THE UNIVERSITY OF CHICAGO

DEVELOPING DIDEROT

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

CHARISEE CHIW

CHICAGO, ILLINOIS
JUNE 2017
To my family
TABLE OF CONTENTS

LIST OF FIGURES ............................................................. vii

ACKNOWLEDGMENTS ............................................................ viii

ABSTRACT ................................................................. ix

1 INTRODUCTION .......................................................... 1
  1.1 Diderot ................................................................. 3
    1.1.1 The Diderot Language .......................................... 3
    1.1.2 The Diderot Compiler .......................................... 5
  1.2 Implementing Tensor Fields ......................................... 7
    1.2.1 Concepts ...................................................... 7
    1.2.2 Field Normalization .......................................... 10
  1.3 Contributions ........................................................ 11
    1.3.1 Language expressivity ........................................ 11
    1.3.2 Correctness and Testing ...................................... 13
    1.3.3 Extension of language ........................................ 14
  1.4 Dissertation Overview ............................................. 14

2 DESIGN ................................................................. 15
  2.1 EIN notation ......................................................... 16
  2.2 Generating EIN Operators .......................................... 21
    2.2.1 Implementation ............................................... 21
    2.2.2 Advantages ................................................... 22
  2.3 Field Reconstruction ................................................ 25
  2.4 Discussion ........................................................... 27
    2.4.1 The Case for a New IR ........................................ 27
    2.4.2 EIN as building blocks ....................................... 30
    2.4.3 Programmability ............................................... 31

3 IMPLEMENTATION TECHNIQUES ......................................... 33
  3.1 Normalization ....................................................... 34
    3.1.1 Substitution .................................................. 35
    3.1.2 Rewriting ..................................................... 38
  3.2 Optimization and Transformations .................................. 42
    3.2.1 Shift ......................................................... 44
    3.2.2 Tshape ........................................................ 46
    3.2.3 Split ........................................................ 50
    3.2.4 Slice ........................................................ 53
    3.2.5 Examples ..................................................... 56
  3.3 Benchmarks .......................................................... 58
    3.3.1 Experimental Framework ...................................... 58
    3.3.2 The Effect of implementing EIN ......................... 59
3.3.3 The Effect of Compiler Settings ........................................ 61
3.3.4 First-Order versus Higher-Order .......................................... 66

4 PROPERTIES OF NORMALIZATION ........................................... 67
4.1 Type Preservation ................................................................. 67
  4.1.1 Typing EIN Operators .................................................... 67
  4.1.2 Type preservation Theorem ............................................. 71
4.2 Value Preservation ............................................................... 77
  4.2.1 Value Definition .......................................................... 77
  4.2.2 Value Preservation Theorem ............................................. 83
4.3 Termination ...................................................................... 87
  4.3.1 Normal Form .............................................................. 87
  4.3.2 Termination ............................................................... 89
4.4 Discussion ..................................................................... 94

5 AUTOMATIC TESTING MODEL ............................................... 95
5.1 Diderot’s Automatic Testing model ........................................ 96
  5.1.1 Overview ................................................................. 97
  5.1.2 Generating test cases ................................................... 97
  5.1.3 Testing Frame .......................................................... 99
  5.1.4 Generate Diderot program ............................................. 100
  5.1.5 Data Creation .......................................................... 101
  5.1.6 Evaluation ............................................................... 102
  5.1.7 Numerical Instability ................................................... 104
  5.1.8 Application details ...................................................... 105
5.2 Visualization Verification .................................................... 106
  5.2.1 Concept ................................................................. 107
  5.2.2 Pipeline ................................................................. 107
5.3 Results and Performance for DATm ........................................ 109
  5.3.1 Experimental Framework ............................................. 109
  5.3.2 Exhaustive vs. Random Testing ....................................... 110
  5.3.3 Progress ................................................................. 110
  5.3.4 Bugs ................................................................. 112
  5.3.5 Visualization Results ................................................... 116
5.4 Discussion ................................................................. 116

6 EXTENSION ................................................................. 119
6.1 Background ................................................................. 120
6.2 Creating and Visualizing FEM data ....................................... 122
6.3 Our Approach ............................................................... 124
6.4 Demonstration ............................................................... 126
  6.4.1 Communication and evaluating visualization ......................... 126
  6.4.2 PDE Example .......................................................... 129
6.5 Discussion ............................................................... 131
# Table of Contents

7 APPLICATIONS .................................................................................. 132
  7.1 Streamline the process of adding to the Diderot language ............... 132
  7.2 Visualization Applications .......................................................... 134

8 RELATED WORK ............................................................................... 138

9 CONCLUSION ................................................................................... 144
  9.1 Future Work ................................................................................ 144
    9.1.1 Correctness and Testing ...................................................... 144
    9.1.2 Design and Implementation .............................................. 145
    9.1.3 FEM and Diderot ............................................................... 146
    9.1.4 Index notation on the surface language ................................. 147

REFERENCES ....................................................................................... 150

A PROOFS ............................................................................................ 156
  A.1 Type Preservation Proof ............................................................ 156
  A.2 Value Preservation Proof .......................................................... 170
  A.3 Termination ............................................................................... 182
    A.3.1 Size reduction ................................................................... 182
    A.3.2 Termination implies Normal Form ....................................... 190
    A.3.3 Normal Form implies Termination ....................................... 200
# LIST OF FIGURES

1.1 Diderot program to compute surface curvature [42] from [16] .......................... 4

2.1 EIN IR syntax .................................................................................. 20

3.1 Measuring the effect of implementing EIN ............................................. 60

3.2 Measurement of applying optimizations techniques .............................. 62

3.3 Size of programs at different phases of the compiler ............................ 63

3.4 Measurement of applying optimizations techniques .............................. 64

3.5 Programs written with first or higher order operators ............................ 65

4.1 Typing Judgments .............................................................................. 72

4.2 Inversion Lemma for Typing Judgements ............................................ 73

4.3 Value definitions for EIN .................................................................... 77

4.4 Value Judgements ............................................................................... 82

5.1 Subset of language tested with DATm .................................................. 98

5.2 Automated Testing models ................................................................... 103

5.3 Example of metamorphic testing result ............................................... 106

5.4 Measurements from applying DATm using an exhaustive and random search for test cases ................................................................. 111

5.5 Measurements from applying DATm over different snapshots of the compiler. . 112

5.6 List of bugs uncovered by DATm ......................................................... 113

5.7 Results from applying DAVm ............................................................. 116

6.1 Visualizing FEM data with different visualization tools ......................... 123

6.2 Establish communication between Diderot and Firedrake .................... 128

6.3 Applying representation invariance principle ...................................... 129

6.4 Visualize PDE with Diderot ............................................................... 131
ACKNOWLEDGMENTS

My co-advisor, John Reppy, has been a guide to me throughout the years. He has been both deeply involved with various aspects of our work together and trusting in my own pursuits. The trust and freedom I got as a grad student has helped me become a better researcher. Thank you for your patience and respect.

I would like to thank my co-advisor Gordon Kindlmann for being a mentor. He ignited my interests in developing Diderot, work that shaped my dissertation, by sharing his expertise. More importantly, thank you for being a true mentor and empowering me to keep going.

Thank you to my committee member Ridgway Scott and his student Hannah Morgan for teaching me about FEM. Thank you to Andrew McRae and the rest of the Firedrake team for their collaborative efforts. Thank you to Anne Rogers for being a role model and keeping her office door open. Thank you to my fellow office mates, Lamont Samuels, Kavon Farvardin, Joe Wingerter, and Brian Hempel for the hours of chatting about life.

I would like to thank the wonderful and supportive people in my life. My parents, Elena and Wayne, and my good friend Bijal Makadia who drove halfway across the country to move me into my apartment in Chicago. I would like to thank Robert Singer, Anna Olson, and Grace Thornton for helping me make this place my home. I want to thank my relatives Tia Isabel, Charmaine, and Susan for their support and kindness.

Portions of this research were supported by National Science Foundation under award CCF-1564298.
This dissertation describes the development of Diderot. Diderot is a domain-specific language for scientific visualization and image analysis. Algorithms in this domain are used to visually explore data and compute features and properties. Diderot is designed to enable the transition of ideas into code by providing a high-level language that mimics a mathematically friendly notation so that users can express mathematical concept into working code. The execution obstacle for the Diderot compiler is how to bridge the wide semantic gap between higher-order operations on tensor fields and their implementation as efficient executable code. It is important and challenging to work on develop the Diderot language so that the user can write intuitive code, compile programs with complicated tensor math, and believe in the correctness of the compiler. This dissertation is focused on the design and implementation of these ideas in the compiler, addressing the technical challenges that arise, formalizing the properties in our rewrite system, confidently testing our implementation, and extending the language to another domain.
CHAPTER 1
INTRODUCTION

This dissertation describes the development of Diderot. Diderot is a domain-specific language for scientific visualization and image analysis. Algorithms in this domain are used to visually explore data and compute features and properties. Diderot is designed to enable the transition of ideas into code by providing a high-level language that mimics a mathematically friendly notation so that users can express mathematical concept into working code. The execution obstacle for the Diderot compiler is how to bridge the wide semantic gap between higher-order operations on tensor fields and their implementation as efficient executable code. It is important and challenging to work on develop the Diderot language so that the user can write intuitive code, compile programs with complicated tensor math, and believe in the correctness of the compiler. This dissertation is focused on the design and implementation of these ideas in the compiler, addressing the technical challenges that arise, formalizing the properties in our rewrite system, confidently testing our implementation, and extending the language to another domain.

Diderot is a parallel domain-specific language for the analysis and visualization of multidimensional scientific images, such as those produced by CT and MRI scanners. Many visualization methods seek to measure properties from continuous tensor fields reconstructed from the discrete image data and not just the data itself; these algorithms require a high level of tensor mathematics. A novel aspect of Diderot’s design is that it supports a form of higher-order programming where tensor fields (i.e., functions from 3D points to tensor values) are first-class values. Diderot is designed to support algorithms that are based on differential tensor calculus and supports a higher-order mathematical model that can be directly manipulated.

The implementation challenge is taking high-level code and transforming it to efficient executables. To address the implementation challenge, we created a new intermediate representation EIN to concisely represent new and existing operations in the Diderot language. We
introduce the intermediate representation and describe its execution. Adding the EIN representation to the Diderot compiler has greatly increased the expressiveness of the language, which, in turn, enables a richer set of algorithms to be directly programmed in Diderot. Unfortunately, programs written with the richer language brought along new challenges.

Transformations inside the compiler resulted in a combinatorial explosion in the size of the IR. As a result, Diderot programs either took too long to compile or did not compile at all. This problem significantly limited the use of the new features. To address the compilation problem, we have developed a number of techniques that keep the size of the IR in check while not restricting the expressivity of our language. We describe our compilation techniques and provide examples. We measure the impact of the techniques used together and applied at different levels of abstraction.

A key question for a high level language, such as Diderot, is how do we know that the implementation is correct? To ensure confidence we address evaluating the correctness of our implementation with two complimenting parts; proofs and automated testing. The normalization process is a key part of the compiler but it is complicated and requires examination. We describe the formal properties of the normalization process and prove that it is type preserving, value preserving, and terminating.

While the proofs serve to illustrate the properties of the normalization system, they do not test the full implementation of the compiler from source to executables. Unfortunately, manual testing can be time-consuming, prone to biases, and insufficient to testing the large combination of possible test programs. We introduce our automated testing model for implementing property-based testing. It successfully generates and evaluates thousands of Diderot programs based on a ground-truth.

The core of a visualization program written in Diderot is independent of the source of the data. Yet Diderot could only define a field as discrete image data convolved with a convolution kernel. If Diderot could define other types of fields then we could apply visualization programs to other types of data. Diderot could potentially be used to debug
and visualize fields created by another domain. We take a first step towards visualizing fields defined by finite element data.

## 1.1 Diderot

The Diderot language is the platform for the research presented in this dissertation. Previous work provided a description of the Diderot language [13, 14, 16, 41]. In this section we provide an overview of the computational core of the Diderot language and its compiler.

### 1.1.1 The Diderot Language

We use the Diderot language to extract features from a dataset. In Section 7.2 we describe some visualization features in more detail. We provide an example of a Diderot program in Figure 1.1 and describe samples of the Diderot code in the following text.

The computational core of Diderot is organized around two families of types: tensors and tensor fields. Tensors include scalars (0th-order), vectors (1st-order), and matrices (2nd-order), and are the concrete values that the system computes with. A value with type “\texttt{tensor}[d_1, \ldots, d_n]” is an \(n\)th-order tensor in \(\mathbb{R}^{d_1 \times \cdots \times d_n}\); we refer to \(d_1, \ldots, d_n\) as the \textit{shape} of the tensor.\(^1\) Diderot supports the standard linear algebra operations on tensors, such as addition and subtraction, inner, outer, and colon products, trace, Eigenvectors and values, \textit{etc.} Diderot’s expression syntax is designed to mimic mathematical notation, while still retaining the flavor of a programming notation. For example, one writes “\((u \otimes v) / |u \otimes v|\)” for the normalized outer product of two tensors.

In textbooks and research papers about visualization and analysis, methods are often mathematically defined in terms of fields, while implementation details are presented separately in terms of the data representation [39]. In visualization algorithms, tensor fields serve as a mathematical abstraction of the data sets produced by various digital imaging

---

\(^1\) The exclusive internal use of the orthonormal elementary basis for representing tensors means that covariant and contra-variant indices can be treated equally.
technologies (e.g., Diffusion MRI). These imaging technologies sample physical objects at discrete points producing a multidimensional grid of sample values called voxels. A novel feature of Diderot is that it supports programming directly with fields, instead of with the discrete voxels.

Images are multi-dimensional arrays of tensor values. In our syntax “image(d)[σ], the image has d-dimensions (d axes), and each value is a tensor shape σ. A 3D grayscale image is “image(3)[]. We use convolution (⊗) with kernels to reconstruct a continuous representation from the samples, and we model the reconstruction in the language as a continuous tensor field. A value with type “field#k(d)[d1 ... dn]” is a $C^k$ continuous function (i.e., we can apply up to k levels of differentiation) in $\mathbb{R}^d \to \mathbb{R}^{d1 \times \cdots \times d_n}$. Note, empty brackets ‘[ ]’ indicates a scalar or scalar field.

```diderot
image(3)[] img = image("quad-patches.nrrd");
field #2(3)[] F = bspln3 ⊗ img;
field #0(2)[3] RGB = tent ⊗ image("2d-bow.nrrd");
...
strand RayCast (int ui, int vi) {
  ...
  update {
    ...
    vec3 grad = -∇F(pos);
    vec3 norm = normalize(grad);
    tensor [3,3] H = ∇⊗∇ F(pos);
    tensor [3,3] G = -(P•H•P) / |grad|;
    real disc = sqrt(2.0*|G|^2 - trace(G)^2);
    real k1 = (trace(G) + disc)/2.0;
    real k2 = (trace(G) - disc)/2.0;
    vec3 matRGB =
      RGB([max(-1.0, min(1.0, 6.0*k1)),
           max(-1.0, min(1.0, 6.0*k2))]);
  }
  ...
}
```

Figure 1.1: Diderot program to compute surface curvature [42] from [16].

As mentioned above, tensor fields can be defined by convolving a reconstruction kernel
with an image. For example, the following Diderot declaration (from Figure 1.1) defines a 3D scalar field F:

\[
\text{field } \#2(3)[]F = \text{bspln3 } \otimes \text{ img;}
\]

The field F is reconstructed using the bspln3 kernel from the file img1.nrrd. The continuity of F is \(C^2\), which is determined by the choice of the bspln3 kernel. Mathematically, fields are functions and we can apply them to points in their domain, which we call probing the field. For example, if p is a point in \(\mathbb{R}^3\) (i.e., it has type \text{tensor}[3]), then \(F(p)\) will evaluate to a scalar (since F is a scalar field).

The real power of programming with fields comes from Diderot’s support for higher-order operators, which allows fields to be defined in terms of combinations of other fields. Just as in mathematics, it is normal to write “\(A + B\)” to denote \(\lambda p. (A(p) + B(p))\), Diderot lifts most tensor operations to work on fields, so if \(A\) and \(B\) are fields of the same type, then \(A+B\) denotes the field that is their lifted sum. In addition to lifted operators, Diderot also supports the standard differentiation operators on fields (\(\nabla\), \(\nabla\cdot\), \(\nabla\otimes\), and \(\nabla\times\)). By differentiating fields, we are able to extract geometric features from the images. For instance, we can use the Hessian of the field \(F\) to compute curvature of surfaces (from Figure 1.1).

\[
\text{tensor } [3,3] \ H = \nabla \otimes \nabla \ F(\text{pos});
\]

While the original implementation of Diderot supported tensors and fields, it had minimal support for lifted operators and only supported \(\nabla\), \(\nabla\otimes\), and a restricted form of \(\nabla\times\).

**1.1.2 The Diderot Compiler**

The Diderot compiler is organized into three main phases: the front-end, optimization and lowering, and code generation. This dissertation is primarily concerned with the optimization

---

2. We use the Teem library’s Nrrd file format to represent multidimensional data sets (both input and output) [68].
and lowering phase, but we include a brief description of the other phases to provide context. The front-end consists of parsing, type checking, and simplification. Although Diderot is a monomorphic language, most of its operators have instances at multiple types. For example, addition works on integers, tensors of all shapes, fields, and combinations of fields and tensors. The type-checker uses a mix of ad hoc overloading and polymorphism to handle these operators. The output of type-checking is a typed AST where operators are instantiated at specific monotypes. The typed AST is then converted into a simplified representation, where user-defined functions are inlined and named temporaries are introduced for intermediate values.

