categories, one can construct the tensor $\mathsf{antisym} = \mathsf{id} - \sigma\ ::\ \mathsf{T} \otimes \mathsf{T} \overset{z}{\rightsquigarrow} \mathsf{T} \otimes \mathsf{T}$. This tensor can be rendered graphically as the difference  $-$ , but it is useful enough to receive the special notation . Indeed, composing it with an arbitrary tensor gives its antisymmetric part with respect to the two connected indices.[13]

[12] In what follows the False branch corresponds to the left operand and True to the right one. Any two-element type would do.

[13] Consider the case of a matrix $M$. Let $A_{ij} = M_{ij} - M_{ji}$ and $S_{ij} = M_{ij} + M_{ji}$. It is easy to see that $S$ is symmetric and that $A$ is antisymmetric, and $A$ is the composition of antisym with $M$. Furthermore, $M$ is the average of $A$ and $S$: $M = (A + S)/2$. This is why $A$ is called the antisymmetric part of $M$.

## 5 Tensor calculus: Fields and their derivatives

We have up to now worked with tensors as elements of certain vector spaces, but to further illustrate the capabilities of ALBERT, we apply it to tensor *calculus*, starting with the notion of fields.

### 5.1 Tensor fields

In this context a field means that a different value is associated with every position on a given manifold.[14] We denote the position parameter by $\mathbf{X}$. Hence, the scalars (from Definition 1) are no longer just real numbers, but rather real-valued expressions depending on $\mathbf{X}$.[15] For instance, $\mathbf{X}$ could be a position on the surface of the earth, and a scalar field could be the temperature at each such point.

A vector field associates a vector with every position; for instance, the wind direction. The perhaps surprising aspect is that each of these vectors may inhabit a different, *local*, vector space, which can be thought of as tangent to the manifold at the considered point. So in our example, we assumed that the wind is parallel to the earth surface. Hereafter, we assume such a local space for each category $z$, and call it $T_z$, leaving the dependency on position $\mathbf{X}$ implicit. Even though in the typical case the local vector space is different at each position, it keeps the same dimensionality. Therefore, as an object, it is independent of $\mathbf{X}$. In Haskell terms, $T_z$ is an associated type; see Section 5.2.

When we deal with matrix representations and want to perform computations with them, we need a way to identify the position $\mathbf{X}$. For a general manifold, this is difficult to do, but we will restrict our scope to the case where a single coordinate system is sufficient. We also need a basis at each position, which gives a meaning to the entries in a tensor matrix representation (the meaning of these coordinates change with position). Furthermore, different choices of coordinate system are possible for the same manifold. The coordinates used to identify the position will be referred to as the global coordinates, while the coordinates of a tensor will be referred to as local coordinates. This terminology is not usual in mathematical praxis. However, we find that making this distinction is useful to lift ambiguities.[16]

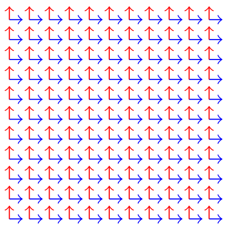
While the choice of basis field is arbitrary from an algebraic perspective, some choices of basis will make certain computations easier than others. Given a system of coordinates to identify positions in the manifold, there is a canonical way to define the local basis field. It is to let the base vectors be the partial derivatives of the position $\mathbf{X}$ with respect to each global coordinate. This yields base vectors which are tangent to coordinate lines in the manifold. In Figure 5, one example follows Cartesian coordinate lines, and the other polar coordinate lines. In the polar case, we have the base vector fields $(\mathbf{e}_\rho, \mathbf{e}_\theta)$[17] as basis for $T_{M_\mathsf{P}}$, with $\mathbf{e}_\rho = \partial\mathbf{X}/\partial\rho$ and $\mathbf{e}_\theta = \partial\mathbf{X}/\partial\theta$.

[14] A manifold is a topological space that looks like Euclidean space near each point.

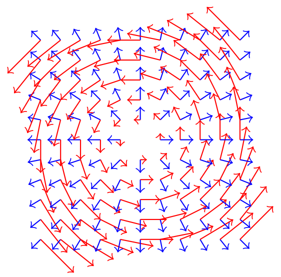
[15] We support such expressions in our implementation, but this generalisation being orthogonal to the rest of the development, we won't discuss it further.

[16] In general, the global view is an *atlas* (a collection of charts) and *transition maps* (between overlapping charts). The coordinates within each chart are called local coordinates in the usual praxis. What is called "local basis" here is usually called a "frame" (as part of a "frame bundle").

[17] When used as indices $\rho$ and $\theta$ act as symbols which the index variables range over. Essentially, $\rho = 1$ and $\theta = 2$. They are not index variables themselves.

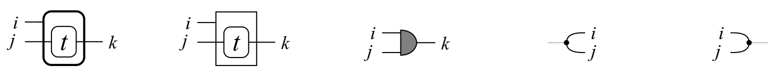


(a) Cartesian



(b) Polar

Fig. 5: Possible fields of bases for a local space field covering the Euclidean plane. In both examples, basis vectors are tangent to coordinate lines; either Cartesian or polar coordinates. In the second instance, the basis vectors are undefined at the origin.



| | | | | |
|---|---|---|---|---|
| ∇ t | ∂ t | Γ | g′ | g |
| $\nabla_i t_j{}^k$ | $\partial_i t_j{}^k$ | $\Gamma_{ij}{}^k$ | $g'^{ij}$ | $g_{ij}$ |
| $T_z \otimes T_z \overset{z}{\rightsquigarrow} T_z$ | $T_z \otimes T_z \overset{z}{\rightsquigarrow} T_z$ | $T_z \otimes T_z \overset{z}{\rightsquigarrow} T_z$ | $\mathbf{1} \overset{z}{\rightsquigarrow} T_z \otimes T_z$ | $T_z \otimes T_z \overset{z}{\rightsquigarrow} \mathbf{1}$ |

Fig. 6: Tensor field primitives in various notations, and their types.

### *5.2 Metrics and index juggling*

An important additional structure that one can add to vector spaces are associated *metrics*. The (covariant) metric, noted $g$, is what defines the inner product of (local) vectors. It is a tensor field that, when given two vectors as input, returns a scalar, their inner product. One defines the contravariant metric $g'$ as the inverse of the covariant metric.[18] This can be specified as $g_{ik}g'^{kj} = \delta_i^j$.

To capture this algebraic structure, we distinguish the vector space for which we define the metric as an associated type, $T_z$, of a new category class MetricCategory.

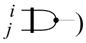
The various notations for metrics are shown in Figure 6.

```
class (CompactClosed z) ⇒ MetricCategory z where
  type T_z
  g  :: (T_z ⊗ T_z) ⇝^z 1
  g′ :: 1 ⇝^z (T_z ⊗ T_z)
```

Then we can embed the metric morphism in ALBERT as follows:

```
metric :: MetricCategory z ⇒ P z r T_z ⊸ P z r T_z ⊸ R z r
metric i j = tensorEmbed₁ g (merge (i, j))
```

[18] In the literature, both metrics are written using the same symbol (typically $g$), relying on the context to disambiguate.

The metric tensor is symmetric—a fact which can be expressed as its antisymmetric part (  ) being zero. The coefficients of the matrix representation of the metric are given by the inner product of the basis vectors. For the usual Cartesian basis, the metric is represented by the identity matrix. For the basis $(\mathbf{e}_\rho, \mathbf{e}_\theta)$ defined above, the matrix representation of the metric is

$$g_{\mathsf{Polar}} = \begin{bmatrix} \rho^2 & 0 \\ 0 & 1 \end{bmatrix}$$

With this knowledge, we can define an instance for coordinate representations of the MetricCategory class:

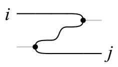
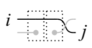
```
instance MetricCategory M_Polar where
  type T_{M_Polar} = Atom Polar
  g = Tab $ λ (Atom i, Atom j) _ → case (i, j) of
    (Rho,   Rho)   → variable Rho ^ 2
    (Theta, Theta) → 1
    _              → 0
```

The scalars associated with this additive category are expressions where the variables $\rho$ and $\theta$ occur, and the variable function embeds a coordinate name into such expression types:

```
variable :: e → S_e
```

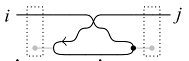
Thus, the Polar data type serves triple duty. First, it is the set of variables used in such expressions. Second, it serves to identify the meaning of coordinates as a parameter in the $M_{\mathsf{Polar}}$ type. Third, it is used to construct a representation of the atomic vector space. For the matrix category, an object is a set indexing the base vectors, so Atom is a simple wrapper around the coordinate type:

```
newtype Atom s = Atom s
```

One can contrast the type of metrics with that of $\eta$ and $\epsilon$. First, only the tangent space $T_z$ has a metric, while every object has a dual. Indeed, $\eta$ and $\epsilon$ are both realised as $\delta$ in the tensor instance, and are an identity operation in Einstein notation. They have no geometric significance and thus can be defined generically. In contrast, the metric depends on the geometric properties of the space $T_z$ that they operate upon. Second, the metrics do not dualise objects, whereas $\eta$ and $\epsilon$ do. Accordingly, our diagrammatic notation does not make any special mark on the input/output wires of metrics. Yet, metrics satisfy a variant of the snake laws:  is equal to  .

By combining the compact closed categorical structure with metrics, one can construct the following two morphisms (in ROGER):

```
juggleDown :: MetricCategory z ⇒ T_z^* ⇝^z T_z
juggleUp   :: MetricCategory z ⇒ T_z ⇝^z T_z^*
juggleDown = ρ̄ ∘ σ ∘ ((ϵ ∘ σ) ⊗ id) ∘ ᾱ ∘ (id ⊗ g′) ∘ ρ
juggleUp = ρ̄ ∘ (id ⊗ g) ∘ α ∘ (σ ⊗ id) ∘ ᾱ ∘ (id ⊗ η) ∘ ρ
```

In diagram form juggleUp is 

. The existence of such morphisms have a direct consequence on Einstein notation: any subscript index $\mathsf{P}\, z\, \mathsf{r}\, \mathsf{a}$ can be raised into $\mathsf{P}\, z\, \mathsf{r}\, \mathsf{a}^*$, and *vice versa*. In other words, any index can be used as super or subscript as needed. As a first example, given a vector $v^i$, the covector $v_i$ is a shorthand for $v^k g_{ki}$. It is important to note that the position of the indices changes the value of the expression: the actual numbers (the components) of a vector $v^i$ are different from those of the corresponding covector $v_i$ unless the metric representation is the identity (as in global Cartesian coordinates). As another example, assuming a tensor $\mathsf{t} : T_z \otimes T_z \overset{z}{\rightsquigarrow} T_z$, the expression $t_{ijl}$ is a shorthand for $t_{ij}{}^k g_{kl}$.

