Simplified Index Rules for Integrating Tensor Index Notation into Programming Languages with Static Type Systems

Satoshi Egi

Rakuten Group, Inc.
satoshi.egi@rakuten.com

Abstract. Tensor index notation is indispensable in physics and differential geometry, but integrating it into programming languages has been difficult because conventional index rules are operator-specific and indexed expressions are often treated as special forms rather than ordinary functions. We present a type-directed approach that makes indexed tensor operations first-class, user-definable functions in a statically typed language. The approach combines simplified, unified index rules with a classification of functions into scalar and tensor functions. Scalar functions, whose parameter types do not unify with tensor types, are automatically lifted element-wise to tensor arguments; tensor functions receive tensors directly and manipulate their structure, for example in tensor contraction. This classification is realized by Hindley-Milner (HM)-based type inference with type classes, so users can define and use tensor operators without writing type annotations. Index completion rules for tensors with omitted indices reuse the same machinery for differential forms. The result is not merely shorter notation: tensor operators can be defined by users, composed with higher-order functions, and statically checked without operator-specific index transformation logic. We demonstrate the approach in the Egison programming language, where programs closely follow mathematical expressions.

Keywords: tensor calculus · tensor index notation · differential forms · type systems · Hindley-Milner type inference

1 Introduction

Mathematical notation has evolved over centuries to provide concise ways to represent complex ideas. Successful notations have been systematically incorporated into programming languages [3]. However, tensor index notation, developed by Ricci and Levi-Civita [20], remains particularly challenging to integrate due to its complex semantics and context-dependent rules.

The fundamental challenge lies in the complex and operator-specific index transformation rules. Consider tensor addition and multiplication for vectors u and v: $u_i + v_i$ produces a vector, but $u_i v_i$ is mathematically invalid; $u^i + v_i$ is invalid, while $u^i v_i$ yields the inner product; $u_i + v_j$ is invalid, but $u_i v_j$ produces the tensor product. These inconsistent rules mean each tensor operator requires its own specialized index handling logic.

Our Approach and Contributions. The central contribution of this paper is to make tensor index notation part of the ordinary function language: indexed tensor operations can be defined by users, passed to higher-order functions, and statically checked, rather than being fixed primitives or macro-expanded expression blocks. We achieve this through three mechanisms. (1) We introduce unified index rules that work consistently across all tensor operations (Table 1, right). The key design choice is to relax strict mathematical index rules while preserving all valid mathematical expressions. Expressions such as $u_i + v_j$, which standard rules reject, produce higher-rank results, but in return all index rules become uniform and require no operator-specific logic. (2) We classify all functions into scalar functions and tensor functions by HM-based type inference, without requiring type annotations. Scalar functions such as + are automatically lifted element-wise to tensors, so existing scalar code becomes reusable in tensor expressions. (3) We introduce index completion rules for tensors with omitted indices, enabling concise differential-form operators such as the wedge product, exterior derivative, and Hodge star. Together, these

Addition	Multiplication
$u_i + v_i$: vector	$u_i v_i$: invalid
$u_i + v_j$: invalid	$u_i v_j$: matrix
$u^i + v_i$: invalid	$u^i v_i$: scalar

(a) Standard index rules

Addition	Multiplication
$u_i + v_i$: vector	$u_i v_i$: vector
$u_i + v_j$: matrix	$u_i v_j$: matrix
$u^i + v_i$: matrix	$u^i v_i$: scalar

(b) Our simplified rules

Table 1. Index rules in mathematics vs. our approach

mechanisms let programs follow mathematical formulae closely while keeping tensor operators extensible and type safe.

2 Scalar and Tensor Parameters

Our approach introduces two types of function parameters: *scalar parameters*, whose types do not unify with `Tensor a`, trigger automatic element-wise application via `tensorMap` (unary) or `tensorMap2` (binary); and *tensor parameters*, whose types can unify with `Tensor a`, receive tensors directly and preserve their structure.