Optimization and lowering involves a series of three IRs that are based on a common Static Single Assignment (SSA) form [22] control-flow graph (CFG) representation. High-IR is essentially an SSA version of the source language that supports the surface language types and operations. Specifically, fields and operations on fields are represented at the High-IR level. Mid-IR supports linear-algebra operations on tensors and reconstruction-kernel evaluation. At Mid-IR stage, higher-order types (i.e., fields) and operations (e.g., probes and differentiation) have been translated into concrete tensor operations. Low-IR supports basic operations on hardware-vectors (e.g., Intel’s SSE registers), scalars, and memory objects. The optimization and lowering phase uses several different kinds of transformations in the process of converting High-IR to Low-IR. These include traditional optimizations at each level, such as dead-variable elimination and value numbering; domain-specific optimizations that are specific the particular IRs; lowering transformations that expand higher-level operations into equivalent sequences of lower-level operations; and normalization of the High-IR representation to enable lowering of field operations. This last transformation is of particular importance and we discuss it in detail in Chapter 3.

Code generation involves mapping the Low-IR CFG to a block-structured IR with expression trees. We then generate either vectorized C++ code or OpenCL code from the IR, which is compiled to produce either a library or an executable.
1.2 Implementing Tensor Fields

Operations on fields can be classified as either *declarative*, which are operations that define field values, or *computational*, which are operations that query a field to extract a concrete value. Translating computational field operations into executable code is one of the central challenges of the Diderot compiler.

1.2.1 Concepts

In this section, we given an informal description of the basic techniques used to implement the translation. First we provide some additional mathematical concepts, then we introduce the basic mechanisms for implementing continuous tensor fields.

Some background Permutation tensor or Levi-Civita tensor is represented in EIN as $\mathcal{E}_{ijk}$.

\[ \mathcal{E}_\alpha \begin{cases} +1 & \alpha \text{ is cyclic (0,1,2)} \\ -1 & \alpha \text{ is anti-cyclic (2,1,0)} \\ 0 & \text{otherwise} \end{cases} \]  

(1.1)

Kronecker delta function is represented in EIN as $\delta_{ij}$.

\[ \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases} \]  

(1.2)

Other properties of kronecker delta value

\[ \delta_{xz}\delta_{zy} = \delta_{xy} \]  

(1.3)

---

3. We describe probing a field to extract a value at a point. The other computational operation is testing if a point lies in the domain of a field, which produces a boolean result.
and
\[ \delta_{ii} = 3 \] (1.4)

Let us define an orthonormal basis \( \beta \) with unit basis vectors as \( b_i, b_j, \ldots \). The following properties can be found in [37]. Each basis vector is linearly independent and normalized such that
\[
\begin{align*}
b_i \cdot b_j &= \begin{cases} 
1 & i=j \\
0 & \text{otherwise}
\end{cases} \\
\delta_{ij} &= b_i \cdot b_j 
\end{align*}
\] (1.5) (1.6)

Any vector \( u \) can be defined by a linear combination of these basis vectors and have the following properties.
\[
u = \Sigma_i u_i b_i
\]

or a component of a tensor
\[
u_j = u \cdot b_j
\] (1.7)

**Tensor Fields** In the base case, a scalar field \( F \) is defined as the convolution \( V \circledast H \) of an image \( V \) with a reconstruction kernel \( H \), where \( H \) is a separable kernel function that can be expressed over multiple arguments (e.g., \( H(x, y) = h(x)h(y) \) in 2D). Probing the field \( F \) at a point \( p \) involves mapping \( p \) to a region of \( V \) and then computing a weighted sum of the voxel values in the region (the weights are computed using the kernel) [16]. Let us assume that \( F \) is a 2D field; then \( F(p) \) can be computed as
\[
(V \circledast H)(p) = \sum_{i=1}^{s} \sum_{j=1}^{s} (V[n_0 + i, n_1 + j]h(f_0 - i)h(f_1 - j))
\]

where the *support* of the kernel \( H \) is \( 2s \), \( M \) is a matrix for array orientation, \( x \) is \( p \) mapped to \( V \)'s coordinate system (image space) using \( M \), \( n = [x] \), \( n = (n_0, n_1) \), \( f = x - n \), and
\( f = (f_0, f_1). \)

**Differentiating Tensor fields**  We introduced a notation to note differentiation \( \nabla^{(i)} \) [16]. The superscript indicates the level of differentiation, where the term \( \nabla^{(2)} \) indicates the second derivative (Hessian) and not the Laplacian (or the Trace of the Hessian).

We probe a field expression involving a higher-order operation such as in the expression \( \nabla F(p) \). We can normalize the expression using direct-style operators as follows:

\[
\nabla F(p) \implies \nabla((V \otimes H)(p)) \implies (V \otimes (\nabla H))(p)
\]

We record multiple levels of kernel differentiation by adding a superscript to \( \nabla \):

\[
\nabla(V \otimes \nabla^{(i)} H) \implies V \otimes \nabla^{(i+1)} H
\] (1.8)

Because kernels are separable, their differentiation is straightforward:

\[
\nabla H(x, y) = \begin{bmatrix}
\frac{\partial}{\partial x} \\
\frac{\partial}{\partial y}
\end{bmatrix} H(x, y) = \begin{bmatrix}
\frac{\partial}{\partial x} H(x, y) \\
\frac{\partial}{\partial y} H(x, y)
\end{bmatrix} = \begin{bmatrix}
\frac{\partial}{\partial x} (h(x)h(y)) \\
\frac{\partial}{\partial y} (h(x)h(y))
\end{bmatrix} = \begin{bmatrix}
(h'(x)h(y)) \\
(h(x)h'(y))
\end{bmatrix}
\]

The result is in world-space and needs to be transformed back to index space. Consider
vector $g$ made up of directional derivatives $d$:

$$
\mathbf{g} = \begin{bmatrix}
\nabla \varphi \cdot d_i \\
\nabla \varphi \cdot d_j \\
\nabla \varphi \cdot d_k 
\end{bmatrix}
$$

Consider a component of $g$

$$
g = \nabla \varphi \cdot d_i \\
= \nabla_j \varphi \cdot (b_j (b_j \cdot d_i)) \quad \text{use orthonormality of } \mathcal{B} \text{ to rewrite.}
\]

$$
= \sum_j (\nabla_j \varphi \cdot b_j) (d_i \cdot b_j) \quad \text{by associativity of multiplication}
$$

$$
= [d_i \cdot b_1 \; d_i \cdot b_2 \; d_i \cdot b_3] \begin{bmatrix}
\nabla \varphi \cdot b_1 \\
\nabla \varphi \cdot b_2 \\
\nabla \varphi \cdot b_3
\end{bmatrix} \quad \text{by unfolding summation index } j
$$

The vector form

$$
\mathbf{g} = \begin{bmatrix}
d_1 \cdot b_1 \\
d_2 \cdot b_1 \\
d_3 \cdot b_1
\end{bmatrix}
\begin{bmatrix}
\nabla \varphi \cdot b_1 \\
\nabla \varphi \cdot b_2 \\
\nabla \varphi \cdot b_3
\end{bmatrix}
$$

$$
= M^T [\nabla \varphi]_\mathcal{B} \quad \text{by variable substitution for matrix}
$$

$$
(M^{-T})g = [\nabla \varphi]_\mathcal{B} \quad \text{by dividing both sides by } M
$$

We multiply the result by the inverse transpose of the transformation matrix to covert the result to index space.

### 1.2.2 Field Normalization

Part of the challenge of translating a language that does computations on tensor fields is the requirement of field normalization. The Diderot users can define a tensor field with image data and convolution kernel or as the result of a computation. Diderot, allows fields to be defined by complex expressions involving lifted tensor operations and differentiation operators. In order to compile probes of arbitrary fields, we must perform a normalization
of the field expressions before lowering. The basic strategy of normalization is to push differentiation down to the leaves where it can be represented using the derivatives of the kernel functions and to push probes down to the convolutions. For example, an expression \((F + G)(p)\) can be rewritten as \(F(p) + G(p)\), which pushes the probe down the expression tree. This transformation has lowered a higher order expression \((F + G)\) to a first-order sum of tensors.

A normalized or recognizable field term is a field probed at a position \(F(x)\). If a differentiated operator is used in the surface language, then the differentiation term is distributed over a computation and pushed down to the kernel in a convolution operator \(V \odot \nabla H(x)\). The normalization is enforced so that the compiler can recognize these probed field terms and before field reconstruction Section 2.3.

1.3 Contributions

Our work is important to the development of the Diderot language. In this section we describe our contributions organized by area. We improved the programmability and the expressivity of the language by supporting a high-level of math directly in the code. We have worked on illustrating correctness and developing a rigorous automated model to test the new features in the language. We also provide the first step in extending Diderot to another domain.

1.3.1 Language expressivity

Our work in the compiler enables Diderot’s higher-order programming model. We created a new intermediate representation for the Diderot compiler, called EIN. We describe the details for the EIN IR design, implementation, and how we addressed some of the technical challenges in its implementation.
Design and implementation of EIN

We created the EIN IR to represent the mathematical core to Diderot programs. EIN expressions are used to concisely represent field reconstruction and operations on and between tensor and tensor fields. Besides adding generality to existing operators, we are able to extend the Diderot model to provide lifted versions of tensor operators at the field level. As a result, Diderot is a richer and more complete language.

Our normalization process handles generic EIN operators. We created systematic substitution process and a robust rewriting system, designed around the IR, to do necessary domain-specific rewriting and index-inspired optimizations. Our implementation of EIN supports the translation of a high-level of math written directly in Diderot into executable code. As a result, a user can rely on the compiler to do the necessary derivations, such as differentiation and field normalization. In Chapter 2 we describe the design of the EIN notation and the generation of EIN operators.

Compilation Techniques

As previously discussed, compilation issues significantly restricted our use of new language features. To handle the compilation issues we developed a number of implementation techniques. Our methods reduce the size of the IR in different passes of the compiler while maintaining the mathematical meaning behind the computation. We also measure the impact of the techniques used together and applied at different levels of abstraction. We demonstrate that EIN can not only compile more programs than previously possible but also, it compiles faster and offers faster executables. Chapter 3 describes how we approach the compilation issue and a measurement of our approach.
1.3.2 Correctness and Testing

Testing a compiler for a high-level mathematical programming language poses a number of challenges not found in previous work on testing compilers. We describe the formal properties of the normalization process, a core part of the Diderot compiler. We also provide an automated model for testing the implementation of the high level language based on a ground truth.

Hand-written Proofs

To increase our confidence in the compiler, we formally describe the properties of our rewriting system. We aim to prove that the rewrite system is type-preserving, value preserving, and terminating. Chapter 4 provides the details for these proofs. A simple case analysis on the rewrite rules combined with algebraic reasoning can be used to show that each rule satisfies correctness. We will define a well-founded size metric on EIN expressions and normal form to be a subset of EIN expressions. We show that the rewrite rules always decrease the size of an expression. For any syntactic construct, we intend to prove that for any EIN expression we can apply rewrites until termination, at which point we will have reached a normal form expression.

Automatic Testing model

We present DATm, Diderot’s automated testing model to check the correctness of the core operations in the programming language [15]. DATm can automatically create test programs and predict what the outcome should be. We measure the accuracy of the computations written in the Diderot language based on how accurately the output of the program represents the mathematical equivalent of the computations. Chapter 5 introduces the pipeline for DATm, a tool that can automatically create and test tens of thousands of Diderot test programs and has found numerous bugs. We make a case for the necessity of extensive
testing by describing bugs that are deep in the compiler and only could be found with a unique application of operations. Lastly, we demonstrate that a metamorphic testing approach can be enabled by using parts of the testing model.

1.3.3 Extension of language

We demonstrate our first and modest approach of visualizing FEM data with Diderot. We describe our approach in detail and provide examples. Using Diderot, we do a simple sampling and a volume rendering of a FEM field. These examples showcase Diderot’s ability to debug and provide a visualization result. In addition, it provides motivation for future work. Chapter 6 describes the extension of the Diderot language to represent other types of fields.

1.4 Dissertation Overview

The rest of the dissertation is organized as follows:

- Chapter 2 describes the design and motivation behind the EIN IR.
- Chapter 3 offers the implementation and compilation techniques.
- Chapter 4 describes and defends properties of the normalization phase.
- Chapter 5 introduces property-based automated testing of the compiler.
- Chapter 6 illustrates the extension of Diderot to other types of data.
- Chapter 7 provides applications of the work presented in this dissertation.
- Chapter 8 surveys related work.
- Chapter 9 concludes and describes future work
CHAPTER 2
DESIGN

We rely on the expressivity of the Diderot language to support the implementation of visualization ideas. Visualization algorithms involve computing certain properties from a dataset. The mathematical core of these ideas are ingrained in tensor calculus. Central to them are operations on and between tensors fields. For a scientist or mathematician, it might be natural to think of these concepts first in a mathematical notation rather than in lower level code. For instance, we might write operations between two fields as \( c = a + b \) or \( d = \nabla(ac) \) while intending to compute \( c[x] = a[x] + b[x] \) or \( d[x] = \nabla(ac)[x] \), respectively. The tensor operations are lifted to operate between tensor fields. The actual implementation of these computations into lower level code can be difficult and tedious.

Diderot eases the transformation of ideas into workable intuitive code by allowing the math-like notation to be written directly in the language. The work in this chapter generalized the Diderot model to provide lifted versions of the standard linear algebra operations (e.g., tensor addition, dot products, norms, determinants, etc.) on tensor fields.

By enabling lifted operations the Diderot code can enable the implementation of more complicated computations. Crest Lines are places were the surface curvature is maximal along the curvature direction [48]. The user should be able to turn their mathematical reasoning directly into code by building on the surface curvature code (Figure 1.1).

\[
\text{real } k1 = (\text{trace}(G) + \text{disc})/2.0;
\]
\[
\text{real } \text{out} = \nabla \otimes \nabla \ldots k1;
\]

To compute crest lines, the value \( k1 \) is differentiated twice (among applying other tensor operations). This is not possible since \( k1 \) is a tensor type and not a field type. The program needs to be written using first-order terms so that the derivative operator is only applied to field types. The Diderot user would need to apply the differentiation operator by hand (by applying the chain rule, quotient rule manually etc.) a process that is tedious, time-consuming, and error-prone. By enabling lifted operations in the surface code the field and
derivatives can be referred to easily.

\[
\text{field } \#2(3)[] \quad k1 = (\text{trace}(G) + \text{disc})/2.0;
\]
\[
\text{field } \#0(3)[] \quad \text{out} = \nabla \otimes \nabla \ldots k1;
\]

We are able to apply tensor operations between and on tensor fields.

This level of expressiveness makes writing Diderot code easier, faster, and more intuitive. Diderot users are able to analyze and manipulate tensor fields and then rely on the compiler to handle the necessary derivations. The richer language allows users to focus on their ideas, rather than the difficult and tedious implementation.

In order to support the higher level language, we designed a new intermediate representation for the Diderot compiler, called EIN. EIN provides a concise and expressive internal representation for tensor and tensor-field operations.\(^1\) We also adapt the IR to include field reconstruction. The EIN representation makes it possible to support a richer set of higher-order operations, that were not feasible previously. Additionally, the implementation makes it easy to define new operations and extend the language.

This chapter describes the design of the EIN IR. Section 2.1 introduces EIN notation. Section 2.2 describes the generation and creation of EIN operators. Section 2.3 illustrates the implementation of field reconstruction in EIN. We end with a discussion in Section 2.4.

## 2.1 EIN notation

We have developed a new intermediate representation, that we call EIN, which is much more compact than the full expansion of tensor expressions, while permitting index-specific operations. This new representation is embedded in the same SSA-based representation as the direct-style operators, except that we now have EIN assignment nodes of the form

\[
t = \lambda \text{params} \langle e \rangle_\sigma(\text{args})
\]

\(^1\) Our representation was inspired by \textit{Einstein Index Notation}, which is a concise written notation for tensor calculus invented by Albert Einstein [29].
where

- \( t \) is a tensor or field variable being assigned
- \( \lambda \text{params}\langle e \rangle_{\sigma} \) is an operator defined in the EIN IR, with formal parameters \( \text{params} \), EIN expression \( e \), and index mapping \( \sigma \).
- \( \text{args} \) are the argument variables to the EIN operator

In an EIN operator, \( \sigma \in (\mathbb{Z} \times \text{INDEXVAR} \times \mathbb{Z})^* \) is a sequence of triples \((lb, i, ub)\), where \( lb \) is the lower bound on \( i \), \( ub \) is the upper bound, and each variable in \( \sigma \) is unique. For example, the outer product between two \( n \)-length vectors \((u \otimes v)\) is represented in High-IR as the assignment

\[
    t_1 = \lambda (U, V) \langle U_i V_j \rangle_{\sigma}(u, v) \quad \text{where} \quad \sigma = \{1 \leq i \leq n, 1 \leq j \leq n\}
\]

We might combine bounds when they are the same (e.g., \( \sigma = \{1 \leq i, j \leq n\} \)). We will omit both bounds for brevity (e.g., \( \sigma = \{\hat{i}, \hat{j}\} \)) in cases where they are unimportant. That is, if \( i \) is an index, we will write \( \hat{i} \) to denote its corresponding triple.