In the mathematical praxis, raising and lowering indices are also referred to as index juggling. In ALBERT, index juggling is realised explicitly: our recent example can be written $\mathsf{t}_{\,\mathsf{i}\,\mathsf{j}}\,(\mathsf{raise}_{\,\mathsf{l}})$. The raising and lowering functions have the following types and are implemented by embedding juggleUp and juggleDown:

$$
\begin{aligned}
&\mathsf{raise} :: \mathsf{MetricCategory}\, z \Rightarrow \mathsf{P}\, z\, \mathsf{r}\, T_z \multimap \mathsf{P}\, z\, \mathsf{r}\, T_z^* \\
&\mathsf{lower} :: \mathsf{MetricCategory}\, z \Rightarrow \mathsf{P}\, z\, \mathsf{r}\, T_z^* \multimap \mathsf{P}\, z\, \mathsf{r}\, T_z
\end{aligned}
$$

### *5.3 Change of global coordinate system*

As we have seen, Einstein notation is carefully set up to work without reference to the system of coordinates used to identify positions in the manifold. In practice, this means that one can reason about tensors, their derivatives (Section 5.4), etc. without worrying about the choice of coordinates. It is only at the very last stage that one chooses a system of global coordinates where the data of the problem is easy to express and calculate with. We have seen that one can convert between bases by applying the appropriate transformations on the representation matrices (see Section 2.2). For canonical bases (defined as partial derivatives of the position), the transform function is obtained by composing with Jacobian. In our running example, we have at our disposal the Jacobian $J(T)$ and its inverse $J(T^*)$ to convert between polar and Cartesian global coordinates with $J(T^*) = \begin{bmatrix} \cos\theta & \sin\theta \\ \rho^{-1}\sin\theta & -\rho^{-1}\cos\theta \end{bmatrix}$.

As an illustration, let $\tilde{v}_{\mathsf{Polar}} = \begin{bmatrix} -\rho^{-1} & 0 \end{bmatrix}$ be the local coordinates of a co-vector field in the polar tangent basis $\tilde{v}_{\mathsf{Polar}} : T_{M_{\mathsf{Polar}}} \overset{M_{\mathsf{Polar}}}{\rightsquigarrow} \mathbf{1}$. Then the local coordinates of the co-vector field in the Cartesian tangent basis are given by $v_{\mathsf{Cartesian}\,i} = J(T)^j_i v_{\mathsf{Polar}\,j}$, and we have: $\tilde{v}_{\mathsf{Cartesian}} = \begin{bmatrix} -\rho^{-1}\cos\theta & -\rho^{-1}\sin\theta \end{bmatrix}$.

### *5.4 Spatial derivative: Levi-Civita connection*

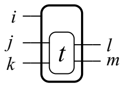
The local vector space can change from position to position, but it is assumed to vary smoothly. One says that a *connection* is defined between neighbouring spaces. This means that we can take the derivative of tensor fields with respect to position. Two cautionary remarks are in order regarding our use of the phrase *spatial derivative*. First, the terminology is meant to include hyperbolic geometries, thus space-time (Minkowski space) is covered. Second, there are other notions of spatial derivatives, but here we will only consider the case of the Levi-Civita connection, more precisely, which has additional properties: see footnote 20.

Considering the simplest case, the derivative of a scalar field s is a covector field: its gradient. That is, given a direction vector $\vec{v}$, the gradient will return the slope of s in the $\vec{v}$ direction. In general, the spatial derivative of a tensor takes a vector argument and returns the variation in this direction. So the derivative of a tensor field is itself a tensor field, whose (covariant) order is one more than its argument, and for this reason, the spatial derivative of tensors is called the covariant derivative. We capture this in the following class, which signals the presence of a connection (and spatial derivative) in a category.

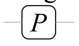
```
class MetricCategory z ⇒ ConnectionCategory z where
  ∇ :: (a ⇝ᶻ b) → ((T_z ⊗ a) ⇝ᶻ b)
```

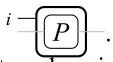
Accordingly, in Einstein notation, the covariant derivative uses an additional (lower) index.

```
deriv :: ConnectionCategory z ⇒ P z r v ⊸ R z r ⊸ R z r
```

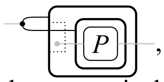
The actual implementation is too involved to present just yet, and deferred to Section 7. In the diagrammatic notation, we represent the covariant derivative as a thick box around the tensor whose derivative is taken. This box adds an input wire for the additional input vector, and propagates the wires of the tensor which it encloses, reflecting the propagation of the types a and b in the type of ∇. In Figure 6, we illustrate with a derivative applied to a tensor of order (1,1), but it can be applied to any morphism, with arbitrary domain and codomain—and hence to tensors of arbitrary order. For instance, the covariant derivative of a tensor t of order (2,2) is written $\nabla_i t_{jk}{}^{lm}$ or 

, but (still) ∇ t in ROGER.

**Example: Laplacian.** As a simple example, we express the Laplacian (the divergence of the gradient) of a given scalar field $P$. Its physical meaning has no bearing on our development; but if $P$ is an energy potential, its gradient is the corresponding force field, and its divergence the density of an associated charge or mass. We write $P$ for the potential field, or 

as a diagram. The scalar character of this field is represented by the lack of indices, or the use of the unit object for its domain and codomain. Its gradient is denoted $\nabla_i P$, or 

. Again, indices or objects indicate that we have an order (1,0) (covector) field. At each point, the local covector is a linear function that takes a direction (vector) into the slope in that direction (a scalar).

The Laplacian is a linear combination of second order derivatives. To compute the linear combination, we need to apply contraction, which always expects an upper and a lower index, but we have two covariant (lower) indices, so we must raise one index by multiplying with the contravariant metric. The Einstein notation is $g'^{ij}\nabla_i\nabla_j P$, the corresponding diagram is 

, and in ROGER it is (∇ ((∇ P) ∘ $\rho$)) ∘ g′. The Einstein notation, while already economical, can be made even more concise by using the index juggling convention: $\nabla_i\nabla^i P$. In ALBERT, the same thing would be written

```
contract (λ ⁱ ᵢ → deriv ᵢ (deriv (lower ⁱ) potential))
```

Here the high index from the contraction must be lowered because deriv can only take a low index as its first argument.

### 5.4.1 Laws of covariant derivatives

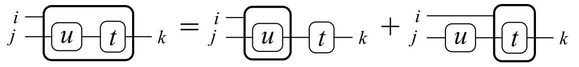
As one might expect, the covariant derivative satisfies the product law for derivatives:

$$\nabla_i(tu) = (\nabla_i t)u + t(\nabla_i u) \tag{5.1}$$

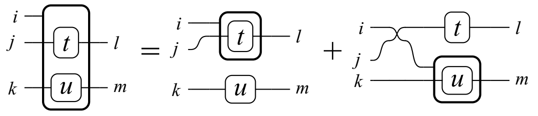
The above formulation is concise, but using it as basis for implementation can be tedious, because one needs to track free and bound variables. A pitfall is that $t$ and $u$ stand for arbitrary expressions, and index variables may occur free in them. Therefore, a specific implementation difficulty is that one needs to preserve the linearity of index variables at the level of the host language.[19] Thus, we find that the morphism EDSL ROGER is a better implementation vehicle in this case. The laws become noticeably more verbose, but not horribly so, and dispense of tracking free variables. In this notation, the product law is expressed as two cases, one for each of the composition ($\circ$) and tensor ($\otimes$) operators:

$$\begin{aligned} \nabla\,(\mathsf{t} \circ \mathsf{u}) &= \mathsf{t} \circ (\nabla\,\mathsf{u}) + (\nabla\,\mathsf{t}) \circ (\mathsf{id} \otimes \mathsf{u}) \\ \nabla\,(\mathsf{t} \otimes \mathsf{u}) &= (\nabla\,\mathsf{t} \otimes \mathsf{u}) \circ \bar{\alpha} + (\mathsf{t} \otimes \nabla\,\mathsf{u}) \circ \alpha \circ (\sigma \otimes \mathsf{id}) \circ \bar{\alpha} \end{aligned}$$

As before, we find the corresponding diagrams more readable:



(5.2)



(5.3)

The derivative of all constant morphisms is zero: $\nabla\,\mathrm{id} = 0$, $\nabla\,\rho = 0$, $\nabla\,\alpha = 0$, etc. This property also holds for the (co)metric tensors:[20] $\nabla\,g = 0 = \nabla\,g'$. Together, the above laws fully specify the structural behaviour of the derivative and the implementation of the covariant derivative falls out from them with no additional difficulty. However, the important case of the derivative of a tensor in a coordinate category remains to be addressed. This question leads us to the concept of affinity.

### 5.4.2 Partial derivatives, christoffel symbols and affinities

One may be tempted to think that the coefficient representation of the covariant derivative is the index-wise derivative of the coefficient representations. While this is true if the metric is the identity everywhere on the manifold, it is not the case in general.

Conventionally, one speaks of the "partial derivative" for the index-wise derivative of coefficients and retains the term "covariant derivative" for the spatial derivative. (The notations are shown in Figure 6, but note that $\partial_i t$ stands for the partial derivative of the

[19] In a draft version of this work, we attempted doing this and found the result inscrutable.

[20] The condition $\nabla\,g = 0 = \nabla\,g'$ holds only for Levi-Civita connections. Even though our general framework can be generalised to support other derivatives, all our examples fit this case.

expression t with respect to the $i$th coordinate, sometimes also written $\partial t/\partial x^i$.) As usual, we express the availability of this new operation by means of a class:

$$\begin{aligned}&\textbf{class}\ \mathsf{MetricCategory}\ z \Rightarrow \mathsf{CoordinateCategory}\ z\ \textbf{where}\\ &\quad \partial :: (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \to ((T_z \otimes \mathsf{a}) \overset{z}{\rightsquigarrow} \mathsf{b})\end{aligned}$$

The Levi-Civita connection is unique for a given metric, but we still require the user to provide an implementation. For categories with coordinates, discussed in the next subsection, a canonical implementation of $\nabla$ is provided.

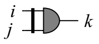
We make the concept available in ALBERT like this:

$$\mathsf{partial} :: \mathsf{CoordinateCategory}\ z \Rightarrow \mathsf{P}\ z\ \mathsf{r}\ \mathsf{v} \multimap \mathsf{R}\ z\ \mathsf{r} \multimap \mathsf{R}\ z\ \mathsf{r}$$

In the general case, to compute the covariant derivative, one must account for the variation of the basis. Therefore, the partial derivative must be corrected by a so called affinity term. The partial derivative accounts for the variation of (the representation of) the tensor field itself as the position varies, while the affinity term accounts for the variation of the basis. The variation of the basis is measured by the *Christoffel symbol*, denoted $\Gamma$ and of type $(T_z \otimes T_z) \overset{z}{\rightsquigarrow} T_z$ (Figure 6). Different choices of local basis field for the *same* manifold will yield different values for it. (Therefore, even though $\Gamma$ is a morphism in a matrix category, and even though it can be transformed to another basis by multiplication with Jacobians, this transformed version will not be the Christoffel symbol for the new basis. This fact is sometimes expressed in textbooks as "$\Gamma$ is not a tensor.")