The type system is based on HM-based type inference [16,12], where all tensors have type `Tensor a`. Type annotations are optional—the HM-based inference algorithm infers the most general type for every expression, so the scalar/tensor classification happens transparently. Users can write tensor programs without type annotations and still benefit from static type checking:

```
-- With type annotations (optional):
def min {Ord a} (x : a) (y : a) : a := if x < y then x else y
-- Without type annotations (inferred automatically):
def min x y := if x < y then x else y
```

Scalar Functions. The `min` function exemplifies scalar behavior. Its inferred type is `{Ord a} => a -> a -> a`: Both parameters `x` and `y` have type `{Ord a} => a`. Since `Tensor` is not an instance of `Ord` (the type class for ordered types supporting comparison), this type does not unify with `Tensor a`, so both are scalar parameters. When tensors are passed to a binary scalar function, the system automatically inserts the two-argument lifting operation `tensorMap2`, a variant of `tensorMap`. For example:

```
min t1 t2 -- transformed to: tensorMap2 min t1 t2
```

`tensorMap2` applies the given binary function to component pairs determined by the symbolic indices. When the two tensors carry different index symbols, the result is the tensor product (all combinations). When they carry the same index symbol, only the diagonal components are computed directly—avoiding the $O(n^2)$ cost of computing all combinations before extracting the diagonal via index reduction. Our index reduction rules (Section 3) then process any repeated index symbols:

$$min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i, \begin{pmatrix}10\\20\\30\end{pmatrix}_j) = \begin{pmatrix}1&1&1\\2&2&2\\3&3&3\end{pmatrix}_{ij} \qquad min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i, \begin{pmatrix}10\\20\\30\end{pmatrix}_i) = \begin{pmatrix}1\\2\\3\end{pmatrix}_i$$

Tensor Functions. The tensor multiplication operator "`.`" demonstrates tensor parameter behavior:

```
def (.) {Num a} (t1 : Tensor a) (t2 : Tensor a) : Tensor a :=
  contractWith (+) (t1 * t2)
```

Since both parameters have type `Tensor a`, the system does *not* insert `tensorMap`—the tensors are processed structurally. Here, `*` is a scalar function (`{Num a} => a -> a -> a`, where `Num` is the type class for numeric types), so it is applied element-wise to the components of `t1` and `t2`, producing a tensor product whose indices are determined by our unified index rules (Section 3). The `contractWith (+)` function then contracts any supersubscript indices by summing diagonal components.

This single definition handles diverse tensor operations depending on the index patterns of the arguments:

$$\begin{pmatrix}1\\2\\3\end{pmatrix}^{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{i} = \mathrm{contractWith}\left(+,\ \begin{pmatrix}10&20&30\\20&40&60\\30&60&90\end{pmatrix}^{i}_{i}\right) = 10+40+90 = 140 \qquad \text{(inner product)}$$

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{j} = \mathrm{contractWith}\left(+,\ \begin{pmatrix}10&20&30\\20&40&60\\30&60&90\end{pmatrix}_{ij}\right) = \begin{pmatrix}10&20&30\\20&40&60\\30&60&90\end{pmatrix}_{ij} \qquad \text{(tensor product)}$$

Type-Directed tensorMap Insertion. The system inserts `tensorMap` when a tensor is passed to a parameter whose type does not unify with `Tensor a` (e.g., concrete types like `Integer`, or constraints such as `{Num a} => a`). Polymorphic functions such as `foldl` have unconstrained type variables that can be instantiated with `Tensor a`, so they are treated as tensor parameters and no `tensorMap` is inserted. In the compilation pipeline, this transformation runs after HM-based inference and before code generation, so insertion is decided from inferred types rather than syntax.

3 Systematic Index Reduction Rules

Our index reduction rules provide a uniform way to handle symbolic indices. When a tensor has multiple indices with the same symbol, we extract diagonal components:

```
[|[|11,12,13|],[|21,22,23|],[|31,32,33|]|]_i_i
-- Result: [|11,22,33|]_i (diagonal extraction)
```

When the same symbol appears as both superscript and subscript, we create a *supersubscript* preserving the diagonal without immediate contraction:

```
[|[|11,12,13|],[|21,22,23|],[|31,32,33|]|]~i_i
-- Result: [|11,22,33|]~_i (supersubscript)
contractWith (+) [|11,22,33|]~_i  -- Result: 66
```

This parameterized contraction enables diverse operations within a unified framework.