One way to think of EIN expressions is that they are a compact way to represent the loop nest that computes their result (although in our compiler we usually unroll their implementation). For example, the trace of a square matrix is defined as

\[
    t_2 = \lambda T \left( \sum_{\sigma} T_{ii} \right)(t_1) \quad \text{where} \quad \sigma = \{\hat{i}\}
\]

Here the summation operation represents a loop indexed by \( i \). This example illustrates that index variables can be bound inside EIN expressions by a summation operator (as well at the EIN operator level).

As part of the translation process, EIN operators are composed to form larger EIN expressions, which the compiler simplifies using rewrite rules (see Section 3.1.2). For example,
the expression $\text{trace}(u \otimes v)$ will automatically reduce the expression to

$$t_2 \implies \lambda (U, V) \left( \sum_i U_i V_i \right)(u, v)$$

thus discovering the identity: $\text{trace}(u \otimes v) = u \cdot v$.

EIN operators provide a mathematically sound and compact representation for tensor and field operations. This section introduces the notation used to represent the tensor and field operations in the Diderot compiler that replaces the old direct-style IR. The grammar of EIN operators is given in Figure 2.1 (we omit the full set of arithmetic operators for compactness).

**Types of Subscript**  A key aspect of the EIN IR is the tracking of indices. Indices can either be variables (denoted by $i$, $j$, and $k$), or constants ($n \in \mathbb{N}$). We also use $\alpha$ and $\beta$ to denote sequences of zero or more indices of either type. A variable index can either be bound in a summation operator or as one of the indices that determine the shape of the EIN operator’s result. EIN expressions include tensors ($T_\alpha$), and fields ($F_\alpha$); the latter two forms have multi-index subscripts that specify the individual component for higher-order shapes. The Levi-Civita symbol ($\varepsilon_{ij}$, $\varepsilon_{ijk}$) and Kronecker delta ($\delta_{ij}$) are used to permute and cancel components based on their indices. Summations have the usual semantics ($\Sigma_{\sigma} e$).

In the remainder of this section, we illustrate the constructs of the EIN operator by example. Consider the EIN operator

$$\lambda \bar{x} \left( \sum_{\sigma'} e \right)_{\sigma} \quad \text{where } \sigma = \{1 \leq i \leq n\} \text{ and } \sigma' = \{c \leq j \leq d\}$$

which has two indices $i, j$. The bound index $i$ ranges from $1$ to $n$ and gives the expression its shape (*i.e.*, a vector in $\mathbb{R}^n$). The *summation* index $j$ ranges from $c$ to $d$. Each component in the resulting vector binds index $i$ and $j$ and evaluates $e$. 

18
Fields and Tensors  EIN expressions describe Fields $F$ and Tensors $T$ similar to traditional index notation.

- A field and tensor EIN expression is appended with a list of indices that refer to the size of the tensor or field. A scalar field is expressed as $F$, a vector field is expressed as $F_i$, $T_{ij}$ is a matrix, and $T_{ijk}$ is a third-order tensor.
- The following are a few examples changing the type, ordering, and binding of two indices $M_{ij}$ on a 2x2 matrix $M$. Using two variable indices

$$
\lambda T\langle T_{ij}\rangle_{ij}(M) = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}
\lambda T\langle T_{ji}\rangle_{ij}(M) = \begin{bmatrix} M_{11} & M_{21} \\ M_{12} & M_{22} \end{bmatrix}
$$

Changing one index to a constant index will slice the matrix

$$
\lambda T\langle T_{2i}\rangle_{i}(M) = \begin{bmatrix} M_{21} & M_{22} \end{bmatrix}
\lambda T\langle T_{i1}\rangle_{i}(M) = \begin{bmatrix} M_{11} \\ M_{21} \end{bmatrix}
$$

Binding the indices to a summation expression creates the trace operation

$$
\lambda T\left(\sum_i T_{ii}\right)(M) = M_{11} + M_{22}
$$

- In lieu of individual indices to specify a shape, an $\alpha$ could be used to represent generic fields and arbitrary-sized tensors ($F_\alpha$ and $T_\alpha$). A transformation that applies to $T_\alpha$ can be applied to all tensors, regardless of shape. One of the benefits of EIN is being able to describe terms with arbitrary shapes, rather than having to handle special cases of shape-specific direct-style operators (as discussed in Section 2.4.1).

Differentiation  The expression $\frac{\partial}{\partial \alpha} \diamond e$ denotes differentiation on field $e$. The concise representation of differentiation allows us to express a wide variety of differentiation operations including the gradient ($\nabla$), divergence ($\nabla \cdot$), curl ($\nabla \times$), and Jacobian ($\nabla \otimes$).
κ ::= \text{sine, arcsine, ... arctangent} \quad \text{Trig functions}

\begin{align*}
\begin{array}{ll}
e ::= & T_\alpha, A_\alpha, B_\alpha \\
& F_\alpha, G_\alpha \\
& \delta_{ij} \\
& \varepsilon_{ij}, \varepsilon_{ijk} \\
& \sum e \\
& \frac{\partial}{\partial x_\alpha} \circ e \\
& e_1 @ e_2 \\
& \text{probe of a field } e_1 \text{ at position } e_2 \\
& \text{lift}(e) \\
& V_\alpha \ast H_\beta \\
& \sqrt{\varepsilon}, -\varepsilon, \exp(e), e^n, \kappa(e) \\
& e + e, e - e, \frac{e}{e}, ee
\end{array}
\end{align*}

\text{Tensor}

\text{Field}

\text{Kronecker deltas}

\text{Levi-Civita tensor}

\text{Summation}

\text{Derivative of } e

\text{Variable index}

\text{Constant index}

\text{Single index}

\text{Sequence of indices}

\text{EIN Operator}

\text{index mapping}

\text{Unary operators}

\text{Binary operators}

Permutation tensor and Kronecker delta The $\varepsilon_\alpha$ and $\delta_{ij}$ expressions are the permutation (or Levi-Civita) tensor and Kronecker delta function, respectively, and are defined in Section 1.2.1. We use these to model index-dependent tensor operations.

Arithmetic Terms Standard arithmetic operations are a natural fit to integrate into EIN.

Field Terms There are several forms of EIN expressions that are special to fields. The probe operator $e_1 @ e_2$ applies field $e_1$ to the point $e_2$ and \text{lift}(e) lifts a tensor $e$ in a field expression. The convolution expression $V_\alpha \ast H_\beta$ is the convolution operation of an image field $V$, with the range $\alpha_1$ and a piecewise polynomial kernel $H$. 

Figure 2.1: The syntax of EIN operators’ E and EIN expressions e in High-IR
2.2 Generating EIN Operators

The development of EIN notation has made Diderot a richer language. Additional support for operators allowed a flexibility for the user to write computations without ambiguous restrictions. In this section we describe the implementation process to create EIN operators (Section 2.2.1) and some of the benefits (Section 2.2.2).

2.2.1 Implementation

To create EIN operators we use EIN expressions as building blocks. The EIN operator can represent a generic operation that needs to be instantiated or a specific operation. In the following we describe these implementation details.

Building Blocks When creating EIN operators we may use EIN expressions as building blocks. They can be used together to represent a range of computations. For instance, the multiplication EIN expression $A_\alpha B_\beta$ could be used in represent the outer product (left) or modulate (right) operator.

$$\lambda (A, B) \langle A_i B_j \rangle_i^j \quad \lambda (A, B) \langle A_i B_i \rangle_i^i$$

The distinction between the two exists in the binding of the variable indices. EIN expressions could be used as building blocks to describe more intricate operators such as the 2-d matrix inverse.

$$\lambda F \left\langle \frac{(\sum_k F_{kk}\delta_{ij}) - F_{ij}}{F_{00} F_{11} - F_{01} F_{10}} \right\rangle_i^j$$

Instantiate generic operations The Diderot compiler generates High-IR, including the EIN operators, from an explicitly typed simplified AST representation. For many related operations, we can define a generic (i.e., shape polymorphic) EIN operator that gets specialized based on its type. A simple example is the tensor addition operator, which works on
tensors of any shape. We define a generic tensor addition operation that is parameterized
over a multi-index meta-variable \( \alpha \) and using de Bruijn numbering:

\[
\Lambda \alpha \lambda (A, B)(A_\alpha + B_\alpha)_\alpha
\]

We specialize the operator to a particular shape by replacing \( \alpha \) with a multi-index that
ranges over the shape.

**Specific operators** At times we might choose to create a EIN operator that refers to a
specific surface-level operation and does not need to be instantiated. Often the operator
body might use constant indices. Constant indices are used to illustrate specific components
of tensors and tensor fields. Our implementation of the 2-d determinant:

\[
\lambda T(T_{00}T_{11} - T_{01}T_{10})
\]

relies on specific components of the tensor to be computed. It is mapped to the 2-d deter-
minant only.

### 2.2.2 Advantages

The implementation of EIN operators has certain advantages. We have a concise represen-
tation for a family of operators. This makes it easier to add generality to existing operators.
We added “lifted” versions of tensor operators which enables a higher-order programming
model. We added new operators to the surface language. We discuss the benefits in more
detail in the following section

**Family of Operations** It can be easy for a single generic EIN operator to represent a
family of operators. Consider the inner product operator \( \bullet \) applied to two tensors. It has
the generic definition:
\[
\lambda (A, B) \langle \sum_{k} A_{\hat{\alpha}k} B_{k\hat{\beta}} \rangle_{\hat{\alpha}\hat{\beta}}
\]
where \(\alpha\) and \(\beta\) are specialized to handle different shapes. The inner product between a vector \(A\) and a matrix \(B\) is realized by instantiating \(\alpha\) to the empty multi-index and \(\beta\) to a single index.
\[
\Rightarrow \lambda (A, B) \left\langle \sum_{k} A_{k} B_{ki} \right\rangle_{i}
\]
This is a concise representation of a range of different operators. EIN makes it easy to add generic versions of operators and support for arbitrary-sized tensor operators with less code.

**Lifted Tensor Operations** We want the Diderot programmer to be able to define a field with a series of *lifted* operators on the surface language. We define that lifted tensor operators as follows:

**Tensor and Field operator** We define tensor operators \(P\) and field operators \(P^{\uparrow}\) as
\[
P : \text{tensor} \rightarrow \text{tensor} \quad P^{\uparrow} : \text{field} \rightarrow \text{field}
\]
We refer to a lifted operation as
\[
P^{\uparrow}(f) = \lambda x. P(f(x))
\] (2.1)
The EIN IR makes implementing lifted operators (*i.e.*, tensor operations lifted to work on fields Equation 2.1) much easier. The inner product operator \(\bullet\) can be lifted to work on fields. Similar to the previous example, the inner product between two fields has the generic definition which can be instantiated by the type checker
\[
\lambda (F, G) \left\langle \sum_{k} F_{\hat{\alpha}k} G_{k\hat{\beta}} \right\rangle_{\hat{\alpha}\hat{\beta}} \Rightarrow \lambda (F, G) \left\langle \sum_{k} F_{k} G_{ki} \right\rangle_{i}
\]
It is also possible to add the application of the operator between a tensor and a tensor field.

\[ \lambda(T, G) \left\langle \sum_{\hat{k}} T_{\alpha k} G_{k\beta} \right\rangle_{\hat{\alpha}\hat{\beta}} \]

The flexibility to create application of tensor operators between and on tensor and tensor fields makes Diderot a more complete and flexible language.

**Adding to the surface language** The work in this chapter has enabled new operators in the Diderot language. We can add new tensor operators and lifted versions of them. This includes shape-specific operators such as the inverse of 2-by-2 matrices (shown earlier), scalar trig operators (\( \cos(\cdot) \)), and the 3-d curl (\( \nabla \times \)).

\[ E_{\text{trigF}} = \lambda F \langle \kappa(F) \rangle \quad E_{\text{curl3}} = \lambda F \left\langle \sum_{jk} E_{ijk} \frac{\partial}{\partial x_j} F_k \right\rangle_i \]

We can include support for arbitrary sized tensors and tensor fields such as normalize (\( \text{normalizeF} \)) and divergence (\( \nabla \cdot \)).

\[ E_{\text{normalizeF}} = \lambda F \left\langle \frac{F_{\alpha}}{\sqrt{\sum_{\beta} F_{\beta} F_{\beta}}} \right\rangle_{\hat{\alpha}} \quad E_{\text{divergence}} = \lambda F \left\langle \sum_j \frac{\partial}{\partial x_j} F_{\alpha j} \right\rangle_{\hat{\alpha}} \]

The new EIN operator includes index-dependent operations. A few examples of 3-d EIN operators that use the permutation tensor include the cross product (\( T \times G \)) and the determinant (\( \text{det}(F) \)).

\[ E_{\text{crossTF}} = \lambda T, G \left\langle \sum_{jk} E_{ijk} T_j G_k \right\rangle_i \quad E_{\text{det3x3F}} = \lambda F \left\langle \sum_{ijk} (F_{0i} F_{1j} F_{2k} E_{ijk}) \right\rangle \]
2.3 Field Reconstruction

This section will illustrate field reconstruction. During the transition from High-IR to Mid-IR, higher order constructs get replaced by lower-order constructs. Probed fields \( v \odot h(x) \) are replaced with terms to directly express computations on images and kernels. We continue using the notation defined in Section 1.2.

**Design** Traditional index notation \([29]\) does not provide the notation needed to show the reconstruction of fields. The EIN expressions \( v_\alpha(\bar{e}), \text{val}(i), \) and \( h^{\bar{\psi}}(e) \) are introduced to represent reconstructed fields inside the Diderot compiler. The expression \( v_\alpha \) denotes an image field. The expression \( v_\alpha[\bar{e}] \) is an image field indexed at a list of integer positions \( \bar{e} \). The specific axis for the fractional \( f \) and integer \( n \) position are represented with a constant index. The \( \text{val}(i) \) notation lifts an index variable to a constant integer; \( e.g., \sum_{i=0}^{n} \text{val}(i) = 0+1+2+\cdots+n \). We use the notation \( h^n \) to refer to the \( n \)th derivative of univariate function \( h \). In EIN expression \( h^{\bar{\psi}} \) the level and type of differentiation is captured in the \( \bar{\psi} \), which is a list of pairs \([ (c, i_1), \ldots, (c, i_m) ] \) that are evaluated like Kronecker-deltas pairs \( (i.e., \psi = (c, i) = \delta_{c, i}) \) and added together.

**Implementation** We build on the exposition from our previous work \([16]\), reproduced here for convenience, to explain the context and contribution of EIN. Let \( \mathbf{F} \) be a 2-d vector field defined as follows:

```plaintext
field #0(2)[2] F = tent \odot \text{img(“i.nrrd”)};
vec2 out = F(p);
```

The output of probing vector field is evaluated as

\[
\begin{bmatrix}
\sum_{ij:1−s} v_0[n_0 + i, n_1 + j]h(f_0 - i)h(f_1 - j) \\
\sum_{ij:1−s} v_1[n_0 + i, n_1 + j]h(f_0 - i)h(f_1 - j)
\end{bmatrix}
\]
In High-IR the operation is represented as a single EIN operator

\[
\text{out} = \lambda (V, H, T) \langle V_i \odot H(T) \rangle_{\{i:2\}} (F, \text{tent}, p) \tag{2.2}
\]

The field (2.2) is reconstructed in EIN notation as

\[
\lambda (v, h, n, f) \left\{ \sum_{jk=1-s}^s v_\alpha [n_0 + \text{val}(j), n_1 + \text{val}(k)] h(f_0 - \text{val}(j)) h(f_1 - \text{val}(k)) \right\} (F, \text{tent}, n, f)
\]

We present a new notation that tracks the differentiation component.

\[
\nabla_i H(x, y) = h^{\delta_{0i}(x)} h^{\delta_{1i}(y)}
\]

A second derivative adds another variable index:

\[
\nabla_{ij} H(x, y) = h^{\delta_{0i} + \delta_{0j}(x)} h^{\delta_{1i} + \delta_{1j}(y)}
\]

Generally, the operation is of the form

\[
\nabla_{\alpha_0, \alpha_1, \ldots, \alpha_n} H(x, y) = h^{\delta_{0\alpha_0} + \delta_{0\alpha_1} + \ldots + \delta_{0\alpha_n}(x)} h^{\delta_{1\alpha_0} + \delta_{1\alpha_1} + \ldots + \delta_{1\alpha_n}(y)} \tag{2.3}
\]

As discussed in Section 1.2, the result is a value in image-space, not in world-space, and it must be transformed back to world-space with transformation matrix \( P \), where \( P = [M^{-T}] \). A multiplication by \( P \) is needed for each index of convolution measured in the derivatives.

More generally the transformation back to world-space has the form:

\[
\nabla_{\alpha_0, \alpha_1, \ldots, \alpha_n} F \longrightarrow \sum_{\beta} P_{\alpha_0 \beta_0} P_{\alpha_1 \beta_1} \ldots P_{\alpha_n \beta_n} (\nabla_{\beta} F)
\]
Example  The Hessian of a scalar field \(\nabla \otimes \nabla \phi(x)\) is represented with EIN operator

\[
G(i, j, k, l) = \sum_{kl=1}^{s} v[n_0 + \text{val}(k), n_1 + \text{val}(l)]h^{\delta_{0i} + \delta_{0j}}(f_0 - \text{val}(k))h^{\delta_{1i} + \delta_{1j}}(f_1 - \text{val}(l))
\]  

(2.4)

\[
t_0 = \lambda(v, h, n, f)\langle G(i, j, k, l) \rangle_{ij}(\text{img, bspln3, n, f})
\]

(2.5)

\[
\text{out} = P \cdot t_0 \cdot P
\]

(2.6)

In (2.5) we multiply by matrix \(P\) twice for each index in image-space to convert the result to world-space.