The Christoffel symbol is often treated abstractly, but it is determined by the (coefficient representation of) the metric:

$$\Gamma_{ij}{}^{k} = \frac{1}{2}g'^{lk}\partial_j g_{il} + \frac{1}{2}g'^{mk}\partial_i g_{jm} - \frac{1}{2}g'^{nk}\partial_n g_{ji} \tag{5.4}$$

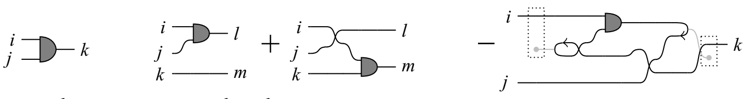
The Christoffel symbol is always symmetric in its first two indices, which is equivalent to asserting that the following diagram is zero: . In the implementation, we make $\Gamma$ a method of the class CoordinateCategory, but with a default definition in terms of Equation (5.4). We make it available in ALBERT by embedding the morphism $\Gamma$ as follows:

$$\begin{aligned}&\mathsf{christoffel} :: \mathsf{CoordinateCategory}\ z \Rightarrow \mathsf{P}\ z\ \mathsf{r}\ T_z \multimap \mathsf{P}\ z\ \mathsf{r}\ T_z \multimap \mathsf{P}\ z\ \mathsf{r}\ T_z^* \multimap \mathsf{R}\ z\ \mathsf{r}\\ &\mathsf{christoffel}_{\,\mathsf{i}\,\mathsf{j}}{}^{\mathsf{k}} = \mathsf{tensorEmbed}\ \Gamma\ (\mathsf{merge}\ ({}_{\mathsf{i}}, {}_{\mathsf{j}}))\ {}^{\mathsf{k}}\end{aligned}$$

For a 2-dimensional atomic vector space, the Christoffel symbol has $2^3 = 8$ components. For our running example of the polar coordinate system and associated canonical tangent space, we can write $\Gamma$ as two $2 \times 2$ symmetric matrices, as follows:

$$\Gamma^{\rho} = \begin{bmatrix} 0 & 0 \\ 0 & -\rho \end{bmatrix} \qquad \Gamma^{\theta} = \begin{bmatrix} 0 & 1/\rho \\ 1/\rho & 0 \end{bmatrix}$$

In textbooks on tensors, for instance that of Lovelock & Rund (1989), one often sees the relation between covariant and partial derivatives expressed as a family of equations, depending on the order of the tensor whose derivative is taken:



| $\Gamma_{ij}{}^{k}$ | $\delta_m{}^{k}\Gamma_{ij}{}^{l} + \delta_l{}^{j}\Gamma_{ik}{}^{m}$ | $-\Gamma_{ik}{}^{j}$ |
|---|---|---|
| $T_z$ | $T_z \otimes T_z$ | $T_z^*$ |

Fig. 7: Affinities aff a at various types a (shown in the third row).

$$
\begin{aligned}
\nabla_i T &= \partial_i T \\
\nabla_i T_j &= \partial_i T_j - \Gamma_{ij}{}^{k} T_k \\
\nabla_i T^j &= \partial_i T^j + \Gamma_{ik}{}^{j} T^k \\
\nabla_i T_j{}^{k} &= \partial_i T_j{}^{k} - \Gamma_{il}{}^{k} T_j{}^{l} + \Gamma_{ij}{}^{l} T_l{}^{k}
\end{aligned}
$$

etc. Using this sort of definition is particularly error prone (can you spot quickly whether there is a mistake in the last line?). In contrast, ROGER captures all cases in one go:

$$\nabla\, \mathsf{t} = \partial\, \mathsf{t} - (\mathsf{t} \circ \mathsf{affinity}) + (\mathsf{affinity} \circ (\mathsf{id} \otimes \mathsf{t})) \tag{5.5}$$

The complexity is pushed down into affinity, which is invoked once for the domain and once for the codomain of t. The affinity is a family of morphisms affinity :: $T_{M_\mathsf{b}} \otimes \mathsf{a} \overset{M_\mathsf{b}}{\rightsquigarrow} \mathsf{a}$ for any object a constructed from the local vector space $T_{M_\mathsf{b}}$ and refers to a specific basis for it, using some coherent canonical choice of bases (with b as a basis for $T_{M_\mathsf{b}}$). The affinity for arbitrary vector spaces is defined by induction on the structure of the corresponding object. The affinity for a product object is the sum of affinities for each of the components, leaving the other component untouched. The affinity for a dual object is the negative affinity of the underlying object, with input and output suitably swapped. This can be coded by a type-dependent set of equations:[21]

```
aff a :: CoordinateCategory z ⇒ P z r T_z ⊸ P z r a ⊸ P z r a* ⊸ R z r
aff T_z         i j         k    =   Γ i j k
aff (a ⊗ b) i (j ⨾ k) (l ⨾ m) =   delta k m ⋆ aff a i j l + delta j l ⋆ aff a i k m
aff a*          i j         k    = − aff a i k j
aff 1           i j         k    =   zeroTensor (i ⨾ j ⨾ k)
```

Here, ALBERT is much more concise than ROGER (we omit the corresponding expressions entirely). The corresponding graphical notation is shown in Figure 7 for each case (but using atomic types in place of a proper induction).

## 5.5 Example: Computing the Laplacian in an arbitrary coordinate system

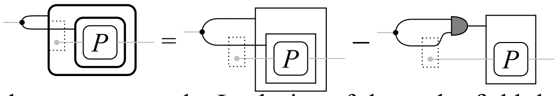
Returning to the example of the potential field, Equation (5.5) and aff tell us that its covariant derivative is equal to its partial derivative, regardless of the value of Γ: $\nabla_i P = \partial_i P$.

[21] For readability, we take some liberties in the presentation of aff: 1. we pattern match on types, but Haskell requires using singleton types for this purpose. 2. The (⨾) operator is used in place of merge in expressions and in place of split in patterns. 3. We use the operators (+) and (−) for the addition and scaling by (− 1) instead of the syntax suggested in Section 3.

Indeed, its domain and codomain are both **1**, and therefore the affinities are both zero. (To compute the value of $\nabla_i P$ in a given coordinate system, one would still need to multiply by the Jacobian as indicated earlier.)

However, when computing the second derivative, a non-zero affinity arises (because the 1st derivative has a non-unit domain.) One has therefore:

$$\nabla_i \nabla^i P = g'^{ij} \nabla_i \nabla_j P = g'^{ij} \partial_i \partial_j P - g'^{kl} {\Gamma_{kl}}^m \partial_m P \tag{5.6}$$

In diagram notation: 

As an illustration, let us compute the Laplacian of the scalar field defined as growing with the negated logarithm of the distance to the origin: $P = -\log(\rho)$.

Its covariant derivative (or gradient) is given by the partial derivatives in polar coordinates (because aff $\mathbf{1} = 0$, as already mentioned). The components in the polar tangent basis are

$$\nabla P = \partial P - 0 + 0 = \begin{bmatrix} \partial P/\partial\rho & \partial P/\partial\theta \end{bmatrix} = \begin{bmatrix} -\rho^{-1} & 0 \end{bmatrix}$$

We have seen above that, by composing with the appropriate Jacobian, the components of the gradient in the Cartesian tangent basis are $\begin{bmatrix} -\rho^{-1}\cos\theta & -\rho^{-1}\sin\theta \end{bmatrix}$.

The second derivative ($\nabla\nabla P$) is a second-order tensor, and computing it manually in the Cartesian basis is error-prone. In contrast, because $P$ has radial symmetry, in the polar tangent basis its partial derivative $\partial P$ has a simple expression (the co-vector $\begin{bmatrix} -\rho^{-1} & 0 \end{bmatrix}$). Thus, only the $\rho - \rho$-component of the second partial derivative is non-zero: $\partial^2 P/\partial\rho^2 = \rho^{-2}$. To compute the covariant derivative, we also need the affinity term, which is obtained by multiplying the gradient $\nabla P$ by $\Gamma$, then multiply by the contravariant metric (recall Equation 5.6). Because only the $\mathbf{e}_\rho$ coefficient of $\nabla P$ is non-zero, is suffices to multiply this component ($-\rho^{-1}$) by $\Gamma^\rho$ to obtain the affinity term $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$. Finally, we get $\nabla\nabla P$ (in polar tangent coordinates) from Equation (5.5): the partial derivative minus the affinity we computed above plus the second affinity which is zero (because the output space is **1**) which gives us $\begin{bmatrix} \rho^{-2} & 0 \\ 0 & -1 \end{bmatrix}$.

The Laplacian is obtained by 2-way contraction with the contravariant metric. One way to do this is to multiply the above by the contravariant metric, to get $\begin{bmatrix} \rho^{-2} & 0 \\ 0 & -\rho^{-2} \end{bmatrix}$, and take the trace, which is zero for $\rho > 0$. (Note that even if it were non-zero there was no need to multiply by any Jacobian because the divergence field is scalar.) The fact that the Laplacian of this scalar field is zero means that the charge density is zero away from the origin: in a two-dimensional space the potential of a point charge at the origin is proportional to this scalar field $P$.

Beyond its illustrative benefits, a takeaway from this example is that all computations were free of trigonometry, and involved many zeros, thanks to the choice of a global coordinate system which had a zero gradient in one of the axes. Even though each step in the computations was presented for illustrative purposes, the MetricCategory $M_{\mathsf{Polar}}$ instance meant that we could run all the above computations as Haskell programs.

$$\begin{tikzcd}
\mathsf{Category} & & \\
\mathsf{SymmetricMonoidal} \arrow[u] & & \\
\mathsf{CompactClosed} \arrow[u] & & \mathsf{Additive} \\
& \mathsf{MetricCategory} \arrow[ul] \arrow[ur, dotted] & \\
\mathsf{ConnectionCategory} \arrow[ur] & & \mathsf{CoordinateCategory} \arrow[ul]
\end{tikzcd}$$

Fig. 8: Inheritance relationships between tensor structures. In the praxis, every instance of a category with a metric Additive is also additive. However, diagrams don't support a good representation for addition, so the dotted line is implemented as a subclass relationship in our library.

```
type P :: (Type → Type → Type) → Type → Type → Type

unit   :: SymmetricMonoidal z ⇒ P z r a ⊸ (P z r a, P z r 1)
unit′  :: SymmetricMonoidal z ⇒ P z r 1 ⊸ P z r a ⊸ P z r a
split  :: SymmetricMonoidal z ⇒ P z r (a ⊗ b) ⊸ (P z r a, P z r b)
merge :: SymmetricMonoidal z ⇒ (P z r a, P z r b) ⊸ P z r (a ⊗ b)

raise  :: MetricCategory z ⇒ P z r T_z ⊸ P z r T_z^*
lower :: MetricCategory z ⇒ P z r T_z^* ⊸ P z r T_z
```

Fig. 9: Syntax of the index language of ALBERT, as types of combinators.

### *5.6 Tensor calculus summary*

At this point we have presented all the classes whose morphisms constitute the combinators of ROGER. Their relationships are summarised in Figure 8.

When it comes to ALBERT, we have two separate sub-languages. First, we have a number of combinators which only manipulate indices, shown in Figure 9. Indices can be split, merged, raised and lowered. Indices for unit vector-spaces are unimportant and can be created or discarded at will. Second, we have a number of combinators which nominally manipulate scalar-valued expressions (addition, multiplication, embedding of constant tensors, Kronecker delta, and contraction, etc; see Figure 10). Various combinators require various amounts of structure in the underlying category $z$.