Formal Specification. The reduction algorithm $E(\{A, xs\})$ processes tensor A with index list xs:

$$\frac{e(xs) = \emptyset}{E(\{A, xs\}) \to \{A, xs\}}, \qquad \frac{\{k, j\} \in e(xs) \quad p(k, xs) = p(j, xs)}{E(\{A, xs\}) \to E(\{\mathrm{diag}(A, k, j), \mathrm{remove}(xs, j)\})}$$

$$\frac{\{k, j\} \in e(xs) \quad p(k, xs) \neq p(j, xs)}{E(\{A, xs\}) \to E(\{\mathrm{diag}(A, k, j), \mathrm{update}(\mathrm{remove}(xs, j), k, 0)\})}$$

Here, e finds duplicate-symbol index pairs, p returns index variance (super/sub), and 0 denotes a supersubscript. If a symbol occurs more than twice, the rules iterate until one representative remains; diagonalization enforces equality of the selected axes, so the final fully diagonal tensor is independent of the order of pair selection.

Conservativity and Diagnostic Trade-off. The relaxation is conservative on the index patterns accepted by standard notation: componentwise operations with matching indices behave as usual, distinct indices form tensor products, and opposite-variance repeated indices are represented as supersubscript diagonals that `contractWith (+)` turns into the standard Einstein sum. The extension is that patterns normally rejected by standard notation are not treated as ill-formed. For example, $u_i + v_j$ is interpreted by lifting `+` over both axes and yields a rank-2 tensor with indices ij. This uniform interpretation is what lets scalar functions and user-defined tensor operators share the same index machinery. The diagnostic trade-off is that an accidental mismatch such as writing v_j where v_i was intended produces a well-formed higher-rank value rather than an immediate index error; such mistakes must be detected from the resulting indices or by later index-sensitive code, not by a separate strict-index checker.

4 Syntactic Sugar

To keep programs close to mathematical notation, we provide compact syntax for symbolic variables, local symbols, and tensor symmetries. The following example combines local-symbol generation and symbolic-index declarations:

```
declare symbol r, θ : MathValue
def R~i_j_k_l : Tensor MathValue := withSymbols [m]
  ∂/∂ Γ~i_j_l x~k - ∂/∂ Γ~i_j_k x~l
  + Γ~m_j_l . Γ~i_m_k - Γ~m_j_k . Γ~i_m_l
```

Symmetry Declarations. We provide a unified syntax combining tensor definition with symmetry declarations inspired by commutator notation. For example, the Riemann curvature tensor satisfies $R_{abcd} = -R_{bacd}$, $R_{abcd} = -R_{abdc}$, and $R_{abcd} = R_{cdab}$, and these symmetries can be declared as:

```
def R[{_a_b}{_c_d}] : Tensor MathValue :=
  withSymbols [i] g_a_i . R~i_b_c_d
```

This simultaneously defines the tensor and enforces antisymmetry within each pair and symmetry between pairs, reducing pattern matching cases by more than half.

Pattern Matching for Tensor Indices. For operators requiring sophisticated index manipulation, we support pattern matching on symbolic tensor indices:

```
def ∇_c T~(a_1)...~(a_r)_(b_1)..._(b_k) :=
  ∂/∂ T~(a_1)...~(a_r)_(b_1)..._(b_k) x~c
  + sum (map (\i -> Γ~(a_i)_d_c .
                T~(a_1)...~(a_(i-1))~d~(a_(i+1))...~(a_r)_(b_1)..._(b_k))
             [1..r])
  - sum (map (\i -> Γ~d_(b_i)_c .
                T~(a_1)...~(a_r)_(b_1)..._(b_(i-1))_d_(b_(i+1))..._(b_k))
             [1..k])
```

5 Index Completion Rules for Differential Forms

We represent differential forms as tensors with omitted indices. An n-form is stored as a rank-n tensor; when the form participates in an operation, our *index completion rules* automatically supply the missing indices.