2.4 Discussion

In this section we discuss the design of the EIN IR. In Section 2.4.1 we describe the original direct-style version of the Diderot compiler and compare it our design. In Section 2.4.2 we discuss the design choice we made in creating new EIN operators. In Section 2.4.3 we describe the improved programmability.

2.4.1 The Case for a New IR

Our initial implementation of Diderot used a direct representation of tensor operations (\(i.e.,\) tensor operations, such as \(\nabla\), were primitive operators) in its intermediate representation (IR) [16]. Using a series of lowering transformations combined with standard compiler optimizations, the representation was translated into a simple vectorized language, from which we generated C or OpenCL code. While sufficient to prototype the design ideals of Diderot, the first version of the Diderot compiler suffered from several limitations and was unable to illustrate a large range of programs. This section defines the direct-style compiler, the design advantage of EIN over the existing IR to creating an expressive IR, and representing certain computations.
Direct-Style: The first design of Diderot used direct-style notation, noted in the following as OP. In direct-style we treat operators as opaque operations, that are later reduced to lower level primitives. As an example, the inner product of a two 2-d vectors is represented in direct-style as OP_InnerP_VecVec (u,v).

\[ u \cdot v = (u[0] * v[0]) + (u[1] * v[1]) \] (2.7)

Direct-style notation gives a compact representation, but requires more tensor-shape specific operators (e.g., OP_InnerP_VecMat(u,m), OP_InnerP_MatMat(m,m), ...). EIN aims to be as compact as direct-style notation while revealing internal details to translate and optimize a broader range of operators.

Expressive IR While the direct-style version of the compiler provides an expressive language for image analysis and visualization, it is lacking when trying to develop algorithms that rely heavily upon higher-order operations. Diderot could not easily support tensor operators P on the field level \( P^\uparrow \) (2.1). In order to lift a tensor operator to the field level (e.g., \( F \bullet G \)) we would have to define a similar set of shape-specific operators but for the different field types. Each of these new operations in the compiler would need special-handling to be translated and optimized, adding complexity to compiler transformations.

The normalization process must also deal with the combination of probes and differentiation with the lifted operators. Our earlier implementation used direct-style tensor and field operators in the High-IR with specific rewrite rules to handle the various combinations of operations (e.g., \( \nabla(e_1 + e_2) \Rightarrow \nabla e_1 + \nabla e_2 \)). This approach suffered from a combinatorial blowup in the number of rules, which made it difficult to add new lifted operators and it did not support index-dependent operations in a general way.

The direct-style approach for applying the differentiation operator is adequate for the basic differentiation of scalar and tensor field (\( \nabla, \nabla \odot \)) but it does not easily generalize to the full range of higher-order operators that we would like to support. For instance, the
divergence cannot be supported with direct-style rewriting (equation Equation 1.8); i.e., \( \nabla \cdot F \implies \nabla \cdot (V \otimes H) \neq V \otimes H^1 \). In this dissertation we describe a better approach to representing tensor and tensor field operations that has allowed us to greatly enrich the expressiveness of Diderot.

**Index-Dependent Operators** Direct notation (like scaling vector field F with 4*F) completely avoids dependence on any particular choice of basis for representing tensor components, while index-dependent means the semantics of the operation (like curl or determinant) explicitly refer to individual components. In other words, direct-style operators are index-free, but there are certain operations, such as the curl of a vector field, whose semantics depends on the indices of components.

To see the problem consider a vector field \( F \) and let \( F_i \) indicate the \( i^{th} \) axis of \( F \). Differentiating the 3-d curl of \( F \) (\( \nabla \otimes (\nabla \times F) \)) is needed for shading renderings of curl-related quantities. Mathematically, the computation is represented as:

\[
\begin{bmatrix}
\frac{\partial^2}{\partial x \partial y} F_2 - \frac{\partial^2}{\partial x \partial z} F_1, & \frac{\partial^2}{\partial y \partial y} F_2 - \frac{\partial^2}{\partial y \partial z} F_1, & \frac{\partial^2}{\partial z \partial y} F_2 - \frac{\partial^2}{\partial z \partial z} F_1 \\
\frac{\partial^2}{\partial x \partial z} F_0 - \frac{\partial^2}{\partial x \partial x} F_2, & \frac{\partial^2}{\partial y \partial y} F_0 - \frac{\partial^2}{\partial y \partial x} F_2, & \frac{\partial^2}{\partial z \partial y} F_0 - \frac{\partial^2}{\partial z \partial x} F_2 \\
\frac{\partial^2}{\partial x \partial x} F_1 - \frac{\partial^2}{\partial x \partial y} F_0, & \frac{\partial^2}{\partial y \partial x} F_1 - \frac{\partial^2}{\partial y \partial y} F_0, & \frac{\partial^2}{\partial z \partial x} F_1 - \frac{\partial^2}{\partial z \partial y} F_0
\end{bmatrix}
\] (2.8)

As can be seen from the matrix above, the terms refer to components of the field and partial differentiation operators, they are index-dependent. Since direct-style operators are opaque with respect to the component indices, they cannot express these sorts of operations. Lacking a way to describe individual indices, the previous direct-style IR could not handle compositions of index-dependent operations except in ad hoc ways. We handled this restriction in the direct-style compiler, by limiting the differentiability of the range of these operators. For example, we give the 3D curl operator the type

\( \nabla \times : field\#k(3)[3] \rightarrow field\#0(3)[3] \)
instead of the mathematically correct type

\[ \nabla \times : field\#k(3) [3] \rightarrow field\#(k-1)(3)[3] \]

By giving the result a differentiability of 0, we prevent it from being used as the argument of a differentiation operator and, thus, our direct-style compiler can handle it. EIN can handle it and presents a compact representation

\[ t = \lambda F \left\{ \sum_{1 \leq k, l \leq 3} \mathcal{E}_{ikl} \frac{\partial}{\partial x^j} F_l \right\}_{\sigma} (F) \quad \mathcal{\sigma} = \{ 1 \leq i, j \leq 3 \} \quad (2.9) \]

### 2.4.2 EIN as building blocks

In this section we will discuss if using EIN expressions as building blocks is the right choice for more complicated operations. The EIN IR has been used to represent a range of tensor operations. Each time a new operator is implemented in EIN we first consider using existing EIN expressions to create it. By using the existing EIN IR it is easy to extend the language because there is less to maintain. The rest of the compiler can then handle these EIN expressions generically, instead of enduring the implementation costs of adding a completely new constructor.

If we are representing more complicated operations then we might consider creating a new operator, referred in the following as “block-like structure”. Consider possible representations of the 3-d determinant and 2-d inverse.

**EIN expressions**

- \( E_{\text{det}-3d} \left\{ \sum_i (F_{0i} \sum_j (F_{1j} \sum_k (F_{2k} \mathcal{E}_{ijk}))) \right\}_{ijk} (\text{DET}-3d(F))_{ij} \)
- \( E_{\text{inv}-2d} \left\{ \frac{(\sum_k F_{kk} \delta_{ij}) - F_{ij}}{F_{ii} F_{jj} - F_{ij} F_{ji}} \right\}_{ij} (\text{INV}-2d(F,ij))_{ij} \)

The block structure can offer a concise representation in place of a more complicated EIN operation. They could enable new rewrites such as \( \text{Inv} \left( \text{Inv}(e) \right) \rightarrow e \), which could be
significant given a large $e$. One setback is that their careful design would still need to keep track of EIN indices when the result of the operation is a non-scalar. The tracking of indices in block-like structures could add more complexity to rewriting system rather than simplicity.

While the block-like structure can offer a printer-friendly representation, the concise representation may not be sustainable. In fact the block-like structure may inevitably be broken into smaller pieces for three reasons. For one, realizable field terms need to be identified before field reconstruction (Section 2.3). Secondly, new rewrite rules might lead to a term that is difficult to express in the block structure. Lastly, decomposing block-like structures into smaller pieces may be the best way to apply compiler optimizations (such as finding common subexpressions before reducing to lower level constructs in Section 3.2). Considering, the limitations and the implementation costs to adding to the IR we choose to represent new EIN operators with existing EIN expressions, but in the future it would be interesting to evaluate the choice further.

2.4.3 Programmability

By enabling lifted operations, the Diderot code can refer to its fields and its spatial derivatives in a mathematically idiomatic way. For instance, Canny edges [10] find optimal smoothing and edge components where the image gradient magnitude is maximized with respect to motion along the (normalized) gradient direction. The rich language allows the computation core of the concept to be used directly in a field definition.

$$\text{field}\mathbf{#2(3)[]}C = -\nabla (|\nabla F|) \cdot \nabla F / |\nabla F|;$$

$$\text{field}\mathbf{#k(3)[}\sigma]\mathbf{D} = \nabla C \ldots;$$

The implementation of the EIN IR greatly simplifies the expression for the quantity of interest.

There is a bottleneck to implementing new ideas in a scientific visualization program. Diderot offers a fast way to write new ideas without having to worry about the low-level details of the code. The work in this chapter expands the expressiveness of the Diderot
language and pushes the boundaries of the type of programs can be written. It is easier for a user to develop algorithms that rely heavily upon higher-order operations. Section 7.2 demonstrates visualization applications enabled by the work in this chapter.
CHAPTER 3
IMPLEMENTATION TECHNIQUES

Developing the EIN IR has made Diderot a more complete language. Code can be written in a high level and more accurately represent how a user thinks about her ideas. By writing at a high level, the user is relying on the compiler to do the necessary derivations, such as applying the differentiation operator, and doing field reconstruction.

The implementation of EIN has raised new technical challenges. Part of the implementation process includes rewriting a computation on high level field terms so that they are in a form that the compiler can recognize and translate to lower level field terms. Fields terms need to be in normalized before field reconstruction (Section 1.2.2). Field normalization includes applying rewrites to push the probe operator down directly around field terms. All EIN expressions with operations at the field level need to be normalized in this way. The process of field normalization motivates the normalization of EIN expressions.

The normalization of EIN expressions is handled generically (Section 3.1). The normalization is done, by systematically composing EIN operators and normalizing their bodies. The rewriting system then applies domain-specific rewriting and simplifications. Our generic implementation of EIN makes it easy to add new operators to the language without having to write much additional code.

A naive implementation of EIN created an unacceptable size in the IR. As a result programs took a long time to compile or did not compile at all. This problem significantly limited the use of the full expressivity of the language. The problem occurs because the normalization process creates a large size when trying to represent the more math-intensive programs enabled by the EIN IR. During normalization the IR expands quickly and makes large complicated expressions. The next phase of the compiler transforms the fields terms into a large number of lower-level operators. To create a smaller IR we want to translate a smaller number of field terms by recognizing common computations inside the large complicated EIN expression. EIN notation adds a level of complexity to reducing expressions by a
sub-term because the mathematical meaning of the computation has to be maintained.

To address the size problem, we have developed a number of techniques to keep the size under control (Section 3.2). We look at an EIN expression in context of the full EIN operator. Is then we can fully compare computations to determine equality for reductions and additional optimizations. We describe the compilation techniques and provide examples.

To evaluate the impact and cost of the compilation techniques we measure the compilation and execution time for various benchmarks (Section 3.3). For the benchmarks we use a mix of Diderot programs with varying degrees of tensor math. We measure the impact of the compilation techniques at different levels of abstraction. The techniques can be controlled or restricted with command-line flags. Overall, the implementation of EIN can allow programs with higher level of math than was previously possible.

This chapter describes the implementation of EIN IR. We describe the normalization process in Section 3.1. We introduce the compilation techniques we developed in Section 3.2. Lastly we evaluate the techniques in Section 3.3.

### 3.1 Normalization

This section describes how EIN operators are normalized. First we use a systematic approach to apply EIN operators to each other, called substitution. Substitution creates a single EIN operator that is easier to normalize. The rewriting system then rewrites the body of the EIN operator to normalize the EIN expressions. The rewriting system applies field-normalization, domain-specific rewriting, index-based rewriting, and other rewriting that can optimize or reduce the terms.

In Section 3.1.1 we describe substitution process. In Section 3.1.2 we present the rewriting rules. This composition of operators and rewriting can create a very large expression. We discuss and address this size problem in further detail in Section 3.2.
3.1.1 Substitution

The process of substitution involves composing EIN operators into one operator that represents the computation. The substitution step is applied in order to simplify the process of rewriting. In this section we describe the implementation of the substitution process followed by an example.

**Brief overview**  Consider the following brief overview of the substitution process. Given two EIN operators where B is an argument A this process will substitute the body of operator B inside A.

\[
B = (P) \langle e \rangle_\alpha (ps) \\
A = (T, X) \langle T_\alpha X \rangle_\beta (B, xs)
\]

The body \(e\) of (the EIN operator of) the argument (B) is used to replace the term with the corresponding parameter identity \(T_\alpha\) in (the body of the EIN operator) A.

\[T_\alpha X \rightarrow eX\]

The arguments of both operators are merged. The EIN expression in the original operator needs to be rewritten in order to reflect the new list of arguments and maintain the same mathematical meaning of the original computation. As a result a single EIN operator represents the computation.

\[\text{out} = (X, P)\langle e'X' \rangle_\beta (xs, ps) \text{ where } e \rightarrow e' \text{ and } X \rightarrow X'\]

**How it is implemented**  We describe the process in more detail in the following text. The following code is written in SML. We break the code up to mark the different steps and use comments to remark the smaller details. We omit some code for conciseness but describe the overall strategy in the comments. We use notation such as “e.body” to refer to the ”body” of EIN operator e. We include EIN notation in brackets for clarity.
The substitution function composes two EIN operators. As input the function receives the outer EIN operator \((e_1)\), inner/replacement EIN operator \((e_2)\), and a parameter identity/argument number \((\text{substId})\). It returns a single EIN operator to represent the computation.

\[
\text{fun substitution} \ (e_1, \ \text{substId}, \ e_2) = \ \text{let}
\]

The parameters of an EIN operator indicate how an argument is represented inside the EIN expression. Each tensor and field term indicate their corresponding parameter with a parameter id (notes as id in the code). During substitution the parameters and arguments of both operators are merged (omitting the one that is currently being replaced). The parameter identities in the EIN body then need to be remapped.

\[
(* \text{getParams() uses parameters from the EIN operators} \ *)
\]
\[
\text{val (params', paramMapp) = getParams (e1.params, e2.params, substId)}
\]
\[
\text{fun rewriteParams e} = (* rewrite parameter ids using paramMap*)
\]

The following functions scans the body of the outer EIN expression \((e_1)\) and looks for terms with a matching parameter id. When it finds it replaces the matching term with the body of the substitution \((e_2)\). The body of \(e_1\) is updated using the parameter map to reflect the new list of arguments.

\[
(* \text{Look for matching parameter ids that match substitution*})
\]
\[
\text{fun rewrite} \ (id, \ \alpha, \ e) =
\]
\[
\text{if (id==substId)}
\]
\[
\text{rewriterSubst (\alpha)}
\]
\[
(* \text{replaces term with the updated body of } e_2 \ *)
\]
\[
\text{function described next.}
\]
\[
(* \text{Scan body*})
\]
\[
\text{fun apply e} = \ \text{(case e)}
\]
\[
(* \text{Look for matching parameter id } \{T_{\alpha}, F_{\alpha}\}*)
\]
\[
\text{of E.Tensor(id, } \alpha) \Rightarrow \text{rewrite(id, } \alpha, \ e) \]
\[
| \ E.Field(id, \alpha) \Rightarrow \text{rewrite(id, } \alpha, \ e) \]
The following function takes the substitution term (the body of e₂) and rewrites it. It considers the parameter map and the tensor/field term it is replacing. The indices in the substitution term are instantiated by the (indices of the) term it is replacing (α). Again, the parameter ids in the body need to reflect the new list of arguments.

(* This method rewrites substitution.
* It remap parameter identities and variable indices.
*)

fun rewriteSubst α = let

  val mapBeta =
  (*create a map for the variable indices using α*)
  fun rewriteBeta β =
  (*use map mapBeta for variable indices*)
  fun rewriteParam id =
  (* use map for parameter ids*)
  fun rewrite = (case e
  (* {T_β,F_β}*)
    of E. Tensor (id, β)
      => E. Tensor (rewriteParam id, rewriteBeta β)
    | E. Field (id, β)
      => E. Field (rewriteParam id, rewriteBeta β)
    | E. Partial (β) (* {∂/∂β}*)
      => E. Partial (rewriteBeta β)
  (*look at subterms*)
    | E. Op1 (op1, e₁)
      => E. Op1 (op1, rewrite e₁)
    | E. Op2 (op2, e₁, e₂)
      => E. Op1 (op1, rewrite e₁, rewrite e₂)
    | E. Sum
      (*Variable indices bound by a summation operator
       *are bumped up to the next unbound index.*)
  ... (*end case*)
  in rewrite (e₂.body) end
The body and parameters in the outer operator are updated and returned.

```ml
val body' = apply (e.body)
in {λparams' (∧ body')\ e1.index} end
```

**Example**  In the following example we use arbitrary-sized tensors to demonstrate a general approach of substitution with arbitrary-sized tensor operators. The outer product and addition between tensor arguments.

```ml
tensor a[ς]; tensor b[ς]; tensor c[σ];
tensor t1[ς] = a + b;
tensor t2[σς] = c ⊗ t1;
```

expressed in High-IR as two EIN operators

```
t1 = λ(A, B) ⟨Aγ + Bγ⟩_γ (a, b) where γ = (1 ≤ γ ≤ ς)
t2 = λ(C, T) ⟨CαTβ⟩_αβ (c, t1) where α = (1 ≤ α ≤ σ) and β = (1 ≤ β ≤ ς)
```

We use the variable \( \hat{γ} \) as a shorthand for the following definition. Every variable index \( γ_i \) in \( γ \) is bounded from 1 to \( ς_i \). \( ς \) is the tensor shape to variables \( a, b, \) and \( t_1 \). A similar explanation can be used to define \( \hat{α} \) and \( \hat{β} \).