The semantics in terms of morphisms is provided by the tensorEval function, and $\mathsf{tensorEval}_1$ for the special case of closed tensor expressions. For tensor fields, primitives for metrics and derivatives are also available.

$$\textbf{type}\ \mathsf{R} :: (\mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}) \to \mathsf{Type} \to \mathsf{Type}$$

$$\begin{array}{ll}
\mathsf{plus} & :: (\mathsf{Bool} \to \mathsf{R}\,z\,\mathsf{r}) \multimap \mathsf{R}\,z\,\mathsf{r} \\
(\star) & :: \mathsf{SymmetricMonoidal}\,z \Rightarrow \mathsf{R}\,z\,\mathsf{r} \multimap \mathsf{R}\,z\,\mathsf{r} \multimap \mathsf{R}\,z\,\mathsf{r} \\
\mathsf{zeroTensor} & :: (\mathsf{CompactClosed}\,z, \mathsf{Additive}\,z) \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{R}\,z\,\mathsf{r} \\
\mathsf{constant} & :: (\mathsf{SymmetricMonoidal}\,z, \mathsf{VectorSpace}\,(\mathbf{1} \overset{z}{\leadsto} \mathbf{1})) \Rightarrow \mathsf{S} \to \mathsf{R}\,z\,\mathsf{r} \\
\mathsf{delta} & :: \mathsf{CompactClosed}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}^* \multimap \mathsf{R}\,z\,\mathsf{r} \\
\mathsf{contract} & :: \mathsf{CompactClosed}\,z \Rightarrow (\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}^* \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{R}\,z\,\mathsf{r}) \multimap \mathsf{R}\,z\,\mathsf{r}
\end{array}$$

$$\begin{array}{llll}
\mathsf{tensorEmbed} & :: \mathsf{CompactClosed}\,z & \Rightarrow (\mathsf{a} \overset{z}{\leadsto} \mathsf{b}) \to (\forall\,\mathsf{r}.\,\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{b}^* & \multimap \mathsf{R}\,z\,\mathsf{r}) \\
\mathsf{tensorEmbed}_1 & :: \mathsf{SymmetricMonoidal}\,z & \Rightarrow (\mathsf{a} \overset{z}{\leadsto} \mathbf{1}) \to (\forall\,\mathsf{r}.\,\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} & \multimap \mathsf{R}\,z\,\mathsf{r})
\end{array}$$

$$\begin{array}{ll}
\mathsf{tensorEval} & :: (\mathsf{CompactClosed}\,z, \mathsf{Additive}\,z) \quad \Rightarrow \\
& \quad (\forall\,\mathsf{r}.\,\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{P}\,z\,\mathsf{r}\,\mathsf{b}^* \multimap \mathsf{R}\,z\,\mathsf{r}) \to (\mathsf{a} \overset{z}{\leadsto} \mathsf{b}) \\
\mathsf{tensorEval}_1 & :: (\mathsf{SymmetricMonoidal}\,z, \mathsf{Additive}\,z) \Rightarrow \\
& \quad (\forall\,\mathsf{r}.\,\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \qquad\qquad \multimap \mathsf{R}\,z\,\mathsf{r}) \to (\mathsf{a} \overset{z}{\leadsto} \mathbf{1})
\end{array}$$

$$\begin{array}{l}
\mathsf{deriv} :: \mathsf{ConnectionCategory}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,\mathsf{v} \multimap \mathsf{R}\,z\,\mathsf{r} \multimap \mathsf{R}\,z\,\mathsf{r} \\
\mathsf{partial} :: \mathsf{CoordinateCategory}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,\mathsf{v} \multimap \mathsf{R}\,z\,\mathsf{r} \multimap \mathsf{R}\,z\,\mathsf{r} \\
\mathsf{metric} :: \mathsf{MetricCategory}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,T_z \multimap \mathsf{P}\,z\,\mathsf{r}\,T_z \multimap \mathsf{R}\,z\,\mathsf{r} \\
\mathsf{christoffel} :: \mathsf{CoordinateCategory}\,z \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,T_z \multimap \mathsf{P}\,z\,\mathsf{r}\,T_z \multimap \mathsf{P}\,z\,\mathsf{r}\,T_z^* \multimap \mathsf{R}\,z\,\mathsf{r}
\end{array}$$

Fig. 10: Syntax of the expression sub-language of ALBERT, as types of combinators. We repeat christoffel symbol and metric here even though they can be defined by the user as the embedding of the corresponding ROGER primitives.

## 6 Application: Curvature and general relativity

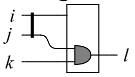
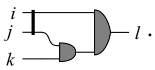
To further demonstrate the applicability of ALBERT, in this section, we present some concepts of general relativity, with particular focus on the notion of curvature. General relativity can be summarised as "matter curves space-time". This informal statement can be expressed as a tensor equation as follows:

$$R^k{}_{kij} + \frac{1}{2} g_{ij} g'^{lm} R^n{}_{nlm} = \kappa T_{ij} \tag{6.1}$$

In the above, $T_{ij}$ represents the contents of space-time in terms of energy, momentum, pressure, etc. depending on the components of the tensor. The gravitational constant is $\kappa$.[22] The tensor $R^l{}_{ijk}$ captures the curvature properties of space-time, and its value solely depends on the metric, as we will see below. Thus, solving the equation for a given $T_{ij}$ amounts to finding a suitable metric. Given such a solution, we can compute the expression for the left-hand-side of the equation and verify that it is equal to the right-hand side. We do so for the example of a point mass in Section 6.2. Before that, we discuss in more detail $R^l{}_{ijk}$, the Riemann curvature tensor.

[22] We omit the cosmological term from the equation because it does not bring any more insight for our purposes.

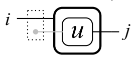
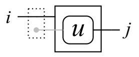
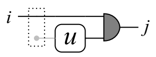
**Definition 6** (Riemann curvature). *The Riemann curvature is a 4-tensor, given by the following identity:* $R^l{}_{ijk} = \partial_i \Gamma_{jk}{}^l - \partial_j \Gamma_{ik}{}^l + \Gamma_{im}{}^l \Gamma_{jk}{}^m - \Gamma_{jn}{}^l \Gamma_{ik}{}^n$.

Each pair of terms is the antisymmetric part of a 4-tensor. Taking advantage of this property, we can make the diagram notation a sum of two terms:  $+$ .
In ROGER, it is even possible to factor the antisymmetrisation operator and obtain the following concise expression: $(\partial\,\Gamma + \Gamma \circ (\mathrm{id} \otimes \Gamma)) \circ \alpha \circ ((\mathrm{id} - \sigma) \otimes \mathrm{id})$. Despite its concision, this form obscures which index plays which role, and thus is rarely found in the literature. The above definition can be encoded directly in ALBERT as follows:

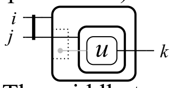
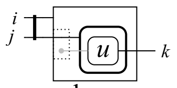
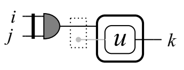
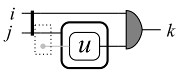
```
curvature :: CoordinateCategory z ⇒ P z r T*_z ⊸ P z r T_z ⊸ P z r T_z ⊸ P z r T_z ⊸ R z r
curvature^l_k i j
  = plus (λ a → case a of
    False → minus (λ b → case b of
      False → partial_i (christoffel_j k ^l)
      True  → partial_j (christoffel_i k ^l))
    True  → contract (λ ^m _m → minus (λ b → case b of
      False → christoffel_i m ^l ⋆ christoffel_j k ^m
      True  → christoffel_j m ^l ⋆ christoffel_i k ^m)))
```

Unfortunately, the operands of each addition must be written in the branch of a case expression, making the expression verbose as a whole. Even though it is defined in terms of Christoffel symbols, the Riemann curvature (as an algebraic object) does not depend on the choice of coordinates. This is a consequence of Theorem 1.

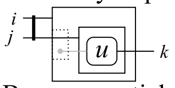
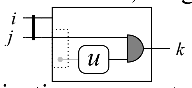
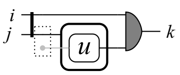
**Theorem 1** (Ricci identity). *For every covector field* u, $\nabla_i \nabla_j u^k - \nabla_j \nabla_i u^k = R^k{}_{ijl} u^l$

**Proof** We carry out the proof using the diagram notation. To be sure, we don't claim that the diagrammatic proof is novel, it is a mere illustration. But we feel it is a good example of using the DSLs: all the steps are defined, type checked, and diagrams are rendered with our library. We note first the following lemma:  $=$  $+$ 
which is an instance of Equation (5.5).

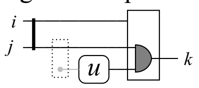
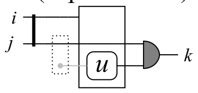
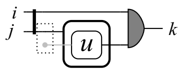
We then compute symbolically the left-hand-side in the theorem's statement, starting with the expansion of the outer derivative into partial derivatives and affinity terms (Equation 5.5):

 $=$  $-$  $+$ 

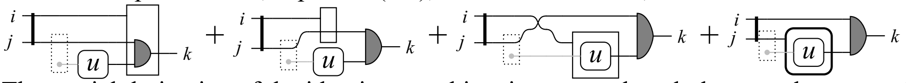
The middle term is zero, by symmetry of Γ. Expanding the first term using the lemma and linearity of partial derivatives, we get

 $+$  $+$ 

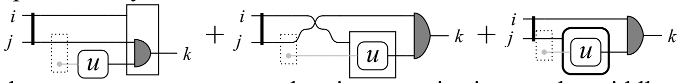
Because partial derivatives commute, the first term is zero. We expand the middle term using the composition rule (Equation 5.2) for partial derivatives:

 $+$  $+$ 

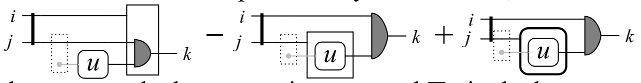
then use the product law, Equation (5.3), on the middle term, and obtain



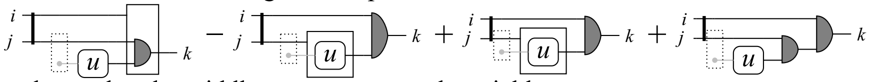
The partial derivative of the identity morphism is zero so the whole second term can be simplified away



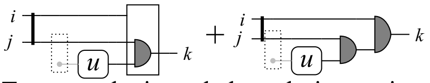
then we commute swap and antisymmetrisation, so the middle term changes sign:



then we use the lemma again to expand $\nabla u$ in the last term



and note that the middle two terms cancel to yield:



To get to the intended result, it remains to use the structural laws of SMCs. ■

## 6.1 *Code for Equation (6.1)*

A contraction of the Riemann curvature occurs twice in Equation (6.1), and as such deserves to be extracted as an intermediate definition. It is called the Ricci tensor.

$$\begin{aligned}
&\mathsf{ricci}, \mathsf{grLhs} :: (\mathsf{Additive}\, z, \mathsf{CoordinateCategory}\, z) \Rightarrow \mathsf{P}\, z\, \mathsf{r}\, T_z \multimap \mathsf{P}\, z\, \mathsf{r}\, T_z \multimap \mathsf{R}\, z\, \mathsf{r}\\
&\mathsf{ricci}_{\mathsf{j}\,\mathsf{k}} = \mathsf{contract}\, (\lambda\, {}^{\mathsf{i}}{}_{\mathsf{i}} \rightarrow \mathsf{curvature}^{\mathsf{i}}{}_{\mathsf{j}\,\mathsf{i}\,\mathsf{k}})
\end{aligned}$$

This tensor can be contracted one more time to obtain what is called the scalar (or Gaussian) curvature:

$$\begin{aligned}
&\mathsf{gaussian} :: (\mathsf{Additive}\, z, \mathsf{CoordinateCategory}\, z) \Rightarrow \mathsf{R}\, z\, \mathsf{r}\\
&\mathsf{gaussian} = \mathsf{constant}\, (1\, /\, 2) \star \mathsf{contract}\, (\lambda\, {}^{\mathsf{i}}{}_{\mathsf{i}} \rightarrow \mathsf{ricci}_{\mathsf{i}}\, (\mathsf{lower}^{\mathsf{i}}))
\end{aligned}$$