Uniform Completion (default): the same fresh index sequence is used for all arguments. For two 2-forms `A` and `B`:

```
A + B  -- becomes -->  A_t1_t2 + B_t1_t2
```

Disjoint Completion (with `!`): different fresh indices are added to different arguments. This is needed for the wedge product:

```
A !. B  -- becomes -->  A_t1_t2 . B_t3_t4
```

For fresh symbols t_i and forms A and B of degrees m and n, uniform and disjoint completion are:

$$\mathcal{C}_u(f(A,B)) = f(A_{t_1 \ldots t_m}, B_{t_1 \ldots t_n}), \quad \mathcal{C}_d(f(A,B)) = f(A_{t_1 \ldots t_m}, B_{t_{m+1} \ldots t_{m+n}})$$

where $\mathcal{C}_d$ is selected by `!`. Syntactically, `!` marks disjoint completion for that application (e.g., `A !. B`). For example, a third argument C of degree ℓ receives $t_1, \ldots, t_\ell$ under uniform completion, but $t_{m+n+1}, \ldots, t_{m+n+\ell}$ under disjoint completion. In the compilation pipeline, completion/reduction runs after HM-based typing and type-directed `tensorMap` insertion.

These rules enable concise operator definitions:

```
def (∧) {Num a} (A : Tensor a) (B : Tensor a) : Tensor a := A !. B
def d (A : Tensor MathValue) : Tensor MathValue :=
  !(flip ∂/∂) coord A
def hodge (A : Tensor MathValue) : Tensor MathValue :=
  let k := dfOrder A in withSymbols [i, j]
    (sqrt (abs (det g_#_#))) *
    (foldl (.) ((ϵ N k)_(i_1)..._(i_N) . A..._(j_1)..._(j_k))
               (map (\n -> g~(i_n)~(j_n)) [1..k]))
```

Here, `dfOrder` returns the form degree, and `ϵ N k` denotes the Levi-Civita tensor (`N`: manifold dimension, `k`: form degree).

6 Demonstration

We demonstrate our system with a complete program computing the Riemann curvature tensor of the 2-sphere (S^2) via curvature forms (Fig. 1). The program implements Christoffel symbols, connection forms, and curvature forms using Cartan's structure equations:

$$\Gamma_{ijk} = \frac{1}{2}\left(\frac{\partial g_{ij}}{\partial x^k} + \frac{\partial g_{ik}}{\partial x^j} - \frac{\partial g_{kj}}{\partial x^i}\right), \quad \Gamma^i_{\ jk} = g^{im}\Gamma_{mjk},$$
$$\Omega^i_{\ j} = d\omega^i_{\ j} + \omega^i_{\ k} \wedge \omega^k_{\ j}.$$

These formulae appear directly in the program, demonstrating that our notation allows mathematical expressions to be translated almost verbatim into executable code. The `antisymmetrize` operator used in Fig. 1 returns the antisymmetric part of its argument, ensuring that wedge products are alternating. Additional examples and interactive demonstrations are available in the Egison Mathematics Notebook [7]. The Egison language itself is described in [6].

7 Related Work

Our approach rests on three mechanisms, and each unlocks a capability that prior systems provide only partially or not at all. *(i) Unified index rules* apply the same rules to every operator, so a definition needs none of the per-operator index logic that the standard rules require—one short definition can cover a family of operations; for example, the single definition of . yields inner, outer, and matrix products. *(ii) Type-directed lifting* applies any ordinary scalar function, including user-defined ones, element-wise to tensors with neither annotation nor redefinition—the `tensorMap` transformation is inserted automatically—so existing code and higher-order functions compose with tensors for free. *(iii) Index completion* represents differential forms as tensors with omitted indices, so the wedge product, exterior derivative, and Hodge star become one-line definitions over the same machinery. No prior system provides all three; we discuss each category below, returning at the end to our own earlier work.

Index-Notation DSLs and Macros. Modern array languages provide index-based front-ends with syntax resembling mathematical notation. NumPy's `einsum` [19] takes a string of index labels; Julia offers macros such as `@tensor` in TensorOperations.jl [11] and Tullio.jl [1]; and frameworks like Tensor Comprehensions [23] compile einsum-like notation to high-performance kernels. These systems are powerful and performance-oriented, and we do not aim to compete on raw array throughput; our goal is notational fidelity, composability, and type-safe classification. The difference is structural (Table 2). In macro-based approaches, symbolic indices are scoped to the indexed block rather than carried by tensor values through ordinary function calls; reuse as a tensor operator therefore requires a host-language wrapper or macro [5]. Element-wise lifting must also be requested explicitly (e.g., broadcasting dots), and there is no counterpart to our index completion for writing differential-form operators in this style. The Wolfram language [24] (`Table`/`Sum`) and Diderot's EIN operator [14] are likewise tied to explicit index specification and

```
[1]: declare symbol r, θ, φ : MathValue

     def x := [| θ, φ |]

     def g_i_j := [| [| r^2, 0 |], [| 0, r^2 * (sin θ)^2 |] |]_i_j
     def g~i~j := [| [| 1 / r^2, 0 |], [| 0, 1 / (r^2 * (sin θ)^2) |] |]~i~j

     def Γ_j_l_k := (1 / 2) * (∂/∂ g_j_l x~k + ∂/∂ g_j_k x~l - ∂/∂ g_k_l x~j)
     def Γ~i_k_l := withSymbols [j] g~i~j . Γ_j_l_k

     def d t := !(flip ∂/∂) x t

     def ω~i_j := Γ~i_j_#
     def Ω~i_j := withSymbols [k] antisymmetrize (d ω~i_j + ω~i_k ∧ ω~k_j)

[2]: Ω~1_1
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