A substitution is made by replacing the term \( T_β \) with the body of the EIN operator \( t_1 \). The variable indices in the body of \( t_1 \) are remapped to \( T_β \). The result is a single EIN operator to represent the two EIN operators.

```
t2 → λ(C, A, B) ⟨Cα(Aβ + Bβ)⟩_σς (c, a, b) where γ is initialized by β
```

### 3.1.2 Rewriting

Our transformation rules serve to do necessary rewriting and to simplify EIN terms. Necessary domain-specific rewriting can include field normalization to create normalized field forms (Section 1.2.2). We demonstrate rewriting based on tensor calculus that are used to
apply the differentiation operator on EIN expressions. We also describe some simplifications that are made based on index-based rewrites and algebraic rewrites.

In the following section we present the rewrites organized by area; domain-specific, differentiation, index-based rewriting, and algebraic rewriting. We use the notation from the design chapter to represent EIN expressions. The transformations \( e_1 \implies e_2 \) are indicated with a ‘\( \implies \)’ to indicate a term \( e_1 \) is rewritten to term \( e_2 \).

**Domain-specific rewrites** Domain-specific rewrites involve pushing a probe operation towards the field terms. The probe operation is pushed past unary operators.

\[
(-F_\alpha)@x \implies -(F_\alpha@x) \quad \sqrt{F_\alpha}@x \implies \sqrt{F_\alpha@x} \quad \sin(F_\alpha)@x \implies \sin(F_\alpha@x)
\]

The probe operator is distributed over binary operators.

\[
(e_1 + e_2)@x \implies (e_1@x) + (e_2@x) \quad (e_1 * e_2)@x \implies e_1@x * e_2@x
\]

\[
(e_1 - e_2)@x \implies (e_1@x) - (e_2@x) \quad \frac{e_1}{e_2}@x \implies \frac{e_1@x}{e_2@x}
\]

The probe of a non-field term is reduced.

\[
\text{lift}(e)@x \implies e \quad \delta_{ij}@x \implies \delta_{ij} \quad \mathcal{E}_{ij}@x \implies \mathcal{E}_{ij} \quad \mathcal{E}_{ijk}@x \implies \mathcal{E}_{ijk}
\]

By pushing the field expression past tensor operators and down to the field term, the operators are computed on the tensor result of a probe rather than the entire field. Consider scaling field \( F_\beta \) by scalar \( s \):

\[
(sF_\beta)(x) \implies s(F_\beta(x))
\]

We simplify the differentiation of a field term by moving the indices on the differentiation
operator onto the kernel inside a convolution expression.

\[ \frac{\partial}{\partial x_{\mu}} \circ (V_{\alpha} \otimes H^\beta) \implies V_{\alpha} \otimes H^\beta_{\mu} \]  

(3.1)

By using variable indices to represent differentiation (rather than just integers) we can represent different types of differentiation. The direct style version of this rule (Equation 1.8) limited the type of differentiation operators that can be supported.

**Differentiation**  The differentiation operator is applied to EIN expressions in order to push the differentiation operator down to the leaves of the field terms. The following EIN rewrites include the tensor calculus identities such as the product rule, quotient rule, chain rule, power rule,

\[ \nabla_{i} \circ (e^n) \implies \text{lift}(n) e^{n-1} (\nabla_{i} \circ e) \quad \nabla \circ (e_1 e) \implies e_1 (\nabla \circ e) + e (\nabla \circ e_1) \]

\[ \nabla_{i} \circ (\exp(e)) \implies \exp(e) (\nabla_{i} \circ e) \quad \nabla \circ \frac{e_1}{e_2} \implies \frac{(\nabla \circ e_1) e_2 - e_1 (\nabla \circ e_2)}{e_2^2} \]

\[ \nabla_{i} \circ (\sqrt{e}) \implies \frac{\nabla \circ e}{\text{lift}(2) \sqrt{e}} \]

doing the differentiation of constants,

\[ \nabla_{i} \circ (\text{lift}(0)) \implies \text{lift}(0) \quad \nabla_{i} \circ (\delta_{ij}) \implies \text{lift}(0) \]

and trigonometric identities.

\[ \nabla_{i} \circ (\cos(e)) \implies -\sin(e) (\nabla_{i} \circ e) \quad \nabla_{i} \circ (\arccos(e)) \implies \frac{\text{lift}(1) (\nabla \circ e)}{\sqrt{\text{lift}(1) - (e \ast e)}} \]

\[ \nabla_{i} \circ (\sin(e)) \implies \cos(e) (\nabla_{i} \circ e) \quad \nabla_{i} \circ (\arcsin(e)) \implies \frac{\text{lift}(1) (\nabla \circ e)}{\sqrt{\text{lift}(1) - (e \ast e)}} \]

\[ \nabla_{i} \circ (\tan(e)) \implies \frac{\nabla \circ e}{\cos(e) \cos(e)} \quad \nabla_{i} \circ (\arctan(e)) \implies \frac{\text{lift}(1) (\nabla \circ e)}{\text{lift}(1) + (e \ast e)} \]

The distribution over an arithmetic operator is generalized:

\[ \nabla_{i} \circ (-e) \implies - (\nabla_{i} \circ e) \quad \nabla_{i} \circ (e_1 - e_2) \implies (\nabla_{i} \circ e_1) - (\nabla_{i} \circ e_2) \]
It is easy to see how one concise rewrite rule

\[ \nabla_\alpha \diamond (e_1 + e_2) \Rightarrow \nabla_\alpha \diamond e_1 + \nabla_\alpha \diamond e_2 \]

can take the place of multiple direct-style rules such as

\[ \nabla \cdot (\varphi_1 + \varphi_2) \Rightarrow (\nabla \cdot \varphi_1) + (\nabla \cdot \varphi_2) \]
\[ \nabla \times (F + G) \Rightarrow (\nabla \times F) + (\nabla \times G) \]

The generality of our rewrites help simplify the total number of rewriting test cases that need to be created.

**Index based optimizations** The EIN IR allows us to do index-based reductions. Applying one of these optimizations removes at least one summation loop from the operation. For instance the variable indices on a differentiation operator could match two indices as an epsilon term.

\[ \mathcal{E}_{ijk} \nabla_{ij} \cdot e \Rightarrow \text{lift}(0) \]

This rewrite enables the compiler to find identities \( \nabla \times \nabla \varphi \Rightarrow 0 \) and \( \nabla \cdot \nabla \times F \Rightarrow 0 \). Two epsilons in an expression with a shared index can be rewritten to deltas [18].

\[ \mathcal{E}_{ijk} \mathcal{E}_{ilm} \Rightarrow \delta_{jl} \delta_{km} - \delta_{jm} \delta_{kl} \quad (3.2) \]

A \( \delta_{ij} \) expression can be applied to tensors, fields, and the del operator.

\[ \delta_{ij} T_j \Rightarrow T_i \quad \delta_{ij} F_j \Rightarrow F_i \quad \nabla_j \diamond \delta_{ij} e \Rightarrow \nabla_i \diamond e \quad (3.3) \]
Algebraic rewrites  The remaining rewrites are used to make simplifications and reductions.

\[-0 \Rightarrow 0 \quad e_1 - 0 \Rightarrow e_1 \quad 0 + e_2 \Rightarrow e_2 \quad 0 * e_1 \Rightarrow 0\]

\[\frac{e_1}{e_2} \Rightarrow \frac{e_1}{e_2} \quad \frac{e_1}{e_3} \Rightarrow \frac{e_1 e_3}{e_2} \quad \frac{e_1}{e_3} \Rightarrow \frac{e_1 e_4}{e_2 e_3} \quad \sqrt{(e)} \sqrt{(e)} \Rightarrow e\]  

(3.4)

3.2 Optimization and Transformations

As discussed in the earlier chapter, the property of field normalization motivates our need for normalization. We chose to simplify rewriting by first composing EIN operators and then applying rewrites to a single body. During normalization we replicate code and break sharing.

The code replication issue can be observed by the fact that rewriting can create multiple iterations of the same terms. Consider the expansion of the gradient of the norm of the gradient:

\[\nabla |\nabla e| \rightarrow \sum_j(t_1 t_2) + \sum_j(t_1 t_2) \]

\[\frac{2 \sqrt{\sum_j(t_1 t_1)}}{2} \]  

(3.5)

where \(t_1\) and \(t_2\) are the gradient and the Hessian of \(e\), respectively.

The expressions \(t_1\) and \(t_2\) are mentioned multiple times but ideally each would only be translated to lower-level constructs once. This approach is commonly known as common subexpression elimination (CSE). In order to apply CSE inside an EIN operator we must be able to compare EIN expressions efficiently and correctly.

EIN notation adds a level of complexity to comparing EIN expressions since so much of the meaning is captured in the indices (such as how a tensor or field is sample). In a single EIN operator a sub-term \(e\) may be used multiple times but we have to analyze the indices to see if it represents the same computation. Consider the expression:

\[e_{ij} e_{k1} - e_{0i} e_{jk} + (e_{ij} e_{k1})(\sum_l e_{ll}).\]
Comparing the EIN expressions literally we can find terms that are exactly the same (such as \(e_{ij}\) and \(e_{ij}\)). This approach is insufficient because it misses other common computations that are embedded inside the EIN operator. To be able to systematically find common computations we need to consider an EIN expression in context of the whole EIN operator. It is then we can distinguish between terms that just have shifted indices (such as \(e_{ij}\) and \(e_{jk}\)), those that create a subset of the same computations (such as \(\sum_l e_{ll}\) and \(e_{ij}\)), and those that are just not equivalent (such as \(e_{k1}\), and \(e_{0i}\)).

This section describes the techniques we developed to find common computations embedded inside a large EIN operator while maintaining the mathematical meaning of the computation. Split is used to split large EIN operators into a series of simple and small operators (Section 3.2.3). As we split we use Hash consing to find equivalent computations. Slice finds nonequivalent fields terms that create many of the same lower level base computations (Section 3.2.4).

There are other potential benefits to simplifying EIN operators with our approach. The best way to generate code from a large complicated EIN expressions can be obtuse. A general code generator would need to expand every operation to work on scalars, which could miss the opportunity for vectorization and lead to poor code generation. The implementation decomposes large and complicated EIN operators into many small and simple ones that are easier to analyze.

The following section describes our compilation techniques. First, to enable these techniques we simplify EIN terms with some rewriting (Section 3.2.1) and impose some formal sense of shape to handle indices correctly (Section 3.2.2). We then describe the implementation details for the split method (Section 3.2.3) and the slice method (Section 3.2.4). Lastly, we present some examples of the methods (Section 3.2.5).
3.2.1 Shift

The goal of the shift method is to move invariant terms outside a summation. Consider the term $\sum(T_\alpha \ast e)$ the invariant terms are moved outside the summation.

$$\sum_\nu (T_\alpha \ast e) \longrightarrow T_\alpha \ast \sum_\nu (e) \text{ where } \forall a \in \alpha. a \notin \nu$$

This simple rewrite can help simplify expressions before the next optimizations passes. The analysis can also turn an embedded summation into two independent loop nests and remove summation loops. We describe an overview of the method followed by an example.

**How it is implemented** In this section we sketch out the shift method. We do an analysis on the structure of the EIN body. Most of the method is made up of a series rewrites to move the summation operator.

The following code is written in SML. We break apart the code to mark the different steps. We use comments to remark on smaller details. We omit some code for conciseness but describe the details in the comments. We include EIN notation in brackets for clarity.

First we define a function filter that helps define invariant terms inside a summation. As input the filter function receives a list of EIN expressions $es$ and a single variable index $i$.

```sml
fun filter (es, i) = let
  (*A is a list of terms that are invariant to index i*)
  (*B is a list of terms that include index i*)
  (case e of E. Tensor(_, a) =>
    (*\forall a \in \alpha. if a \neq i
      * then add the term e to A
      * otherwise add the term to B
    *)
  )
  in (A, B) end
```

We do an analysis on the structure of the EIN body to move the summation operator.

```sml
fun rewrite (body) = (case body
  (* The summation is pushed past arithmetic operators.*)
```

of $E.\text{Sum}(sx, E.\text{Lift}(e)) \Rightarrow E.\text{Lift}(E.\text{Sum}(sx, e))$
| $E.\text{Sum}(sx, E.\text{Op}(rator, e))$
| $E.\text{Op}(rator, E.\text{Sum}(sx, \text{rewrite}(e)))$
| $(\{\sum(-e_1) \rightarrow -\sum(e_1), \sum(\sqrt{e_1}) \rightarrow \sqrt{\sum(e_1)}\} \ast)$
| (* The summation is distributed over binary operators.*)
| $E.\text{Sum}(sx, E.\text{Op2}(rator, e_1, e_2))$
| $E.\text{Op2}(rator, E.\text{Sum}(sx, \text{rewrite}(e_1)), E.\text{Sum}(sx, \text{rewrite}(e_2)))$
| $(\{\sum(e_1 + e_2) \rightarrow \sum(e_1) + \sum(e_2), \sum(e_1 - e_2) \rightarrow \sum(e_1) - \sum(e_2)\} \ast)$
| (*The scalars and invariant terms can be moved outside the product list.*)
| $E.\text{Sum}([i], E.\text{Op}(E.\text{Prod}, es))$
| $E.\text{Op}(E.\text{Prod}, A :: E.\text{Sum}([i], B))$

When there is an embedded summation (at least two variable indices bound at the summation operator) around a product operator then the method applies more analysis.

| $E.\text{Sum}([i, j], E.\text{Op}(E.\text{Prod}, es)) \Rightarrow ...$
| (* We filter es into sublist to reflect invariant terms. C is invariant to both, D is bound to i, E is bound to j, and F is otherwise.*)
| $\text{in } (\text{case filter}\(es, i)\)$
| $\text{of } ([], B) = \text{let}$
| $(\text{case filter}(B, j)\)$
| $\text{of } ([], F) \Rightarrow \{\sum_{ij} F\} \ast\text{original}\ast)$
| $(\text{E, } [i]) \Rightarrow \{\sum_i E\}$
| $(\text{E, F}) \Rightarrow \{\sum_j (E \sum_i F)\}$
| (*end case*)
| $(\text{A, B}) = (\text{case (filter}(A, j), \text{filter}(B, j))\)$
| $\text{of } ((C, [i]), (E, [i])) \Rightarrow \{C \sum_j E\}$
| $(\text{C, D}), (E, [i])) \Rightarrow \{(C \sum_i D)(\sum_j E)\}$
| $(\text{C, [i]), (E, F)) \Rightarrow \{C \sum_j (E \sum_i F)\}$
| $(\text{C, D}), (E, F)) \Rightarrow \{C \sum_i D \sum_j E \ast F\}$
| (*end case*)

The rewrites can move invariant terms and create independent loops. Simpler terms are easier to analyze and compare.
Example The left hand size of the identity \((s^* (v \otimes x)) \bullet u \rightarrow s^*(v \bullet xu)\) is represented as

\[ t = \lambda(s, v, x, u) \left( \sum_j (sv_i x_j u_j) \right) (s, v, x, u) \]

Unless, our compiler analyzed the bodies and identified the pattern, the best way to rewrite or implement the computations is obtuse.

\[ t = \lambda(s, v, x, u) \left( sv_i \sum_j (x_j u_j) \right) (s, v, x, u) \]

By moving the invariant expression \((\sum_j (x_j u_j))\) the dot product is recognizable and compiler has the option to generate vector code.

### 3.2.2 Tshape

To enable some of the compilation techniques we impose a concept of shape, referred to as \(tshape\), of an EIN expression. Consider the following EIN operator

\[ \left\langle \sum_{s x} e \ldots \right\rangle_{index} \]

Our goal is to find the indices included in the EIN expression \(e\) but bound outside of it \((index\ or\ sx)\). We find the \(tshape\) by doing a case analysis on the structure of \(e\).

Consider the following computation:

```plaintext
tensor [d] a;
tensor [d, d] b;
field #k(d) [] F = v \otimes h;
tensor [d, d] t_2 = a \otimes (\nabla \cdot F \cdot b);
```

After normalization the compiler represents the computation with the following EIN operator:

\[ t_2 = \lambda(V, H, A, B) \left\langle A_i \sum_k ((V \otimes (\nabla_k H)) B_{kji}) \right\rangle (v, h, a, b) \]
If we consider the subterm \( e_2 \):

\[
e_2 = \sum_k ((V \otimes (\nabla_k H))B_{kj})
\]

(3.7)

the \( tshape \) of \( e_2 \) is variable index \( j \). Variable index \( k \) is not included because it is bound inside the sub-term. This section provides a sketch of the \( tshape \) function and examples.

**How it is implemented** The \( tshape \) includes the list of variable indices that are used in the EIN expression, but are bound outside of it. We provide a sketch of the \( tshape \) function that analyzes this structure.

The following code is written in SML with slight modifications. The code is broken apart to mark the different steps. The comments describe the smaller details. We add the EIN IR notation in brackets for clarity.