From there, the left-hand-side of Equation (6.1) is defined as follows:

$$\begin{aligned}
&\mathsf{grLhs}_{\mathsf{i}\,\mathsf{j}} = \mathsf{plus}\, (\lambda\, \mathsf{c} \rightarrow \mathbf{case}\, \mathsf{c}\, \mathbf{of}\\
&\quad \mathsf{False} \rightarrow \mathsf{ricci}_{\mathsf{i}\,\mathsf{j}}\\
&\quad \mathsf{True}\ \ \rightarrow \mathsf{gaussian} \star \mathsf{metric}_{\mathsf{i}\,\mathsf{j}})
\end{aligned}$$

We can then convert it to a morphism:

$$\begin{aligned}
&\mathsf{grLhsM} :: (\mathsf{Additive}\, z, \mathsf{CoordinateCategory}\, z) \Rightarrow (T_z \otimes T_z) \overset{z}{\rightsquigarrow} \mathbf{1}\\
&\mathsf{grLhsM} = \mathsf{tensorEval}_1\, (\lambda\, \mathsf{k} \rightarrow \mathsf{split}\, \mathsf{k}\, \&\, \lambda\, ({}_{\mathsf{i}}, {}_{\mathsf{j}}) \rightarrow \mathsf{grLhs}_{\mathsf{i}\,\mathsf{j}})
\end{aligned}$$

In the above, the operator (&) is the linear post-fix application:

$$\begin{aligned}
&(\&) :: \mathsf{a} \multimap (\mathsf{a} \multimap \mathsf{b}) \multimap \mathsf{b}\\
&\mathsf{x}\, \&\, \mathsf{f} = \mathsf{f}\, \mathsf{x}
\end{aligned}$$

### *6.2 Point-mass example (the Schwarzschild metric)*

To complete our test case for ALBERT, in this section, we define the metric which describes the gravitational effects of a point-sized mass: the Schwarzschild metric. We then verify that this metric satisfies of Equation (6.1) by evaluating both sides and checking that they match.

The first step is to define the coordinate system. Roughly speaking, Schwarzschild coordinates are spherical coordinates with an extra component for time.

```
data Spherical = Time | Rho | Theta | Phi
```

The point mass will be located at the origin ($\rho = 0$) at every point in time. In other words, the mass is at rest in this coordinate system. Hence, because we have a point-mass, the $T_{ij}$ tensor is zero everywhere except at the origin, where it is infinite.

As for all coordinate representations, the compact-closed category structure for Matrix Spherical falls out directly from Section 3.1. Likewise for the partial derivative. The only missing piece of the structure is the metric. The Schwarzschild metric is defined in terms of the considered mass $M$ or alternatively by the Schwarzschild radius $r_s$, the two being connected by the equation $r_s = \kappa M c^2 / 4\pi$. We use the parameter $r_s$ in the rest of the section. The metric is then given by tabulating the following function as a matrix.

```
schwarzschild :: Spherical → Spherical → S_Spherical
schwarzschild Time  Time  = − (1 − r_s / rho) ∗ (c ^ 2)
schwarzschild Rho   Rho   =   (1 − r_s / rho) ^ (− 1)
schwarzschild Theta Theta = rho ^ 2
schwarzschild Phi   Phi   = (rho ∗ sin theta) ^ 2
schwarzschild _     _     = 0
rho   = variable Rho
theta = variable Theta
```

We refer the reader to a course in general relativity for the physical meaning of it. We can then define the Matrix Spherical instance as follows:

```
instance MetricCategory M_Spherical where
  type T_{M_Spherical} = Atom Spherical
  g = Tab (λ (Atom i, Atom j) _ → schwarzschild i j)
```

At this point, we can directly evaluate grLhsM with $z =$ Matrix Spherical, and obtain a 4 by 4 matrix of symbolic expressions depending on Spherical coordinate variables. We find that it simplifies to zero everywhere it is defined.[23] Thus, we can verify that the Schwarzschild metric satisfies the general relativity equation.

[23] The metric has two singularities, one at the origin and one at $\rho = r_s$, the event horizon. It satisfies Equation (6.1) everywhere else.

## 7 Implementation of Albert

In this section, we explain the implementation of the combinators of ALBERT (Figures 9 and 10). As outlined in Section 1.1 and in first approximation, it is provided by the library for symmetric monoidal categories of Bernardy & Spiwack (2021). The key idea is that every linear function $\forall \mathsf{r}.\, \mathsf{P}\, z\, \mathsf{r}\, \mathsf{a} \multimap \mathsf{P}\, z\, \mathsf{r}\, \mathsf{b}$ can be converted (naturally) to a morphism of type $\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}$, and back:

$$\mathsf{encode} :: \mathsf{SymmetricMonoidal}\, z \Rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \rightarrow (\forall \mathsf{r}.\, \mathsf{P}\, z\, \mathsf{r}\, \mathsf{a} \multimap \mathsf{P}\, z\, \mathsf{r}\, \mathsf{b})$$
$$\mathsf{decode} :: \mathsf{SymmetricMonoidal}\, z \Rightarrow (\forall \mathsf{r}.\, \mathsf{P}\, z\, \mathsf{r}\, \mathsf{a} \multimap \mathsf{P}\, z\, \mathsf{r}\, \mathsf{b}) \rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b})$$

In fact, any SMC $z$ is isomorphic to the category of Haskell linear functions between corresponding ports (whose hom-set family is $\mathsf{Hom}\,(\mathsf{a}, \mathsf{b}) = \forall \mathsf{r}.\, \mathsf{P}\, z\, \mathsf{r}\, \mathsf{a} \multimap \mathsf{P}\, z\, \mathsf{r}\, \mathsf{b}$). This works even if $z$ has additional structure, and in particular if it is a tensor category.

So, if we were to choose $\mathsf{R}\, z\, \mathsf{r}$ to be $\mathsf{P}\, z\, \mathsf{r}\, \mathbf{1}$, the encode and decode functions would take care of most of the respective tasks of tensorEmbed and tensorEval. It would remain to dualise the codomain and apply unitors appropriately, as shown in Figure 4(a).

With this simple setup, we can implement the embedding of tensors (including $\delta$), as well as multiplication. Unfortunately, there are several complications.

### *7.1 Complication: Supporting derivatives*

The main issue with the above implementation sketch arises when trying to implement the covariant derivative of a tensor expression $\mathsf{t}$ as a sum of its partial derivative and affinity terms. Indeed, the affinity terms of a tensor expression $\mathsf{t}$ depends on the number (and type) of free index variables in it. But this information is not made available by the library of Bernardy & Spiwack (2021) which we base our work upon. Simply put, the type $\mathsf{P}\, z\, \mathsf{r}\, \mathbf{1}$ is opaque. To explain the solution that we employ, we need to first peek inside the implementation of this library.

Its principle is that a port (in our application, an index) is represented as a morphism in the (free) Cartesian extension of the underlying category $z$:

$$\mathbf{data}\ \mathsf{P}\, z\, \mathsf{r}\, \mathsf{a}\ \mathbf{where}\ \mathsf{P} :: \mathsf{CartesianExt}\, z\, \mathsf{r}\, \mathsf{a} \rightarrow \mathsf{P}\, z\, \mathsf{r}\, \mathsf{a}$$

The encode function simply embeds the morphism in the Cartesian extension. The decode function takes a linear function $\mathsf{f} : \mathsf{P}\, z\, \mathsf{r}\, \mathsf{a} \multimap \mathsf{P}\, z\, \mathsf{r}\, \mathsf{b}$ and applies it to the identity morphism $(\mathsf{id} : \mathsf{P}\, z\, \mathsf{a}\, \mathsf{a})$, obtaining $\mathsf{f}\,(\mathsf{P}\ \mathsf{id}) : \mathsf{CartesianExt}\, z\, \mathsf{a}\, \mathsf{b}$. The non-obvious property proven by Bernardy & Spiwack (2021) is that, because $\mathsf{f}$ is linear, $\mathsf{f}\,(\mathsf{P}\ \mathsf{id})$ is always equivalent to a morphism in the underlying category $z$; it never needs to refer to projections or duplications. The function which recovers this morphism has the following signature, but remember that it will crash if the input morphism makes essential use of the Cartesian structure:

$$\mathsf{cartesianToMonoidal} :: \mathsf{CartesianExt}\, z\, \mathsf{a}\, \mathsf{b} \rightarrow \mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}$$

To be able to support tensor derivatives, we cannot rely on decode, and we will have to access the internal function cartesianToMonoidal directly.

With this in mind, we can solve our problem, namely finding an implementation for $\mathsf{R}\,z\,\mathsf{r}$ which lets us track free variables. We do this by embedding a pair of morphisms $\mathsf{t} : \mathsf{x} \overset{z}{\rightsquigarrow} \mathbf{1}$ and $\mathsf{p} : \mathsf{CartesianExt}\,z\,\mathsf{r}\,\mathsf{x}$ (instead of just p). Note that the type x is existentially bound, not a parameter to R.

$$\textbf{data}\ \mathsf{R}\,z\,\mathsf{r}\ \textbf{where}\ \mathsf{Compose} :: (\mathsf{x} \overset{z}{\rightsquigarrow} \mathbf{1}) \to \mathsf{CartesianExt}\,z\,\mathsf{r}\,\mathsf{x} \to \mathsf{R}\,z\,\mathsf{r}$$

Here, x is the context used by the tensor t, without any spurious component. The tensor t carries the payload of the expression. Because it inhabits the non-Cartesian category $z$, we know that it uses the whole context x. Even though the type declares that the morphism p is built from the Cartesian extension of $z$, the implementation will arrange that it will only use the sub-category of $z$ whose derivative is zero. This is enforced by hiding the encode function on $\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a}$ from the user: we only provide the raise and lower functions to manipulate this type, and they are safe because $\nabla g = 0$:

$$\begin{aligned} &\mathsf{raise}\ \ = \mathsf{encode}\,\mathsf{juggleUp}\\ &\mathsf{lower} = \mathsf{encode}\,\mathsf{juggleDown} \end{aligned}$$

With this setup the derivative can be computed correctly on a pair $\mathsf{Compose}\,\mathsf{t}\,\mathsf{p}$. Indeed, the morphism-level derivative $(\nabla :: (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \to ((\mathsf{v} \otimes \mathsf{a}) \overset{z}{\rightsquigarrow} \mathsf{b}))$ need only be applied to the t component of the pair. As for the p component of the pair, the use of the Cartesian fork operator ($\triangle$) ensures that the new index is passed to the right component of $\nabla\,\mathsf{t}$.

$$\begin{aligned} &\mathsf{deriv} :: (\mathsf{ConnectionCategory}\,z) \Rightarrow \mathsf{P}\,z\,\mathsf{r}\,\mathsf{v} \multimap \mathsf{R}\,z\,\mathsf{r} \multimap \mathsf{R}\,z\,\mathsf{r}\\ &\mathsf{deriv}\,(\mathsf{P}\,\mathsf{i})\,(\mathsf{Compose}\,\mathsf{t}\,\mathsf{p}) = \mathsf{Compose}\,(\nabla\,\mathsf{t})\,(\mathsf{i} \triangle \mathsf{p}) \end{aligned}$$

$$(\triangle) :: (\mathsf{Cartesian}\,z) \Rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \to (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{c}) \to \mathsf{a} \overset{z}{\rightsquigarrow} (\mathsf{b} \otimes \mathsf{c})$$