```
[3]: Ω~1_2
```

$$\begin{pmatrix} 0 & \frac{sin(\theta)^2}{2} \\ \frac{-sin(\theta)^2}{2} & 0 \end{pmatrix}$$

```
[4]: Ω~2_1
```

$$\begin{pmatrix} 0 & \frac{-1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}$$

```
[5]: Ω~2_2
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Fig. 1. An Egison session for S^2 curvature computation, including antisymmetrization (e.g., $dx \wedge dy \mapsto \frac{1}{2}dx \wedge dy - \frac{1}{2}dy \wedge dx$).

do not expose tensor operations as composable functions. The HM-based classification itself is host-agnostic and could be ported to such performance-oriented languages.

Library-based Approaches. Libraries such as Åhlander's C++ library [2], ITensor [10], and SageMath [22] provide tensor operators as built-in primitives, each carrying its own index-handling code. Because we instead unify the index rules, this per-operator machinery becomes unnecessary: a scalar operator such as `+` needs no tensor-specific definition at all—it is lifted automatically—and even tensor multiplication is an ordinary two-line user definition, `contractWith (+) (t1 * t2)` (Section 2), from which the inner, outer, and matrix products all follow by index pattern. Users can thus define and extend the tensor algebra themselves under static type checking, rather than depending on a fixed library of built-in operators.

Rank-Based Array Programming. Languages like APL and J use function rank [4] rather than named indices. Because indices are positional, expressions that permute axes (e.g., $A_{ij} + B_{ji}$) require explicit transpose or reordering operations, and the program text does not record which axis each index denotes.

Differential Forms. Existing approaches use specialized data structures: Karczmarczuk [13] uses tree-like p-vectors in Haskell, and Sussman and Wisdom [21] provide their own representations. We instead represent forms as tensors with omitted indices, reusing the tensor infrastructure to obtain the concise operators of capability (iii).

Relation to Our Earlier Work. Our earlier work [8] introduced scalar/tensor parameters for Einstein summation notation. This paper adds four extensions over that foundation: the formal index reduction/completion specifications, support for tensors with omitted indices for

Table 2. Comparison with macro-based index notation, illustrated by Julia's `@tensor`. The issue is not surface syntax for one expression, but whether symbolic indices remain confined to a macro block or are carried through ordinary function calls.

Capability	Macro-based approach (e.g. Julia `@tensor`)	Our approach (tensor operator as ordinary function)
inner product $u^i v_i$	`@tensor s = u[i]*v[i]`	`u~i . v_i`
transpose sum $A_{ij}+B_{ji}$	`@tensor C[i,j] := A[i,j]+B[j,i]`	`A_i_j + B_j_i`
scalar lifting	broadcasting requested explicitly, e.g. `min.(A,B)`	scalar `min` is lifted by type inference
reusable tensor operator	symbolic indices are local to the macro block; reuse requires a host wrapper or macro	`def (.) t1 t2 := ...`, usable as `foldl (.) ...`
omitted indices/forms	no index-completion counterpart	`def wedge A B := A !. B`

differential forms, symmetry-aware declarations and index-pattern operator definitions, and the type-inference-driven scalar/tensor classification that removes explicit scalar/tensor annotations from user programs.

8 Conclusion

We presented unified index rules that, combined with HM-based scalar/tensor function classification and index completion for differential forms, eliminate operator-specific index handling. The approach yields programs that closely resemble mathematical formulae with first-class, composable tensor operators and type safety without annotations, and has been applied in mathematical research [9,15].