As input the function takes the EIN operator’s shape (index) and summation indices (sx) and returns a list of indices for the \( tshape \) of an EIN expression.

```sml
fun get_tShape (index, sx, e) = let
  val outerAlpha = let
    fun add ([], _, s) = ISet.addList(s, List.map(fn (v, _, _) => v) sx)
    | add (r::r, i, s) = add (r, i+1, ISet.add(s, i))
  in
    add (index, 0, ISet.empty)
  end

We find the set of indices that are bound outside the EIN expression.

(* outerAlpha = set of indices supported by original EIN *)
val outerAlpha = let
  fun add ([], _, s) = ISet.addList(s, List.map(fn (v, _, _) => v) sx)
  | add (r::r, i, s) = add (r, i+1, ISet.add(s, i))
  in
    add (index, 0, ISet.empty)
  end

By doing a case analysis of the body get_eShape() returns the indices mentioned in the body.

(* get the indices in EIN expression*)
fun get_eShape (e, ixs) = (case e
  (*Constants \{c\} are scalar and so the shape is empty.*)
  of E.Constants => []
  (*Add the indices in the base cases. \{T_\alpha, F_\alpha, \delta_\alpha, \epsilon_\alpha\}*）
  | E.Tensor(_, alpha) => alpha@ixs
```

47
The next step connects the above functions. First we find the indices used in the EIN body and then remove the constant indices. The remaining indices are only those bound outside the EIN body (∈ outerAlpha).

(* Filter indices.
* Do not include constant indices
* or those bound by an inner summation *)

fun getT ([], rest) = List.rev rest
  | getT ((E.C _)::es, rest) = getT(es, rest)
The result is the list of variable indices bound outside the subexpression \( e \) or the \( t\text{shape}(e) \).

**Tshape Examples** The following provides examples of extracting the tshape from EIN expressions. Unlike tradition index notation, two repeat indices do not imply summation. The shape extracted needs to reflect simple operations A+B.

\[
e_1 = A_{ijk} + B_{ijk} \text{ and } \beta = i, j, k
\]

when sub-terms do not have the same indices \( F \otimes G \),

\[
e_1 = F_i G_j \text{ and } \beta = i, j
\]

the order that the indices appear Transpose(A\( \otimes \)B),

\[
e_1 = A_j B_i \text{ and } \beta = j, i
\]

if the indices repeat in the same term Trace(\( V \otimes H \)),

\[
e_1 = \sum_i V_{ii} \otimes H \text{ and } \beta = []
\]

if the indices repeat in multiple terms modulate(A,B),

\[
e_1 = A_{ij} B_{ij} \text{ and } \beta = i, j
\]

remove indices bound by a summation as in A : B

\[
e_1 = \sum_{ij} A_{ij} B_{ij} = []
\]

or embedded summation \( (V \otimes H) \cdot M \),

\[
e_1 = \sum_j (V \otimes \frac{\partial}{\partial x_j} H) M_{ji} \text{ and } \beta = i
\]

and with summation operators A \( \otimes \)(F \( \cdot \) C),

\[
e_1 = A_i (\sum_k F_k C_{kj}) \text{ and } \beta = i, j.
\]
3.2.3 Split

An EIN operator can produce many of the same computations and we apply the split method to enable CSE (common subexpression elimination). Comparing the EIN expressions in the EIN operator alone is insufficient because common terms may have different indices and comparing the indices directly can provide a false inequality (see introductory prose Section 3.2). To be able to find common computations we need to consider an EIN expression in context of the whole EIN operator. Split enables this comparison by creating new EIN operators to represent computations inside the original EIN operator. It is then possible to compare the new EIN operators to existing ones and enable CSE.

The split method is used to resolve a large EIN operator by creating smaller and simpler ones. The implementation uses $tshape$ (introduced in Section 3.2.2) to analyze EIN expressions. In this section we describe the implementation details followed by an example.

How it is implemented An embedded or inner EIN expression is lifted out to create a new EIN operator and is replaced with a tensor term. The split method does the following decomposition:

$$t_2 = \lambda A, B(e_1 e)_{\sigma}(a, b) \rightarrow t_1 = \lambda B(e_1')_{\hat{\alpha}}(b)$$
$$t_2 = \lambda A, T(T_\alpha e)_{\sigma}(a, t_1) \text{ where } tshape(e_1) = \alpha$$

The implementation details are described in more detail next using this example.

The following code is written in SML. We break the code apart to mark the different steps. We omit several function definitions for conciseness but describe their application in the comments. We also use comments to describe the smaller details. In the code we refer to the terms in the ongoing example.

As input the split operator receives a variable, an EIN operator and arguments and its returns or inserts bindings of EIN operators to variables (illustrated here in comments). The input has the form: $t2 = \text{einopp (args)}$. Using the ongoing example $\text{einopp} = \lambda A, B(e_1 e)_{\sigma}$
and args=[a,b].

def split(t2, einapp, args) = let

We scan the EIN body \((e_1e)\) to find an embedded EIN subexpression \((e_1)\).

def is_Split(sx, body)= (case body
  of E.Tensor _ => (* does not split*)
  | E.Opn(E.Prod, e_1::e_2) =
    if(is.OP (e_1))
      then (*splits*)
        split_core(e_1, sx)
      else (*does not split*)
        | E.Sum(sx', body) = is_Split(sx@sx', body)
        | ...
    (* end case *)

Call function split_core() when we find a term \(e_1\) to lift out.

def split_core(e_1, sx) = let
  (* \(e_1\) is the subexpression that is lifted out
  * \(sx\) is a list of indices bound by a summation operator
  *)
  (* We get the tshape of \(e_1\).*
  val α = tshape(index, sx, e_1)
  (*The outshape \(\{\hat{α}\}\) is the binding to tshape(\(e_1\))*)
  val index = einapp.index
  val outshape =
    List.map (fn (E.V e) => List.nth(e, index)) α

A new operator is created for the EIN expression \(e_1\).

(*create a map for variable indices*)
val (mapp_ix) = scan_ix e_1
(*filter list of parameters and arguments*)
val (mapp_id, B, b) = scan_id e_1
(*Remap the body’s parameter ids and variable indices
  * to only those included in the new EIN operator.
  *)
val e' = reset(mapp_ix, mapp_id, e)
(*Create the new EIN operator with arguments*)
val newEinApp = {λB.⟨e'⟩_{\hat{α}}(b)}

A complete representation of the computation is needed before comparing it to other com-
computations. Creating an EIN operator around the subexpression provides the missing details.

The method can then check the hash table to see if the computation has been derived previously.

\[
\text{val } t_1 = \text{checkTbl (newEinApp)}
\]

\[
\text{(* Bind variable to EIN operator \{t_1 = newEinApp\}*)}
\]

The original \textit{EIN} operator (einopp which is bound to variable \textit{t}_2) is rewritten.

\[
\text{val einexp} = \{\langle T_\alpha e \rangle_{\sigma}\}
\]

\[
\text{val einopp} = \{\lambda A,T \text{ einexp}\}
\]

\[
\text{val args} = [a, t_1]
\]

\[
\text{(* Bind variable to EIN operator \{t_2 = (einopp) args\}*)}
\]

As a result the subexpression is lifted out of the original \textit{EIN} operator.

**Splitting example** Continuing the example in the previous section (3.6). During the split method the sub-term \textit{e}_2 (3.7) is lifted out. A new EIN operator is created by using the sub-term \textit{e}_2 as a body. The variable indices and parameters are remapped accordingly.

\[
t_1 = \lambda (V,H,B) \left( \sum_j ((V \odot (\nabla_j H)) B_{ji}) \right) (v, h, b)
\]

In the original operator the sub-term \textit{e}_2 is replaced with a tensor term \(T_{c\text{shape}(e2)}\).

\[
t_2' = \lambda (A,T) \langle A_i T_j \rangle_{ij}^i (a,t_1)
\]

**Substitution and Split** In a way, the Split method is the inverse of the Substitution method (Section 3.1.1). Substitution composes several EIN operators into a single one. Split transforms a single EIN operator into several pieces. Both methods require some parameter clean-up and index-analysis to maintain the integrity of the computation.
3.2.4 Slice

Our goal is to reduce the number of fields terms that get transformed into lower level constructs. The slice method addresses this goal by targeting sliced field terms (terms that are not equal) and creating a common EIN operator that can do the bulk of the computation just once. In the following text we define sliced field, describe how the slice method is implemented and present examples.

**Sliced fields** We use the phrase “sliced tensor” or “sliced field” to indicate an EIN term that has at least one constant index (Section 2.1). As a reminder, a constant index refers to a specific component of a tensor or tensor field. For instance, a matrix $M_{ij}$ can be sliced to create a vector $M_{i1}$, where $M_{i1} = [M_{01}, M_{11}, M_{21}]$. The tensor components are distinct (i.e. $M_{01} \neq M_{11}$) and independent of each other. Field components are also distinct but they depend on the same source data. Different sliced fields can generate many of the same lower level constructs (base computations between image data and kernels). Translating each sliced field term can create a large IR.

**How it is implemented** We identify sliced field terms in the body of an EIN operator. Sliced field terms are probed fields that use a constant index $c$.

$$g = \lambda F \langle \nabla_\beta F_{c\alpha}(x) \rangle_{\hat{\alpha}\hat{\beta}}(f)$$

We create a new EIN operator $t$ to represent unsliced versions of the field term. The body of the unsliced field $\nabla_\beta F_{c\alpha}(x)$ is changed by converting the constant index to a variable index and it is cleaned up by reinitializing the variable index bindings.

$$t = \lambda F \langle \nabla_\nu F_\mu(x) \rangle_{\hat{\mu}\hat{\nu}}(f)$$
We check if the un-sliced field has been used before creating a new variable $t$. If so, then we save space from creating many unused operators.

The slice operation is pushed to a tensor that samples the un-sliced field $t$.

$$g = \lambda T \langle T_{c \alpha \beta} \rangle \hat{\alpha} \hat{\beta}(t)$$

The tensor term $T_{c \alpha \beta}$ indicates how the field $(t)$ is sampled.

**Example** Consider the determinant of a second-order tensor field $M$

```plaintext
tensor[3,3]T = M(p);
tensor[]G = det(M)(p);
```

transformed to EIN as

$$T = \lambda(F, T) \langle F_{ij}(T) \rangle_{ij}^{; ;} \quad (M, p)$$

$$G = \lambda(F, T) \left\langle \sum_{ijk} \varepsilon_{ijk} F_{0i}(T) F_{1j}(T) F_{2k}(T) \right\rangle_{ij}^{; ;} \quad (M, p) \quad (3.8)$$

A naive implementation splits the EIN operator $G$ and creates unique terms $F_{ij}, F_{0i}, F_{1i}, F_{2i}$.

```plaintext
T = \lambda(F, T) \langle F_{ij}(T) \rangle_{ij}^{; ;} \quad (M, p)
t_0 = \lambda(F, T) \langle F_{0i}(T) \rangle_i^{; ;} \quad (M, p)
t_1 = \lambda(F, T) \langle F_{1i}(T) \rangle_i^{; ;} \quad (M, p)
t_2 = \lambda(F, T) \langle F_{2i}(T) \rangle_i^{; ;} \quad (M, p)
G \longrightarrow \lambda(A, B, C) \left\langle \sum_{ijk} \varepsilon_{ijk} A_i B_j C_k \right\rangle \quad (t_0, t_1, t_2)
```

In the next phase of the compiler each field term $(F_{ij}, F_{0i}, F_{1i}, F_{2i})$ is transformed to lower level constructs and create many common computations. Instead we use slice to transform a sliced field term into a tensor operator indexing the unsliced field. For simplicity, we express
the result with some rewriting.

\[ T = \lambda(F, T) \left\langle F_{ij}(T) \right\rangle^{ij} \quad (M, p) \]
\[ G \rightarrow \lambda(T) \left\langle \sum_{ijk} E_{ijk} T_{ij} T_{jk} \right\rangle^{ij} (T) \]

There is only one field term \( F_{ij} \) instead of four. Consider another example, if the program computed the Hessian of the determinant

\textbf{tensor} [3,3] \( G = \nabla \otimes \nabla \det(M) \);

then the additional EIN operators are created:

\[
\begin{align*}
  g_0 &= \lambda P \left\langle \nabla_j F_{0i}(T) \right\rangle^{ij} (M, p) \\
  h_0 &= \lambda P \left\langle \nabla_j F_{0i}(T) \right\rangle^{ijk} (M, p) \\
  g_1 &= \lambda P \left\langle \nabla_j F_{1i}(T) \right\rangle^{ij} (M, p) \\
  h_1 &= \lambda P \left\langle \nabla_j F_{1i}(T) \right\rangle^{ijk} (M, p) \\
  g_2 &= \lambda P \left\langle \nabla_j F_{2i}(T) \right\rangle^{ij} (M, p) \\
  h_2 &= \lambda P \left\langle \nabla_j F_{2i}(T) \right\rangle^{ijk} (M, p)
\end{align*}
\]

Our method reduces the 10 field terms down to 3 unique field terms. The difference can be more significant when considering more complicated tensor computations. This method creates a smaller representation of the computation while maintaining mathematical meaning.

\textbf{Slice and Split comparison} The goal of split is to find the computations (that maybe written differently) embedded in an EIN expression. The goal of slice is to find a specific type of computation (that are not equal) and creates a general EIN operator (that can be sampled). In other words, in effect split reduces expressions that are exactly the same and slice reduces terms that are not the same but create many of the same basic operators later on. The split method is beneficial to any program that creates redundant terms. Slice is only helpful to programs that create sliced tensor fields in an EIN body (such as the determinant operator).
3.2.5 Examples

In the following section we provide some applications of the methods used in this chapter. The first example illustrates the implementation steps and by chance it reduces a tensor computation and finds a tensor identity. The second example applies the compilation methods to a computation created at a later stage of the compiler. The third example illustrates an index-based simplification enabled by substitution.

Tensor Identity The following is an example of a tensor term being rewritten using our methods and revealing a tensor identity. Consider \((A \times B) \cdot (C \times D)\) represented with a single EIN operator as

\[
t = \lambda(a, b, c, d) \langle e \rangle (A, B, C, D)
\]

\[
e = \sum_i ((\sum_{jk} E_{ijk} a_j b_k) (\sum_{lm} E_{ilm} c_l d_m))
\]

The body is normalized in the rewriting system with rewrites Equation 3.2 and Equation 3.3 and unused indices are removed.

\[
e = \sum_i ((\sum_{jk} E_{ijk} a_j b_k) (\sum_{lm} E_{ilm} c_l d_m)) \rightarrow \sum_{jk} ((a_j b_k c_j d_k) - (a_j b_k c_k d_j))
\]

The next phase of the compiler would not be sure how to generate optimal code for these computations. Shift moves the invariant inside loop outside of the outer loop.

\[
e \rightarrow (\sum_j (a_j c_j) \sum_k (b_k d_k)) - (\sum_j (a_j d_j) \sum_k (b_k c_k))
\]

The split algorithm would create four EIN operators of the form \(\sum_i A_i B_i\) that which makes it easier to find common expressions in the rest of the program and take advantage of vector hardware. As a result we find tensor identity:

\[
(A \times B) \cdot (C \times D) \rightarrow (A \cdot C) * (B \cdot D) - (A \cdot D) * (B \cdot C).
\]
**Post field reconstruction**  EIN notation is used to represent tensor operators used for field reconstruction (Section 2.3). The compiler can concisely represent the diverse set of field types that are differentiated. Computations between tensors and reconstructed fields were opaque in the direct-style compiler. By writing operations in EIN notation we can use the optimizations strategies previously discussed.

Consider the transformation \( \text{out} = P \cdot t_0 \cdot P \) and \( G \) from Equation 2.6 and Equation 2.4, respectively. Function \( G \) initializes the variable indices in the field component \( t_0 \). The following defines three ways to represent this computation. The first approach is by representing the entire computation in one operator

\[
\text{out} = \lambda(v, h, n, f, P) \left( \sum_{kl=0}^{1} G(k, l, m, n) P_{ik} P_{jl} \right) (\text{img, bspln3, n, f, P}) \quad (3.10)
\]

The second approach applies \( \text{split} \) (Section 3.2.3) to create two EIN operators.

\[
t_0 = \lambda(v, h, n, f) \langle G(i, j, k, l) \rangle_{ij} \\
\text{out} = \lambda(T, P) \langle e_2 \rangle_{ij} (t_0, P) \quad \text{where} \quad e_2 = \sum_{kl=0}^{2} T_{kl} P_{ik} P_{jl} 
\quad (3.11)
\]

The third approach applies shift and split together. \( \text{shift} \) rewrites \( e_2 \)

\[
e_2 = \sum_{kl=0}^{2} T_{kl} P_{ik} P_{jl} \rightarrow \sum_{k=0}^{2} P_{ik} \sum_{l=0}^{2} T_{kl} P_{jl}
\]

and then \( \text{split} \) creates new operators.

\[
t_1 = \lambda(T, P) \left( \sum_{k=0}^{1} T_{ik} P_{jk} \right) (t_0, P) \quad \text{out} = \lambda(T, P) \left( \sum_{k=0}^{1} P_{ik} T_{kj} \right) (t_1, P) \quad (3.12)
\]

We revisit these computations in Section 3.3.3.
Index-based optimization  The simple act of substitution can take advantage of index-base optimizations. The trace of the Hessian $Tr(\nabla \otimes \nabla \varphi)$ is mapped to two EIN expressions in High-IR as

$$
t_1 = \lambda(F) \left\langle \frac{\partial}{\partial x_{jk}} F \right\rangle_{jk} (\varphi)
$$

$$
t_2 = \lambda(T) \left\langle \sum_i T_{ii} \right\rangle (t_1)
$$

The outer indices for $t_1 (j, k)$ are mapped to indices in term $T_{ii}$.

$$
t_2 \rightarrow \lambda(F) \left\langle \sum_i \frac{\partial}{\partial x_{ii}} F \right\rangle (\varphi) \quad \text{where } j \text{ is mapped to } i \text{ and } k \text{ is mapped to } i
$$

When these operations are applied, the result will be the Laplacian (typically noted in math textbooks with $\Delta$).