The transcoding functions between ALBERT and its categorical semantics then become:

$$\begin{aligned} &\mathsf{embedTensor}_1 :: (\mathsf{a} \overset{z}{\rightsquigarrow} \mathbf{1}) \to \mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{R}\,z\,\mathsf{r}\\ &\mathsf{embedTensor}_1\,\mathsf{t}\,\mathsf{p} = \mathsf{Compose}\,\mathsf{t}\,\mathsf{p} \end{aligned}$$

$$\begin{aligned} &\mathsf{evalTensor}_1 :: (\mathsf{SymmetricMonoidal}\,z) \Rightarrow (\forall\,\mathsf{r}.\,\mathsf{P}\,z\,\mathsf{r}\,\mathsf{a} \multimap \mathsf{R}\,z\,\mathsf{r}) \to \mathsf{a} \overset{z}{\rightsquigarrow} \mathbf{1}\\ &\mathsf{evalTensor}_1\,\mathsf{f} = \mathsf{decoder}\,(\mathsf{f}\,(\mathsf{P}\,\mathsf{id}))\ \textbf{where}\\ &\quad \mathsf{decoder} :: (\mathsf{SymmetricMonoidal}\,z) \Rightarrow \mathsf{R}\,z\,\mathsf{a} \to \mathsf{a} \overset{z}{\rightsquigarrow} \mathbf{1}\\ &\quad \mathsf{decoder}\,(\mathsf{Compose}\,\mathsf{t}\,\mathsf{p}) = \mathsf{t} \circ \mathsf{cartesianToMonoidal}\,\mathsf{p} \end{aligned}$$

Multiplication remains straightforward to implement:

$$\mathsf{Compose}\,\mathsf{t}_1\,\mathsf{p}_1 \star \mathsf{Compose}\,\mathsf{t}_2\,\mathsf{p}_2 = \mathsf{Compose}\,(\bar{\rho} \circ (\mathsf{t}_1 \otimes \mathsf{t}_2))\,(\mathsf{p}_1 \triangle \mathsf{p}_2)$$

## *7.2 Complication: Supporting addition of tensors*

The type of the addition operator ($\mathsf{plus} :: (\mathsf{Bool} \rightarrow \mathsf{R}\,z\,\mathsf{r}) \multimap \mathsf{R}\,z\,\mathsf{r}$) poses a problem for the above implementation. The issue is that the semantics of plus f depends on both f False and f True. However, plus is linear in f, and thus we can call f only once in the implementation of plus. The way to work around this problem is to embed a non-linear implementation inside a linear datatype. That is, it suffices to add a *constructor* $\mathsf{Plus} :: (\mathsf{Bool} \rightarrow \mathsf{R}\,z\,\mathsf{r}) \multimap \mathsf{R}\,z\,\mathsf{r}$ to the R datatype. Operationally, if f is the argument to plus, it is *stored once* in the R data structure, and it is only at the moment of evaluating plus f to a morphism that we invoke f. But, at the evaluation point, there is no linearity restriction on the tensor expression of type $\mathsf{R}\,z\,\mathsf{r}$: it can be used an arbitrary number of times, and, according to the typing rules of Linear Haskell, it means that its payload (f) has itself no usage restrictions. The fact that the evaluation function has no linearity restriction on its tensor expression argument (t) means that t cannot have any free index variables in it. At first sight, this is a drawback. However, there is a good reason for restricting evaluation to closed tensor expressions: it ensures that index variables do not escape the scope of the tensor expression where they belong.

The rest of the implementation of ALBERT needs to be modified to support this new constructor, by defining a case for sums. Fortunately, such a case is never difficult to handle: we simply distribute every operation over the operands of the sum.

## *7.3 Complication: Contraction*

Unfortunately, our definition for R so far does not in fact support contraction. This is because all indices are inputs of the expression, and to connect inputs together we need to use the $\eta$ combinator. To do so, we need an additional **1** input—but no such input is explicitly available in the R type. The workaround for the problem is to add such a unit input explicitly. Hence, the final implementation of the R type is

$$
\begin{array}{l}
\textbf{data}\ \mathsf{S}\,z\,\mathsf{r}\ \textbf{where} \\
\quad \mathsf{Compose} :: (\mathsf{x} \overset{z}{\rightsquigarrow} \mathbf{1}) \rightarrow \mathsf{CartesianExt}\,z\,\mathsf{r}\,\mathsf{x} \rightarrow \mathsf{S}\,z\,\mathsf{r} \\
\quad \mathsf{Plus} :: (\mathsf{Bool} \rightarrow \mathsf{S}\,z\,\mathsf{r}) \multimap \mathsf{S}\,z\,\mathsf{r} \\
\textbf{type}\ \mathsf{R}\,z\,\mathsf{r} = \mathsf{P}\,z\,\mathsf{r}\,\mathbf{1} \multimap \mathsf{S}\,z\,\mathsf{r}
\end{array}
$$

The implementation of contract is then:

$$\mathsf{contract}\ \mathsf{f}\ \mathsf{u} = \mathsf{uncurry}\,(\mathsf{uncurry}\ \mathsf{f})\,(\mathsf{encode}\,((\eta \otimes \mathsf{id}) \circ \rho)\,\mathsf{u})$$

The other combinators can simply ignore or thread this unit input. The core of the final implementation is shown in Figure 11.

**– – Part of the implementation borrowed from Bernardy & Spiwack (2021) – –**

```
$\mathsf{cartesianToMonoidal} :: \mathsf{CartesianExt}\ z\ \mathsf{a}\ \mathsf{b} \to \mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}$
$\mathsf{encode} :: \mathsf{SymmetricMonoidal}\ z \Rightarrow (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b}) \to (\forall\, \mathsf{r}.\, \mathsf{P}\ z\ \mathsf{r}\ \mathsf{a} \multimap \mathsf{P}\ z\ \mathsf{r}\ \mathsf{b})$
$\mathsf{decode} :: \mathsf{SymmetricMonoidal}\ z \Rightarrow (\forall\, \mathsf{r}.\, \mathsf{P}\ z\ \mathsf{r}\ \mathsf{a} \multimap \mathsf{P}\ z\ \mathsf{r}\ \mathsf{b}) \to (\mathsf{a} \overset{z}{\rightsquigarrow} \mathsf{b})$
$\mathbf{data}\ \mathsf{P}\ z\ \mathsf{r}\ \mathsf{a}\ \mathbf{where}\ \mathsf{P} :: \mathsf{CartesianExt}\ z\ \mathsf{r}\ \mathsf{a} \to \mathsf{P}\ z\ \mathsf{r}\ \mathsf{a}$
$\mathsf{curry} :: (\mathsf{SymmetricMonoidal}\ z) \Rightarrow (\mathsf{P}\ z\ \mathsf{r}\ (\mathsf{a} \otimes \mathsf{b}) \multimap \mathsf{k}) \multimap (\mathsf{P}\ z\ \mathsf{r}\ \mathsf{a} \multimap \mathsf{P}\ z\ \mathsf{r}\ \mathsf{b} \multimap \mathsf{k})$
$\mathsf{curry\ f\ a\ b} = \mathsf{f\ (merge\ (a, b))}$
$\mathsf{uncurry} :: (\mathsf{SymmetricMonoidal}\ z) \Rightarrow (\mathsf{P}\ z\ \mathsf{r}\ \mathsf{a} \multimap \mathsf{P}\ z\ \mathsf{r}\ \mathsf{b} \multimap \mathsf{k}) \multimap (\mathsf{P}\ z\ \mathsf{r}\ (\mathsf{a} \otimes \mathsf{b}) \multimap \mathsf{k})$
$\mathsf{uncurry\ f\ p} = \mathbf{case}\ \mathsf{split\ p}\ \mathbf{of}\ \mathsf{(a, b)} \to \mathsf{f\ a\ b}$
```

**– – Part of the implementation specific to the present work – –**

```
$\mathsf{zeroTensor} = \mathsf{tensorEmbed}_1\ 0$
$\mathsf{constant\ s\ (P\ u)} = \mathsf{Compose\ (s} \triangleleft \mathsf{id)\ u}$
$\mathsf{plus\ f\ u} = \mathsf{Plus}\ (\lambda\, \mathsf{b} \to \mathsf{f\ b\ u})$
$(\mathsf{f} \star \mathsf{g})\ \mathsf{u} = \mathsf{dupU\ u}\ \&\ \lambda\, (\mathsf{u}_1, \mathsf{u}_2) \to \mathsf{f\ u}_1 \star\star \mathsf{g\ u}_2\ \mathbf{where}$
   $(\star\star) :: (\mathsf{SymmetricMonoidal}\ z) \Rightarrow \mathsf{S}\ z\ \mathsf{r} \multimap \mathsf{S}\ z\ \mathsf{r} \multimap \mathsf{S}\ z\ \mathsf{r}$
   $\mathsf{Compose\ t}_1\ \mathsf{p}_1 \star\star \mathsf{Compose\ t}_2\ \mathsf{p}_2 = \mathsf{Compose}\ (\bar{\rho} \circ (\mathsf{t}_1 \otimes \mathsf{t}_2))\ (\mathsf{p}_1 \vartriangle \mathsf{p}_2)$
   $\mathsf{Plus\ f}$          $\star\star\ \mathsf{t}$                 $= \mathsf{Plus}\ (\lambda\, \mathsf{c} \to \mathsf{f\ c} \star\star \mathsf{t})$
   $\mathsf{t}$               $\star\star\ \mathsf{Plus\ f}$            $= \mathsf{Plus}\ (\lambda\, \mathsf{c} \to \mathsf{t} \star\star \mathsf{f\ c})$
$\mathsf{deriv\ i\ r\ u} = \mathsf{deriv'\ i\ (r\ u)}\ \mathbf{where}$
   $\mathsf{deriv'} :: (\mathsf{ConnectionCategory}\ z) \Rightarrow \mathsf{P}\ z\ \mathsf{r}\ T_z \multimap \mathsf{S}\ z\ \mathsf{r} \multimap \mathsf{S}\ z\ \mathsf{r}$
   $\mathsf{deriv'\ (P\ i)}$     $\mathsf{(Compose\ t\ p)} = \mathsf{Compose}\ (\nabla\, \mathsf{t})\ (\mathsf{i} \vartriangle \mathsf{p})$
   $\mathsf{deriv'\ p}$         $\mathsf{(Plus\ f)}$          $= \mathsf{Plus}\ (\lambda\, \mathsf{c} \to \mathsf{deriv'\ p\ (f\ c)})$
$\mathsf{tensorEval}_1\ \mathsf{f} = \mathsf{tensorEval'}_0\ (\mathsf{uncurry\ f\ (P\ (FreeCart.Embed}\ \rho)))\ \mathbf{where}$
   $\mathsf{tensorEval'}_0 :: (\mathsf{SymmetricMonoidal}\ z, \mathsf{Additive}\ z) \Rightarrow \mathsf{S}\ z\ \mathsf{a} \to \mathsf{a} \overset{z}{\rightsquigarrow} \mathbf{1}$
   $\mathsf{tensorEval'}_0\ \mathsf{u} = \mathbf{case}\ \mathsf{u}\ \mathbf{of}$
      $\mathsf{(Compose\ t\ p)} \to \mathsf{t} \circ \mathsf{cartesianToMonoidal\ p}$
      $\mathsf{Plus\ f}$            $\to \mathsf{tensorEval'}_0\ \mathsf{(f\ False)} + \mathsf{tensorEval'}_0\ \mathsf{(f\ True)}$
$\mathsf{tensorEval\ f} = \bar{\rho} \circ \sigma \circ (\mathsf{tensorEval}_1\ (\mathsf{Interface.uncurry\ f}) \otimes \mathsf{id}) \circ \bar{\alpha} \circ (\mathsf{id} \otimes \eta) \circ \rho$
$\mathsf{tensorEmbed}_1\ \mathsf{f\ (P\ q)\ (P\ u)} = \mathsf{Compose}\ (\mathsf{f} \circ \bar{\rho})\ (\mathsf{q} \vartriangle \mathsf{u})$
$\mathsf{tensorEmbed\ t\ i\ j} = \mathsf{tensorEmbed}_1\ (\epsilon \circ (\mathsf{t} \otimes \mathsf{id}))\ (\mathsf{merge\ (i, j)})$
$\mathsf{delta\ i\ j\ u} = \mathsf{eatU\ u\ (delta'\ i\ j)}\ \mathbf{where}$
   $\mathsf{delta'} :: \mathsf{CompactClosed}\ z \Rightarrow \mathsf{P}\ z\ \mathsf{r}\ \mathsf{a} \multimap \mathsf{P}\ z\ \mathsf{r}\ \mathsf{a}^* \multimap \mathsf{S}\ z\ \mathsf{r}$
   $\mathsf{delta'\ (P\ i)\ (P\ j)} = \mathsf{Compose}\ \epsilon\ (\mathsf{i} \vartriangle \mathsf{j})$
$\mathsf{eatU} :: (\mathsf{SymmetricMonoidal}\ z) \Rightarrow \mathsf{P}\ z\ \mathsf{r}\ \mathbf{1} \multimap \mathsf{S}\ z\ \mathsf{r} \multimap \mathsf{S}\ z\ \mathsf{r}$
$\mathsf{eatU\ (P\ f)\ (Compose\ t\ p)} = \mathsf{Compose}\ (\mathsf{t} \circ \bar{\rho})\ (\mathsf{p} \vartriangle \mathsf{f})$
$\mathsf{eatU\ p\ (Plus\ f)} = \mathsf{Plus}\ (\lambda\, \mathsf{b} \to \mathsf{eatU\ p\ (f\ b)})$
$\mathsf{dupU} :: \mathsf{SymmetricMonoidal}\ z \Rightarrow \mathsf{P}\ z\ \mathsf{r}\ \mathbf{1} \multimap (\mathsf{P}\ z\ \mathsf{r}\ \mathbf{1}, \mathsf{P}\ z\ \mathsf{r}\ \mathbf{1})$
$\mathsf{dupU} = \mathsf{split} \circ (\mathsf{encode}\ \rho)$

$\mathsf{raise}\ \ = \mathsf{encode\ juggleUp}$
$\mathsf{lower} = \mathsf{encode\ juggleDown}$
```