While our implementation is in Egison, the core idea—classifying functions via HM-based type inference—is applicable to other statically-typed functional languages, though full integration of symbolic index handling would require compiler-level extensions. Future directions include integration with theorem provers such as Lean [17] and performance-oriented languages like Formura [18].

Acknowledgments. The author thanks Hiroki Fukagawa for encouraging the import of differential-form notation into programming languages; Yutaka Shikano for intensive discussions on the content of the paper; Momoko Hattori and Mayuko Kori for developing a user-friendly interface for the proposed system; and Hiromi Hirano, Hidehiko Masuhara, and Michal J. Gajda for helpful feedback on earlier versions of the paper.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Abbott, M., et al.: Tullio.jl: An einsum macro for Julia. `https://github.com/mcabbott/Tullio.jl` (2020), accessed: 2026-06-13
2. Åhlander, K.: Einstein summation for multidimensional arrays. Computers & Mathematics with Applications **44**(8-9), 1007–1017 (2002)
3. Backus, J.W.: The history of FORTRAN I, II and III. IEEE Ann. Hist. Comput. **1**(1), 21–37 (1979). `https://doi.org/10.1109/MAHC.1979.10013`
4. Bernecky, R.: An introduction to function rank. In: Proceedings of the International Conference on APL. pp. 39–43. ACM (1988). `https://doi.org/10.1145/55626.55632`

5. Bird, R.S., Wadler, P.: Introduction to Functional Programming. Prentice Hall (1988)
6. Egi, S.: The Egison Programming Language. http://www.egison.org/ (2011), accessed: 2026-02-26
7. Egi, S.: Egison Mathematics Notebook. http://www.egison.org/math/ (2017), accessed: 2026-02-26
8. Egi, S.: Scalar and Tensor Parameters for Importing Tensor Index Notation including Einstein Summation Notation. In: Proceedings of the 18th Scheme and Functional Programming Workshop, Co-located with ICFP (2017)
9. Egi, S., Maeda, Y., Rosenberg, S.: Diffeomorphism groups of circle bundles over symplectic manifolds. arXiv preprint arXiv:2011.01800 (2020)
10. Fishman, M., White, S.R., Stoudenmire, E.M.: The ITensor software library for tensor network calculations (2020)
11. Haegeman, J., et al.: TensorOperations.jl: Julia package for tensor contractions and related operations. https://github.com/Jutho/TensorOperations.jl (2020). https://doi.org/10.5281/zenodo.4292006, accessed: 2026-06-13
12. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society **146**, 29–60 (1969). https://doi.org/10.2307/1995158
13. Karczmarczuk, J.: Functional coding of differential forms. In: Scottish Workshop on Functional Programming (1999)
14. Kindlmann, G.L., Chiw, C., Seltzer, N., Samuels, L., Reppy, J.H.: Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. IEEE Trans. Vis. Comput. Graph. **22**(1), 867–876 (2016). https://doi.org/10.1109/TVCG.2015.2467449
15. Maeda, Y., Rosenberg, S., Torres-Ardila, F.: The geometry of loop spaces II: Characteristic classes. Advances in Mathematics **287**, 485–518 (2016)
16. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://doi.org/10.1016/0022-0000(78)90014-4
17. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: CADE-25. LNCS, vol. 9195, pp. 378–388. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_26
18. Muranushi, T., et al.: Automatic generation of efficient codes from mathematical descriptions of stencil computation. In: FHPC@ICFP 2016. pp. 17–22. ACM (2016). https://doi.org/10.1145/2975991.2975994
19. NumPy Developers: NumPy. https://www.numpy.org (2025), accessed: 2026-02-26
20. Ricci, M., Levi-Civita, T.: Méthodes de calcul différentiel absolu et leurs applications. Mathematische Annalen **54**(1-2), 125–201 (1900)
21. Sussman, G.J., Wisdom, J.: Functional Differential Geometry. MIT Press (2013)
22. The Sage Developers: SageMath - Open-Source Mathematical Software System. https://www.sagemath.org/ (2025), accessed: 2026-02-26
23. Vasilache, N., et al.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. CoRR **abs/1802.04730** (2018)
24. Wolfram Research: Wolfram Language Documentation. https://reference.wolfram.com/language/ (2025), accessed: 2026-02-26