3.3 Benchmarks

Addressing the technical and compilation challenges in implementation the EIN IR has enabled a richer language that can be used to support richer visualization programs (Section 7.2). It is useful to evaluate our implementation on the compiler and the language. In this section, we present three sets of benchmark results. The first set is an evaluation of implementing EIN over the original direct-style compiler. The second set of numbers measures the effect of the techniques described in Section 3.2. The third is an evaluation of the effect of using the higher-order features of the language versus equivalent first-order implementations.

3.3.1 Experimental Framework

The benchmarks were run on an Apple iMac with a 2.7 GHz Intel core i5 processor, 8GB memory, and OS X Yosemite (10.10.5) operating system. Each benchmark was run 10 times and we report the average time in seconds.

The benchmarks are presented in the figures in order of mathematical complexity. Some
of visualization concepts that inspired the benchmarks are described in Section 7.2. Bench-
marks “illust-vr”, “lic2d”, “Mandelbrot”, “ridge3d”, and “vr-lite-cam” are small examples
available in the original compiler [16]. The benchmarks “mode”, “canny”, and “moe” are
used to create figures in [41]. “Mode” finds lines of degeneracy in a stress tensor field revealed
by volume rendering isosurface of tensor mode; “Canny-edges” computes Canny Edges; and
“Moe” volume renders isocontours found using Canny Edges [10].

The benchmarks “dec-crest”, “dec-grad”, “rsvr”, and “mode-rig’ were not featured in
previous work, because they were outside the scope of possibility and involved a higher degree
doing tensor math. Programs “dec-crest” and “dec-grad” are approximations to illustrate the
crest lines on a dodecahedron. Programs “mode-rig” and “rsvr” are both programs created to
measure ridge lines. The micro-benchmarks “det-grad”, “det-hess”, and “det-trig” compute
a single property: The gradient, hessian, and various functions computed on the determinant
doing a field. Their run times are negligible and are omitted.

3.3.2 The Effect of implementing EIN

Our experiments run 18 benchmarks on three versions of the compiler. The first is the original
compiler that does not include EIN notation. The second and third include the design
and implementation techniques included in this chapter, but the second version imposes
restrictions that are meant to reflect a more naive implementation of the design. We measure
the time it takes for the programs to compile. For the programs that compile on at least two
settings, we measure the run times. We use an X symbol to indicate a program has failed
to compile.

Before EIN, the Diderot compiler was restrictive and could not handle programs that use
a high level math. This version of the compiler is considered the “Original compiler”. It
did not have the feature support required to compile some of the programs included in the
benchmark. Figure 3.1 compares the application of EIN with the original compiler.

The development of EIN has impacted the type and complexity of programs that we can
Figure 3.1: The “Original” version of the compiler does not use the EIN IR. “Ein with restrictions” is the more naive implementation of EIN. “EIN” is the baseline with the EIN IR with full optimizations applied. Fully implementing EIN allows more programs to compile than previously possible.
write in Diderot. Figure 3.1 compares the application of EIN with full or restricted level of optimizations. The next section will take a closer look at the impact of individual techniques. Here we can summarize that fully implementing the optimizations enables Diderot to compile programs that otherwise cannot compile.

3.3.3 The Effect of Compiler Settings

As we have discussed previously, a naïve application of our transformations causes unacceptable space blowup. To address the space problem we developed techniques to reduce the size of the IR resulting from lowering passes. While their implementation might allow more complicated Diderot programs to be created we want to evaluate the cost or benefit it might impose on the programs that could already compile. In the following we evaluate the effectiveness of these techniques together and isolated at different levels of abstraction.

Application to higher order constructs

Techniques Split and Slice are effective at reducing the size of the program by finding common subexpressions or reducing field terms. Figure 3.2 measures the effectiveness of applying Split (Section 3.2.3) and Slice (Section 3.2.4) on a high-IR EIN operator. The Slice technique is necessary to compile 3 of the 13 benchmarks. Split is the most consequential technique. Restricting it stops 5 of the programs from being able to compile. Neither technique assert a considerable cost to execution time.

Size Reduction

To take a closer look at how the optimizations are impacting the size of programs, we measure the size of a Diderot program at 6 different phases of the compiler under 5 different settings. Each setting has a set of optimizations turned on, off, or restricted. Missing data, or a line that terminates before phase 6 indicates that the program “stopped” during compilation.
Figure 3.2: Compile and run time measurements when implementing *Slice* and *Split* on High-IR. Doing no amount of splitting prevents most of these programs from compiling so instead we measure its impact by limiting it, “Minimal *Split*”. EIN is the baseline with techniques *Split* and *Slice* implemented.
Figure 3.3: The graphs shows the size of the dec-grad (left) and rsvr program (right) program at different phases in the compiler. The lines means the program could not compile past that phase. “EIN” is the baseline with full optimizations.

and could not continue under said settings. Figure 3.3 illustrates the size of two programs at different phases of the compiler.

Some programs depend on a combination of the optimizations to compile. The program “rsvr” stops as early as phase “4” without Slice. Unless all the optimizations are implemented program “rsvr” cannot compile. Program “dec-grad” is less restrictive. It can compile without Shift and Slice but it stops when Split is not fully applied.

Application to low order constructs

We measured the effect of optimizations at a lower level of abstraction. Section 3.2.5 demonstrates the transformation of differentiation indices in a reconstructed field term. We can write this computation in three different ways: in place Equation 3.10, by applying Split Equation 3.11, or by applying Shift and Split together Equation 3.12. Figure 3.4 measures the impact of these optimizations on this computation.

The implementation of the techniques directly impact compile time. Applying optimizations Shift and Split together offers a consistent speed-up on the execution time and compile time for all 13 benchmarks. 5 programs experienced at least a 20x compile time speed-up. 4 of the 7 benchmarks offered at least a 1.3 speed up on execution time while the rest were comparable.
Figure 3.4: Shift and Split on Mid-IR. Compile and run time measurements for implementing Shift and Split techniques on reconstructed field terms. EIN baseline includes the application of Shift and Split.
Figure 3.5: This figure compares hand-derived first-order programs with their high-order equivalent. We can only compile first-order programs on the original compiler. We can compile the first-order and higher-order programs with EIN.
3.3.4 First-Order versus Higher-Order

High-order versions of program are the preferred way to write Diderot code. Their first-order counterparts require more lines of code and makes the user do derivations by hand. The process can be time-consuming, tedious, and error-prone. For this experiment we choose three programs with a high level of math. We wrote first-order versions of them that could be expressed on the original compiler. Figure 3.5 reports the compile and run time for first-order programs with their higher order counterparts.

The experiment actually test two things; (1) the original compiler versus the EIN compiler and (2) comparing the first-order and higher-order versions of a program ran on the same compiler. The first order programs compiled and ran faster on EIN than on the original compiler. The first-order programs were sometimes faster than their higher-order counterparts but comparable. Ultimately, it is better to skip the hassle of writing first-order versions of programs and use the higher-order versions on the EIN compiler.
CHAPTER 4
PROPERTIES OF NORMALIZATION

The Normalization transformation (Section 3.1.2) plays a key rôle in the compilation of Diderot programs. The transformations are complicated and it would be easy for a bug to go undetected. To increase our confidence in normalization part of the compiler we provide the following analysis. We have defined a type system for EIN operators in Section 4.1 and value judgements in Section 4.2. We aim to prove that normalization preserves both types and the meaning of EIN operators. We have also proven that the rewrite system is terminating in Section 4.3. We include full proofs in the appendix.

4.1 Type Preservation

4.1.1 Typing EIN Operators

At the level of the SSA representation, we have types $\theta \in \text{TYPE}$ that correspond to the surface-level types:

$$
\theta ::= \text{Ten}[d_1, \ldots, d_n] \quad \text{tensors} \\
| \quad \text{Fld}(d)[d_1, \ldots, d_n] \quad \text{fields} \\
| \quad \text{Img}(d)[d_1, \ldots, d_n] \quad \text{images} \\
| \quad \text{Krn} \quad \text{kernels}
$$

An EIN operator $\lambda \bar{x}(e)_{\sigma}$ can then be given a function type $(\theta_1 \times \cdots \times \theta_n) \rightarrow \theta$, where $\theta$ is either $\text{Ten}[d_1, \ldots, d_n]$ or $\text{Fld}(d)[d_1, \ldots, d_n]$ and $\sigma$ is $i_1 : d_1, \ldots, i_n : d_n$. The EIN expression that is body of the operator $(e)$, however, cannot be given a type $\theta$, since it represents a computation indexed by $\sigma$. Thus the type system for EIN expressions must track the index space as part of the context. We then define the syntax of indexed EIN expression types as

$$
\tau ::= (\sigma)\mathcal{T} \mid (\sigma)\mathcal{F}_d
$$

67
where \((\sigma)T\) is the type of indexed tensors and \((\sigma)F_d\) is the type of indexed \(d\)-dimensional fields. We define our typing contexts as \(\Gamma_\sigma \in (\mathbb{Z} \times \text{INDEXVAR} \times \mathbb{Z})^* \times (\text{VAR} \rightarrow \tau)\). The typing context \(\Gamma_\sigma\) includes both the index map and an assignment of types to non-index variables. We present a few typing rules next and refer the reader to Figure 4.1.1 for a complete list of the rules.

**Index Space** \(\sigma\)  
Recall that \(\sigma \in (\mathbb{Z} \times \text{INDEXVAR} \times \mathbb{Z})^*\) with the restriction that the index variables are unique (Section 2.1). It is helpful to view \(\sigma\) as defining a finite map from index variables to the size of their range (i.e., an index variable \(i\) ranges from 0 to \(\sigma(i)\)). As a shorthand, we use “\(\sigma(i) = n\)” to indicate the upper bound of variable index \(i\). To indicate the addition or lack of a binding we use “\(\sigma = \sigma', i : n\)” and “\(\sigma = \sigma'/ i\)”, respectively.

Recall that an EIN index \(\mu\) is either constants (\(\mu \in \mathbb{N}\)) or variables indices \(\mu \in \sigma\). We create an IndexRule function to shorten the judgment statements.

\[
\text{IndexRule}_\alpha = \forall \mu_i \in \alpha, \text{ either } \mu_i \in \mathbb{N} \text{ and } 1 \leq \mu_i \leq d_i, \text{ or } \mu_i \in \text{dom}(\sigma) \text{ and } \sigma(\mu_i) = d_i
\]

**Base case**  
First consider the base case of a tensor variable \(T_\alpha\); the typing rule is

\[
\begin{align*}
\text{Ty}_{\text{Judge} 1) } & \quad \Gamma_\sigma(T_\alpha) = \text{Ten}[d_1, \ldots, d_n] \quad |\alpha| = n \quad \text{IndexRule}(\alpha) \\
\frac{}{\Gamma_\sigma \vdash T_\alpha : (\sigma)T}
\end{align*}
\]

The antecedents of this rule state that \(T_\alpha\) has a type that is compatible with both the multi-index \(\alpha\) and the index map \(\sigma\). A similar rule applies for field variables. The rule for convolution yields an indexed field type.

\[
\begin{align*}
\text{Ty}_{\text{Judge} 2) } & \quad \Gamma_\sigma(V_\alpha) = \text{Img}(d)[d_1, \ldots, d_n] \quad \Gamma_\sigma(H^\beta) = \text{Krn} \\
\frac{}{|\alpha| + |\beta| = n \quad \text{IndexRule}(\alpha\beta)}{\Gamma_\sigma \vdash V_\alpha \circledast H^\beta : (\sigma)F_d}
\end{align*}
\]
Add index to environment  Note that the index space covers both the shape of the image’s range and the differentiation indices. Consider the following typing judgement for the EIN summation form:

\[
\text{Ty}_{\text{Judge} \ 3)} \quad i \notin \text{dom}(\sigma) \quad \Gamma_{\sigma,i:n} \vdash e : (\sigma, i : n)\mathcal{T} \\
\Gamma_{\sigma} \vdash \sum_{i=1}^{n} e : (\sigma)\mathcal{T}
\]

Here we extend the index map with \(i : n\) when checking the body of the summation \(e\). This rule reflects the fact that summation contracts the expression. A similar process occurs for differentiation.

\[
\text{Ty}_{\text{Judge} \ 4)} \quad \sigma(i) = d \quad \sigma' = \sigma \setminus d \quad 1 \leq i \leq d \quad \Gamma_{\sigma'} \vdash e : (\sigma')\mathcal{F}_d \\
\Gamma_{\sigma} \vdash \frac{\partial}{\partial x_i} \circ e : (\sigma)\mathcal{F}_d
\]

Permutation  The term \(\delta_{ij}\) by itself does not change the environment.

\[
\text{Ty}_{\text{Judge} \ 5)} \quad i,j \in \text{dom}(\sigma) \\
\Gamma_{\sigma} \vdash \delta_{ij} : (\sigma)\mathcal{T} \\
\Gamma_{\sigma} \vdash \delta.\delta : (\sigma)\mathcal{T}
\]

The application of a kronecker delta function \(\delta_{ij}\) adds index \(j\) to the environment and removes index \(i\).

\[
\text{Ty}_{\text{Judge} \ 5)} \quad \Gamma_{\sigma'} \vdash e : (\sigma')\mathcal{T} \quad \sigma' = j,\sigma/ i \\
\Gamma_{\sigma} \vdash (\delta_{ij} \circ e) : (\sigma)\mathcal{T} \\
\Gamma_{\sigma} \vdash \delta_{ij} \circ e : (\sigma)\mathcal{F}_d
\]

\[
\text{Ty}_{\text{Judge} \ 5)} \quad \Gamma_{\sigma'} \vdash e : (\sigma')\mathcal{F}_d \quad \sigma' = j,\sigma/ i \\
\Gamma_{\sigma} \vdash (\delta_{ij} \circ e) : (\sigma)\mathcal{F}_d
\]

Similarly, the \(\epsilon\) term by itself does not change the environment.

\[
\text{Ty}_{\text{Judge} \ 6)} \quad \alpha \in \text{dom}(\sigma) \\
\Gamma_{\sigma} \vdash \epsilon_{\alpha} : (\sigma)\mathcal{T} \\
\Gamma_{\sigma} \vdash \epsilon_{ijkl}\epsilon_{ilm} : (\sigma)\mathcal{T}
\]
When applying $\epsilon$ to another term we reflect that term’s type.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\mathcal{E}_\alpha \ast e) : \tau}$$

**Operators** The Probe operation probes a field type $(\sigma)\mathcal{F}_d$ and a tensor Ten[d], unless the term is a permutation term.

$$\frac{\Gamma \vdash \delta_{ij} : \tau}{\Gamma \vdash \delta_{ij}@x : \tau} \quad \frac{\Gamma \vdash \epsilon_\alpha : \tau}{\Gamma \vdash \epsilon_\alpha@x : \tau} \quad \frac{\Gamma \vdash e : (\sigma)\mathcal{F}_d \quad \Gamma \vdash x : \text{Ten}[d]}{\Gamma \vdash e@x : (\sigma)\mathcal{T}}$$

Consider lifting a tensor term to the field level:

$$\frac{\Gamma \vdash e : (\sigma)\mathcal{T}}{\Gamma \vdash \text{lift}(e) : (\sigma)\mathcal{F}_d}$$

The sub-term $e$ is a tensor type $(\sigma)\mathcal{T}$ but the lifted term \text{lift}(e) is a field type $(\sigma)\mathcal{F}_d$. The rest of the judgements are quite straightforward. Some unary operators \{\sqrt{}, -, \kappa, \exp, (\cdot)^n\} can only be applied to scalar values terms such as reals and scalar fields.

$$\frac{\Gamma \vdash e : ()\mathcal{T} \quad \odot_1 \in \{\sqrt{}, -, \kappa, \exp, (\cdot)^n\}}{\Gamma \vdash \odot_1(e) : ()\mathcal{T}} \quad \frac{\Gamma \vdash e : ()\mathcal{F}_d \quad \odot_1 \in \{\sqrt{}, -, \kappa, \exp, (\cdot)^n\}}{\Gamma \vdash \odot_1(e) : ()\mathcal{F}_d}$$

The subexpressions in a subtraction operation have the same type as the result.

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \odot_2 \in \{+, -\}}{\Gamma \vdash (e_1 \odot_2 e_2) : \tau}$$
The full set of typing judgements and corresponding inversion lemmas are contained in Figure 4.1.1 and Figure 4.1.1, respectively.

\[
\sigma = i_1 : d_1, \ldots, i_m : d_m(\sigma, \{x_i \mapsto \theta_i \mid 1 \leq i \leq n\}) \vdash e : (\sigma)\mathcal{T}
\]

\[
\vdash \lambda (x_1 : \theta_1, \ldots, x_n : \theta_1)(e)_{\sigma} : (\theta_1 \times \cdots \times \theta_n) \rightarrow \text{Ten}[d_1, \ldots, d_m]
\]

### 4.1.2 Type preservation Theorem

Given the type system for EIN expressions presented above, we have shown that types are preserved by normalization.