Fig. 11: Final implementation

## 8 Related work

### *8.1 Tensor presentations in introductory texts*

We expect many of our readers not to be already familiar with tensors, and therefore they will need to read pedagogical introductions to the topic, as the authors did when preparing this paper. We believe that a warning is in order, because, typically, introductory texts lean heavily on the representations of tensors as (generalised) arrays of coefficients. Consequently, undue importance is enthused to what happens under change of coordinates in the manifold—even though from an algebraic perspective, coordinate systems do not even enter the picture. All the introductory texts that we could find take this approach (Dullemond & Peeters, 2010; Fleisch, 2011; Grinfeld, 2013; Porat, 2014; Rowland & Weisstein, 2023). The quotes below are by no means intended as singling out particular authors: this approach is pervasive. These are the kinds of definition that we find:

*[Tensors] are geometrical objects over vector spaces, whose coordinates obey certain laws of transformation under change of basis.* *(Porat, 2014)*

*A tensor of rank n is an array of $3^n$ values (in 3-D space) called "tensor components" that combine with multiple directional indicators (basis vectors) to form a quantity that does not vary as the coordinate system is changed.* *(Fleisch, 2011)*

*An nth-rank tensor in m-dimensional space is a mathematical object that has n indices and $m^n$ components and obeys certain transformation rules.* *(Rowland & Weisstein, 2023)*

At the end of the document one finds out that the transformation rules are the multiplication by Jacobians. Other sources take a more pedagogical approach and start with vectors and covectors:

*$y^\alpha$ : contravariant vector* *(Dullemond & Peeters, 2010, p. 13)*

Here $y$ is considered to be some array of numbers, enthused with a property (covariance), which refers to the fact that the inverse of the Jacobian should multiply the coefficients when changing coordinate systems. (Furthermore, $\alpha$ is implicitly lambda bound here—a kind of liberty that most textbooks take in all areas of mathematics.)

For the student already familiar with linear algebra, this can be particularly confusing, because *every* vector or matrix transforms this way. So why bother highlighting this property? This presentation continues with

*The object $g_{\mu\nu}$ is a kind of tensor that is neither a matrix nor a vector or covector.* *(Dullemond & Peeters, 2010, p. 17)*

In algebraic terms, it is not hard to state that the metric is a linear function of two arguments, but the representational approaches incites the authors to beat around the bush. Besides, the representational approaches defines the metric as the Gram matrix of the base vectors, and it being a tensor requires some proof.

In the algebraic definition this kind of pitfall is avoided. Because of the awkward definition of tensors in the representational view, there is much discussion about what is and what isn't a tensor. One can find statements like the following:

*$v^\mu + w_\mu$ is not a tensor.* *(Dullemond & Peeters, 2010, p. 30)*

Algebraically, this is attempting to add objects of different type together. In ALBERT, the above expression is not well-typed either. (And yet, to confuse matters even more, this expression *can* be made sense of with the pervasive index juggling conventions.)

But there is a more subtle way in which something might be "not a tensor":

*The Christoffel symbol is not a tensor because it contains all the information about the curvature of the coordinate system and can therefore be transformed entirely to zero if the coordinates are straightened. Nevertheless we treat it as any ordinary tensor in terms of the index notation.* *(Dullemond & Peeters, 2010, p. 36)*

We believe that this kind of statement is puzzling to the novice. Indeed, the first sentence states that because $\Gamma$ (the Christoffel symbol) is zero in some bases, then it cannot be a tensor (implicit to the above is that no Jacobian can transform a zero tensor to a nonzero tensor). The student might still wonder if this statement applies for a manifold that cannot be straightened (i.e. one that is not flat), and why it can be handled like a tensor anyway.

In the algebraic view that we employ, $\Gamma$ is a morphism like any other. So is it a tensor after all? In fact, the value of $\Gamma$ for any given basis is a tensor. But $\Gamma$ is defined as an expression which explicitly references the basis, and changes with it. The confusing aspect is that the coefficient representation of *any* tensor changes when the basis changes. But $\Gamma$ changes *as a geometric object* as the coordinate system changes, which means that it cannot be converted to another basis by the usual jacobian-based transformation while retaining the properties of the Christoffel symbol.

What we find puzzling is that the choice of the representational approach appears to be a conscious one:

*[We] have used the coordinate approach to tensors, as opposed to the formal geometrical approach. Although this approach is a bit old fashioned, I still find it the easier to comprehend on first learning, especially if the learner is not a student of mathematics or physics.* *(Porat, 2014)*

It is not to say that the algebraic approach is absent from the literature. While it appears to be chiefly geared towards specialists, it also can be found in textbooks, but we could not find one that explicitly makes the link between all notations. For instance, Bowen & Wang (1976) take an algebraic approach, and as such only manipulate the category-oriented (point-free) notations. Using only this notation is sufficient for them because they do not manipulate any complicated tensor expressions, but it also means that readers will have a hard time connecting other notations to the point-free language. Jeevanjee (2011) does better by describing the tensor algebras, and discussing various representations. However, the equivalence between the various languages is not mentioned.

As mentioned earlier Thorne & Blandford (2015) do mention this equivalence informally. We refer the more advanced reader to Bleecker (2005) for a full development in the language of linear algebra and differential forms, without any connection to the underlying categorical structures.

### *8.2 Einstein notation*

The Einstein notation appears to arise as a generalisation of the notation to denote elements of matrices. This indexing notation is so well established that we could not trace where it originates.

However, not every expression which involves accessing indices can be mapped to a tensor. In a nutshell, indices must be treated abstractly and used linearly. Thus, under these conditions, which are exactly those of ALBERT, there is an equivalence between Einstein notation and the algebraic specification of tensors as morphisms.

Despite our best efforts, some distance remains between ALBERT and standard Einstein notation. First the Einstein notation leaves the binders of indices implicit, but because ALBERT is embedded in a functional language, every index variable must be lambda bound. In particular, contraction is written as contract $(\lambda\, {}^{\mathsf{k}}{}_{\mathsf{k}} \rightarrow \ldots)$. One could imagine a pre-processing step to shorten the notation and more closely approach the Einstein notation, but we find that sticking to the lambda calculus convention avoids a source of confusion.

Second, it should be noted that the use of standalone subscript and superscript variables is a small liberty that we took in typesetting the paper: while the Haskell compiler accepts super and subscript characters in variable names, it disallows them as the first character. Third, the operands of an addition are **case** branches. We believe that an alternative syntax should be provided by GHC, because it is generally useful in presence of linear types. Fourth, a pattern syntax for merge and split would shorten index manipulation. Pattern synonyms are already available in GHC, but are not currently compatible with linear types. This is a technical shortcoming that could be addressed in Haskell implementations.

While this paper covers all the principles of the tensor notations, we have not aimed for an exhaustive coverage. In particular, we did not discuss the Levi-Civita *tensor*, which has many uses and is related to antisymmetrisation.

### *8.3* DSLs *for array programming and scientific programming*

There is overlap between the present work and languages oriented to scientific programming: they can both be used to describe array-oriented computations. Array-oriented programming languages have a long history, perhaps starting with APL (Iverson, 1962). Notable standalone languages include Single-Assignment C (Scholz, 1994) or SISAL (Feo *et al.*, 1990) and even Matlab (Gilat, 2004). When it comes to array EDSLs, there is an abundance of libraries available. Limiting ourselves to the Haskell ecosystem, notable examples include RePa (Lippmeier *et al.*, 2010) and accelerate (Chakravarty *et al.*, 2011).

An important difference between this work and most array-oriented languages is that they focus on representations first, and semantics come only as a means to support optimisations. In contrast, our approach puts categorical semantics at the core. This means that constructing a dataflow representation (typically as a free SMC) is natural. This representation can then be optionally optimised and interpreted as operations on matrices and arrays, perhaps relying on the aforementioned libraries or a dedicated compiler (Kjolstad *et al.*, 2017) as backend.

The implementation model that we suggest is to construct a dataflow graph (typically as a free categorical representation), optimise it and then interpret it as operations on matrices. This kind of model was already at the heart of SISAL, but has been popularised recently in machine-learning applications by the TensorFlow library (Abadi *et al*., 2016). We note, however, that TensorFlow (and similar machine learning packages) do not offer an index-based notation.[24] Instead, the programmer must keep track of dimensions using indices. The situation is similar to de Bruijn index representations for lambda terms. Our work, on the other hand, rejects non-tensor (non-linear) primitives, and as such would preclude many useful array operations. These would need to be added as an additional layer.

Another salient feature of the present work is its specific ability to support calculus. Most scientific programming languages provide arrays, but let the user figure out how to model tensor calculus operations. However, there are exceptions. Sussman and Wisdom (2013) develop a lisp-based DSL for differential geometry. It is a point-free language close in spirit to ROGER, even though the categorical structures remain implicit. Diderot (Chiw *et al*., 2012) is specifically oriented towards tensor calculus, with explicit support for indices, as in Einstein notation. Cadabra (Peeters, 2006) is another computer-algebra system supporting tensor calculus expressions.

## *8.4 Categorical semantics*

It is generally more convenient to provide instances of categorical structures rather than handling lambda terms directly. In this paper, we have used matrix and diagram instances, but many other applications exist (Elliott, 2017). A major selling point of categorical semantics is that they avoid the need of manipulating variables explicitly (perhaps as de Bruijn indices). This advantage is already identifiable in the work of Cousineau *et al.* (1985), but Elliott (2017) has shown how to leverage them in EDSLs. He does so by providing a compiler plugin that translates lambda terms of the EDSL to morphisms in a closed Cartesian category. This last characteristic means that no support for linear types or specific support of SMCs is available.

These shortcomings have been addressed by Bernardy & Spiwack (2021), who show how to evaluate linear functions to morphisms in an SMC, with no compiler modification. While they lay down the foundation for the present work, their library is unfortunately not sufficient for our purposes. The technical additions provided by this paper include the ability to represent tensor derivatives (Section 7.1), tensor addition (Section 7.2) and contraction (Section 7.3).

Categorical semantics are particularly well suited to perform automatic differentiation (AD), as Elliott (2018) has shown. We have shown that derivatives can be represented, and we expect their symbolic computation can be done with standard techniques. We have not shown how AD can be performed, but because we use categorical semantics, Elliott-style AD is the natural approach to implement our interface for derivatives. In fact, it is the method that we have used to implement the Schwarzschild metric example (Section 6.2).

[24] There is a function called einsum in both PyTorch and Tensorflow but it is not typed, and not a general syntax for the Einstein notation. It only applies to one term using contraction or shape transformation.

### *8.5 Penrose diagram notation*

The diagram notation that we have used is a (graphical) DSL in its own right. It is particularly suited for morphisms in SMCs (and their extensions; such as compact closed categories). Selinger (2011) provides a survey for various kinds of categories. The correspondence between definitional and topological equivalence is their main advantage: it means that topological intuitions can be leveraged for formal proofs (Blinn, 2002; Hinze, 2012). Kissinger (2012) even built a tool on this premise.

To our knowledge, the diagram notation was in fact first developed for tensor calculus by Penrose (1971). Later it was generalised to represent morphisms in monoidal categories (Joyal & Street, 1991). For tensors, there is a considerable amount of variation between the diagrammatic notation used by various authors. We have adhered to a standardised subset of this notation. Furthermore, we have ensured that each diagram is built strictly using well-defined building blocks, combined strictly using sequential ($\circ$) and parallel ($\otimes$) composition.

In contrast, the literature on tensor applications does not prescribe a direction for reading diagrams (whereas ours are always read left to right). As long as the various input and outputs of atomic morphisms are clearly identified, there is no problem in not specifying a direction for diagrams of compact closed categories because all directions are equivalent thanks to the snake laws.

In his seminal work, Penrose additionally does not make a graphical difference between the dual $\mathsf{a}^*$ and $\mathsf{a}$. This means, for instance, that metrics are (graphically) indistinguishable from $\eta$ and $\epsilon$. Most of the time the difference is inconsequential; however, it becomes important when expressing covariant derivatives in terms of partial derivatives and affinities. Indeed, the affinities for $\mathsf{a}$ and $\mathsf{a}^*$ are not the same.

## 9 Conclusion

We have studied three equally expressive notations for tensor algebra and tensor calculus, using the DSL methodology: a point-free morphism language (ROGER) based directly on the categorical semantics; an index-based language (ALBERT) based on the Einstein notation in wide use in the literature; and a streamlined version of the diagram notation by Penrose.

ROGER is ideal to define semantics (as instances of the categorical structures). Its main drawback is that its syntax make it difficult to track connections between components. In particular, it is difficult to recognise if two expressions are equivalent.

The diagrammatic notation is good at representing connections between building blocks of complex expressions and makes it easy to check for equivalence. Therefore, it is ideal for presenting morphisms and proofs of equivalence. Its main drawback is that it lacks an established way to deal with polymorphism.

The Einstein notation is a natural extension of matrix element notation, and its programming language counterpart is a natural extension of array indexing. Indeed its textual, compact nature makes it easy to input into computers and easy to typeset in books. The connection between the building blocks of a complex Einstein notation expression is done by inspecting indices and checking their repeated occurrences. It is not as obvious as in diagram notation, but much less arduous than in point-free notation. This notation is chosen in most of the literature on tensors, even though we find that diagram notation is often pedagogically superior.

We formalised it as an EDSL, ALBERT, where every index is represented by a (linear) lambda bound variable. Expressions in ALBERT are very close to standard Einstein notation, and they evaluate to representations in any of the instances of the abstract categorical structures. Following this workflow, one can build a program that can simultaneously be used to manipulate tensors as matrices and produce diagrams in Penrose notation for the same computations. This means that ALBERT users enjoy most of the benefits of all representations.

The connection between the categorical structures, Einstein notation and diagrams is not a new one. However, despite our best efforts, we have not seen it precisely documented anywhere before this work. In one direction, it is not difficult to see that Einstein notation is an instance of the abstract structure. However, the other direction (from Einstein notation to categories) is not so obvious. To the best of our understanding, the connection relies essentially on the isomorphism between linear functions (between indices) and morphisms in an SMC. This isomorphism is known in the functional programming community (Benton, 1995), but we could not find any presentation of tensors which points out this fact. Furthermore, on its own, this isomorphism is not sufficient to account for all aspects of tensor calculus: the semantics of sums and derivatives need careful treatment (Section 7).

Another difficulty that we faced when studying tensor calculus using mathematics textbooks is that they mix concepts and notations from the abstract algebraic level with those at representational level, without warning. This kind of freedom can be disturbing for someone used to the rigid conventions of lambda calculi and programming languages in general. We hope that this paper helps the growing crowd of (functional) programmers to approach tensor notation and its applications.

## Conflicts of Interest

None.

## References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. & Zheng, X. (2016) Tensorflow: A system for large-scale machine learning. In OSDI, pp. 265–283.

Barr, M. & Wells, C. (1999) *Category Theory for Computing Science*, 3rd ed. Prentice Hall.

Benton, P. N. (1995) A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 121–135.

Bernardy, J.-P. & Spiwack, A. (2021) Evaluating linear functions to symmetric monoidal categories. In Proceedings of the 14th ACM SIGPLAN International Haskell Symposium (Haskell'21), August 26–27, 2021, Virtual Event, Republic of Korea. ACM, pp. 17–33.

Bleecker, D. (2005) *Gauge Theory and Variational Principles*. Courier Corporation.

Blinn, J. (2002) Using tensor diagrams to represent and solve geometric problems. In Lecture Notes for a Tutorial Given at Siggraph 2002.

Bowen, R. M. & Wang, C.-C. (1976) *Introduction to Vectors and Tensors. Volume 1: Linear and Multilinear Algebra*. Springer.

Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L. & Grover, V. (2011) Accelerating Haskell array codes with multicore gpus. In Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming. ACM, pp. 3–14.

Chiw, C., Kindlmann, G. L., Reppy, J. H., Samuels, L. & Seltzer, N. (2012) Diderot: A parallel DSL for image analysis and visualization. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'12, Beijing, China, June 11–16, 2012. ACM, pp. 111–120.

Cousineau, G., Curien, P. & Mauny, M. (1985) The categorical abstract machine. In Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16–19, 1985, Proceedings. Springer, pp. 50–64.

Dullemond, K. & Peeters, K. (2010) Introduction to tensor calculus. Lecture notes available online.

Elliott, C. (2017) Compiling to categories. *Proc. ACM Program. Lang.* **1**(ICFP), 27:1–27:27.

Elliott, C. (2018) The simple essence of automatic differentiation. *Proc. ACM Program. Lang.* **2**(ICFP), 70:1–70:29.

Feo, J., Cann, D. C. & Oldehoeft, R. R. (1990) A report on the sisal language project. *J. Parallel Distributed Comput.* **10**(4), 349–366.

Fleisch, D. (2011) *A Student's Guide to Vectors and Tensors*. Cambridge University Press.

Gilat, A. (2004) *MATLAB: An Introduction with Applications*. Wiley.

Grinfeld, P. (2013) *Introduction to Tensor Analysis and the Calculus of Moving Surfaces*. Springer.

Hinze, R. (2012) Kan extensions for program optimisation or: Art and dan explain an old trick. In Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25–27, 2012. Proceedings. Springer, pp. 324–362.

Iverson, K. E. (1962) *A Programming Language*. Wiley.

Jeevanjee, N. (2011) *An Introduction to Tensors and Group Theory for Physicists*. Springer.

Joyal, A. & Street, R. (1991) The geometry of tensor calculus, i. *Adv. Math.* **88**(1), 55–112.

Kissinger, A. (2012) *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing*. PhD Thesis. University of Oxford.

Kjolstad, F., Kamil, S., Chou, S., Lugato, D. & Amarasinghe, S. P. (2017) The tensor algebra compiler. *Proc. ACM Program. Lang.* **1**(OOPSLA), 77:1–77:29.

Lippmeier, B., Keller, G., Chakravarty, M. M. T., Leshchinskiy, R. & Peyton Jones, S. L. (2010) Regular, shape-polymorphic, parallel arrays in Haskell. In ICFP, pp. 261–272.

Lovelock, D. & Rund, H. (1989) *Tensors, Differential Forms, and Variational Principles*. Courier Corporation.

Orchard, D. A. & Schrijvers, T. (2010) Haskell type constraints unleashed. In Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19–21, 2010. Proceedings. Springer, pp. 56–71.

Peeters, K. (2006) A field-theory motivated approach to symbolic computer algebra. arXiv preprint cs/0608005.

Penrose, R. (1971) Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications*, vol. 1, pp. 221–244.

Porat, B. (2014) A gentle introduction to tensors. Available from www.ese.wustl.edu/~nehorai/.

Rowland, T. & Weisstein, E. W. (2023) "tensor." from mathworld–a wolfram web resource. Available at: https://mathworld.wolfram.com/Tensor.html.

Scholz, S.-B. (1994) Single assignment c-functional programming using imperative style. In Proceedings of IFL.

Sculthorpe, N., Bracker, J., Giorgidze, G. & Gill, A. (2013) The constrained-monad problem. In ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA, September 25–27, 2013. ACM, pp. 287–298.

Selinger, P. (2011) A survey of graphical languages for monoidal categories. In *New Structures for Physics*, pp. 289–355.

Sussman, G. J. & Wisdom, J. (2013) *Functional Differential Geometry*. MIT Press.

Thorne, K. S. & Blandford, R. D. (2015) *Modern Classical Physics: Optics, Fluids, Plasmas, Elasticity, Relativity, and Statistical Physics*. Princeton University Press.