**Theorem 4.1.1** (Type preservation). If \( \vdash \Gamma_{\sigma} \text{ ok}, \Gamma_{\sigma} \vdash e : \tau \), and \( e \rightarrow e' \), then \( \Gamma_{\sigma} \vdash e' : \tau \)

**Proof.** Given a derivation \( d \) of the form \( e \rightarrow e' \) we state \( T(d) \) as a shorthand for the claim that the derivation preserves the type of the expression \( e \). For each rule, the structure of the left-hand-side term determines the last typing rule(s) that apply in the derivation of \( \Gamma_{\sigma} \vdash e : \tau \). We then apply a standard inversion lemma and derive the type of the right-hand-side of the rewrite.

Provided below are key cases of the proof (Section A.1).

**Rule adds an index to the environment:**

The rewrite rule \(( R4)\) has the form \( (\sum_{i=1}^{i=n} e_1)@x \rightarrow (\sum_{i=1}^{i=n} (e_1)@x)\).

Find \( \Gamma_{\sigma} \vdash ((\sum_{i=1}^{i=n} (e_1))@x) \)

This type of structure inside a probe operation results in a tensor type.

\((\sum_{i=1}^{i=n} e_1)@x\)

Find \( \Gamma_{\sigma} \vdash (e_1) \)
\[
\begin{align*}
\text{TyJudge 1)} & \quad \Gamma_{\sigma}(T_{\alpha}) = \text{Ten}[d_1, \ldots, d_n] \quad |\alpha| = n \quad \text{IndexRule}(\alpha) \\
\Gamma_{\sigma} & \vdash T_{\alpha} : (\sigma)T \\
\Gamma_{\sigma}(F_{\alpha}) = \text{Fld}(d)[d_1, \ldots, d_n] \quad |\alpha| = n \quad \text{IndexRule}(\alpha) \\
\Gamma_{\sigma} & \vdash F_{\alpha} : (\sigma)F_d \\
\text{TyJudge 2)} & \quad \Gamma_{\sigma}(V_{\alpha}) = \text{Img}(d)[d_1, \ldots, d_n] \\
\Gamma_{\sigma}(H^{\beta}) = \text{Krn}[\alpha] + |\beta| = n \quad \text{IndexRule}(\alpha \beta) \\
\Gamma_{\sigma} & \vdash V_{\alpha} \otimes H^{\beta} : (\sigma)F_d \\
i & \notin \text{dom}(\sigma) \\
\Gamma_{\sigma,i:n} & \vdash e : (\sigma, i : n)T \quad i \notin \text{dom}(\sigma) \\
\Gamma_{\sigma,i:n} & \vdash e : (\sigma, i : n)F_d \\
\Gamma_{\sigma} & \vdash \sum_{i=1}^n e : (\sigma)T \\
\Gamma_{\sigma} & \vdash \sum_{i=1}^n e : (\sigma)F_d \\
\text{TyJudge 3)} & \quad \sigma(i) = d \quad \sigma' = \sigma \setminus d \quad 1 \leq i \leq d \\
\Gamma_{\sigma} & \vdash \sigma' : (\sigma')F_d \\
\Gamma_{\sigma} & \vdash \frac{\partial}{\partial x_i} \circ e : (\sigma)F_d \\
\text{TyJudge 4)} & \quad \frac{i, j \in \text{dom}(\sigma)}{
\Gamma_{\sigma} \vdash \delta_{ij} : (\sigma)T \\
\Gamma_{\sigma} & \vdash \epsilon_{ij} : (\sigma)T \\
\Gamma_{\sigma} & \vdash (\delta_{ij} \ast e) : (\sigma)T \\
\Gamma_{\sigma} & \vdash (\delta_{ij} \ast e) : (\sigma)F_d \\
\text{TyJudge 5)} & \quad \frac{\alpha \in \text{dom}(\sigma)}{
\Gamma_{\sigma} \vdash \epsilon_{\alpha} : (\sigma)T \\
\Gamma_{\sigma} & \vdash \epsilon_{\alpha} : (\sigma)T \\
\Gamma_{\sigma} & \vdash (\epsilon_{\alpha} \ast e) : (\sigma)T \\
\Gamma_{\sigma} & \vdash (\epsilon_{\alpha} \ast e) : (\sigma)F_d \\
\text{TyJudge 6)} & \quad \frac{\Gamma_{\sigma} \vdash \alpha : \tau}{
\Gamma_{\sigma} \vdash \epsilon_\alpha : \tau \\
\Gamma_{\sigma} & \vdash \epsilon_\alpha : \tau \\
\Gamma_{\sigma} & \vdash x : \text{Ten}[d] \\
\Gamma_{\sigma} & \vdash e : (\sigma)F_d \\
\Gamma_{\sigma} & \vdash e @ x : (\sigma)T \\
\text{TyJudge 7)} & \quad \frac{\Gamma_{\sigma} \vdash e : (\sigma)T}{
\Gamma_{\sigma} \vdash \text{lift}(e) : (\sigma)F_d} \\
\text{TyJudge 8)} & \quad \frac{\Gamma_{\sigma} \vdash e : (\sigma)T \quad \odot_1 \in \{\sqrt{}, -, \kappa, \exp, (\cdot)^n\}}{
\Gamma_{\sigma} \vdash \odot_1(e) : (\sigma)T} \\
\Gamma_{\sigma} & \vdash \odot_1(e) : (\sigma)T \\
\text{TyJudge 9)} & \quad \frac{\Gamma_{\sigma} \vdash e : (\sigma)T \quad \odot_1 \in \{\sqrt{}, -, \kappa, \exp, (\cdot)^n\}}{
\Gamma_{\sigma} \vdash \odot_1(e) : (\sigma)F_d} \\
\Gamma_{\sigma} & \vdash \odot_1(e) : (\sigma)F_d \\
\text{TyJudge 10)} & \quad \frac{\Gamma_{\sigma} \vdash e_1 : \tau \quad \odot_2 \in \{+, -\}}{
\Gamma_{\sigma} \vdash (e_1 \odot_2 e_2) : \tau} \\
\Gamma_{\sigma} & \vdash (e_1 \odot_2 e_2) : \tau \\
\Gamma_{\sigma} & \vdash e : \tau \\
\Gamma_{\sigma} & \vdash e : \tau \\
\text{TyJudge 11)} & \quad \frac{\Gamma_{\sigma} \vdash e_1 : (\sigma)T \quad \Gamma_{\sigma} \vdash e_2 : (\sigma)T}{
\Gamma_{\sigma} \vdash (e_1 \ast e_2) : (\sigma)T \\
\Gamma_{\sigma} & \vdash (e_1 \ast e_2) : (\sigma)F_d} \\
\text{TyJudge 12)} & \quad \frac{\Gamma_{\sigma} \vdash e_1 : (\sigma)T \quad \Gamma_{\sigma} \vdash e_2 : (\sigma)T}{
\Gamma_{\sigma} \vdash e_1 / e_2 : (\sigma)T \\
\Gamma_{\sigma} & \vdash e_1 / e_2 : (\sigma)F_d} \\
\Gamma_{\sigma} & \vdash e_1 / e_2 : (\sigma)F_d}
\end{align*}
\]

Figure 4.1: Typing Rules for each EIN expression.
Figure 4.2: The inversion lemma makes inferences based on a structural type judgements. Given a conclusion (left), we can infer something about the type \( \tau \) (right).

\[
\begin{align*}
&\Gamma_\sigma \vdash c : \tau \quad \mapsto \tau = (\sigma)T \\
&\Gamma_\sigma \vdash T_\alpha : \tau \quad \mapsto \tau = (\sigma)T \\
&\Gamma_\sigma \vdash F_\alpha : \tau \quad \mapsto \tau = (\sigma)F_d
\end{align*}
\]

\[\vdots\]

\[\Gamma_\sigma,i:n \vdash e_1 : (\sigma,i:n)F_d(\text{Ty}_\text{Inv} \ 3)\]

\[
\begin{array}{c}
\Gamma_\sigma \vdash \left( \sum_{i=1}^{i=n} (e_1) \right) : (\sigma)F_d(\text{Ty}_\text{Inv} \ 7) \\
\Gamma_\sigma \vdash (\sum_{i=1}^{i=n} (e_1)) \circ x : (\sigma)T
\end{array}
\]

Find \( \Gamma_\sigma \vdash (\sum_{i=1}^{i=n} (e_1)) \circ x \)

Given that \( \Gamma_\sigma \vdash e_1 : (\sigma,i:n)F_d \)

then \( \Gamma_\sigma \vdash e_1 \circ x : (\sigma,i:n)T \) by Ty\text{Judge} 7 and \( \Gamma_\sigma \vdash \sum_{i=1}^{i=n} (e_1 \circ x) : (\sigma)T \) by Ty\text{Judge} 3

\[\text{T( R4) OK}\]

\textbf{Chain rule:}

The rewrite rule ( R6) has the form \( \nabla_i \circ (e_1 \ast e_2) \rightarrow e_1(\nabla_i \circ e_2) + e_2(\nabla_i \circ e_1) \).

Find \( \Gamma_\sigma \vdash (\nabla_i (e_1 \ast e_2)) \)

This type of structure inside a derivative operation results in a field type.

Given the subterm: \( \Gamma_{\sigma \backslash i} \vdash e_1 \ast e_2 : (\sigma/\ i)F_d \)

then by Ty\text{Judge} 4 we know it’s derivative \( \Gamma_\sigma \vdash \nabla_i \circ (e_1 \ast e_2) : (\sigma)F_d \)

Find \( \Gamma_\sigma \vdash ((e_1) \text{ and } (e_2)) \)

\[
\begin{array}{c}
\Gamma_{\sigma \backslash i} \vdash e_1 \text{ and } e_2 : (\sigma \backslash i)F_d; (\text{Ty}_\text{Inv} \ 11) \\
(\Gamma_{\sigma \backslash i} \vdash e_1 \ast e_2 : (\sigma \backslash i)F_d(\text{Ty}_\text{Inv} \ 4))
\end{array}
\]

\( \Gamma_\sigma \vdash \nabla_{i:d}(e_1 \ast e_2) : (\sigma)F_d \)

Find \( \Gamma_\sigma \vdash (e_1 \ast \nabla_i(e_2)) \)

Given that \( \Gamma_\sigma \vdash e_1, e_2 : (\sigma \backslash i)F_d \) then \( \Gamma_\sigma \vdash \nabla_i(e_1), \nabla_i(e_2) : (\sigma)F_d \) by Ty\text{Judge} 4,
\[\Gamma_\sigma \vdash e_1 \ast \nabla_i (e_2), e_2 \ast \nabla_i (e_1) : (\sigma) F_d\] by Ty\_Judge 11 ,
and \[\Gamma_\sigma \vdash e_1 \ast \nabla_i (e_2) + e_2 \ast \nabla_i (e_1) : (\sigma) F_d\] by Ty\_Judge 10.

\begin{align*}
T( \text{R6}) &\text{ OK} \\
\end{align*}

**A quotient rule:**

The rewrite rule (R7) has the form \(\nabla_i \odot (e_1 e_2) \Rightarrow (\nabla_i \odot e_1) e_2 - e_1 (\nabla_i \odot e_2)\).

Find \(\Gamma_\sigma \vdash (\nabla_i \odot (\frac{e_1}{e_2}))\)

This type of structure inside a derivative operation results in a field type.

Given the subterm: \(\Gamma_\sigma/ i \vdash e_1 e_2 : (\sigma/ i) F_d\)
then by Ty\_Judge 4 we know it’s derivative \(\Gamma_\sigma \vdash \nabla_i \odot (e_1 e_2) : (\sigma) F_d\)

Find \(\Gamma_\sigma \vdash (e_1 \text{ and } e_2)\)
\(\Gamma_{\sigma \setminus i} \vdash e_1 : (\sigma \setminus i) F_d\) and \(\Gamma \vdash e_2 : () F_d\), (Ty\_Inv 12)

\[
\begin{align*}
(\Gamma_{\sigma \setminus i} \vdash \frac{e_1}{e_2} : (\sigma \setminus i) F_d &\text{(Ty\_Inv 4)} \\
\Gamma_\sigma \vdash \nabla_i d(\frac{e_1}{e_2}) : (\sigma) F_d &\text{by Ty\_Judge 11} \\
\end{align*}
\]

Given that \(\Gamma \vdash e_2 : () F_d\) then \(\Gamma \vdash e_2 \ast e_2 : () F_d\) by Ty\_Judge 11

Given that \(\Gamma \vdash e_2 : () F_d\) then \(\Gamma_d \vdash \nabla_i d \odot e_2 : (i) F_d\) by Ty\_Judge 4

Given that \(\Gamma_d \vdash \nabla_i d \odot e_2 : (i) F_d\) and \(\Gamma \vdash e_1 : (\sigma \setminus d) F_d\)

then \(\Gamma_\sigma \vdash e_1 \nabla_i d \odot e_2 : (\sigma) F_d\) by Ty\_Judge 11

Given that \(\Gamma_d \vdash \nabla_i d \odot e_1 : (i) F_d\) and \(\Gamma \vdash e_2 : (\sigma \setminus d) F_d\)

then \(\Gamma_\sigma \vdash e_2 \nabla_i d \odot e_1 : (\sigma) F_d\) by Ty\_Judge 11

Find \(\Gamma_\sigma \vdash (\frac{\nabla_i \odot e_1 e_2}{e_2} - \frac{e_1 (\nabla_i \odot e_2)}{e_2})\)

Given that \(\Gamma_\sigma \vdash ((\nabla_i \odot e_1) \ast e_2), (e_1 \ast \nabla_i \odot e_2) : (\sigma) F_d\) and \(\Gamma \vdash e_2 \ast e_2 : () F_d\)

then \(\Gamma_\sigma \vdash ((\nabla_i \odot e_1) \ast e_2) - (e_1 \ast \nabla_i \odot e_2) : (\sigma) F_d\) by Ty\_Judge 10

and \(\Gamma_\sigma \vdash (\frac{\nabla_i \odot e_1 e_2 - e_1 (\nabla_i \odot e_2)}{e_2} : (\sigma) F_d\) by Ty\_Judge 12

\begin{align*}
T( \text{R7}) &\text{ OK} \\
\end{align*}

**A trig operation:**

74
The rewrite rule (R10) has the form $\nabla_i \diamond (\text{sine}(e_1)) \Rightarrow (\text{cosine}(e_1)) \ast (\nabla_i \diamond e_1)$.

Find $\Gamma \vdash (\nabla_i \diamond (\text{sine}(e_1)))$

This type of structure inside a derivative operation results in a field type

and the $\text{sine}(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \text{sine}(e_1) : () \mathcal{F}_d$ then $\Gamma_i \vdash (\nabla_i \diamond (\text{sine}(e_1))) : (i) \mathcal{F}_d$ by TyJudge 4

Find $\Gamma \vdash (e_1)$

$\Gamma \vdash e_1 : () \mathcal{F}_d (\text{TyInv} \ 9)$

$\Gamma \vdash \text{sine}(e_1) : () \mathcal{F}_d$

Find $\Gamma \vdash ((\text{cosine}(e_1)) \ast (\nabla_i \diamond e_1))$

Given that $\Gamma \vdash e_1 : () \mathcal{F}_d$ then $\Gamma_d \vdash \nabla_i \diamond e_1 : (i) \mathcal{F}_d$ by TyJudge 4,

$\Gamma \vdash \text{cosine}(e_1) : () \mathcal{F}_d$ by TyJudge 9, and $\Gamma_d \vdash (\text{cosine}(e_1)) \ast (\nabla_i \diamond e_1) : (i) \mathcal{F}_d$ by TyJudge 11.

T( R10) OK

An algebraic operation:

The rewrite rule (R27) has the form $\frac{e_1}{e_3} \frac{e_2}{e_3} \Rightarrow \frac{e_1 e_2}{e_3}$.

$\Gamma \vdash \frac{e_1}{e_3}, \frac{e_2}{e_3} : \tau$

case analysis on $\tau$, $\tau = (\sigma) \mathcal{T}$

Find $\Gamma \vdash (\tau(e_1), \tau(e_2), \text{ and } \tau(e_3))$

$\Gamma \vdash e_1 : (\sigma) \mathcal{T}, \Gamma \vdash e_2 : (\mathcal{T}) (\text{TyInv} \ 12)$

$\Gamma \vdash \frac{e_1}{e_2} : (\sigma) \mathcal{T}$ and $e_3 : (\mathcal{T}) (\text{TyInv} \ 12)$

$\Gamma \vdash \frac{e_1}{e_2}, \frac{e_1}{e_3} : (\sigma) \mathcal{T}$

Find $\Gamma \vdash (\frac{e_1}{e_2 e_3})$

Given that $\Gamma \vdash e_1 : (\sigma) \mathcal{T}, \Gamma \vdash e_2, e_3 : () \mathcal{T}$

then $\Gamma \vdash e_2 \ast e_3 : () \mathcal{T}$ by (TyJudge 11), and $\Gamma \vdash \frac{e_1}{e_2 e_3} : (\sigma) \mathcal{T}$ by (TyJudge 12).

T( R27 for $\tau = (\sigma) \mathcal{T}$)

$\tau = (\sigma) \mathcal{F}_d$

Similar to previous case
A complicated $\delta$ typing judgment:

We define a few variables $\sigma_2 = \sigma'/ij$, $\sigma_j = \sigma'/j$, and $\sigma_i = \sigma'i/j$.

The rewrite rule (R40) has the form $\delta_{ij} \nabla_j \diamond e_1 \Rightarrow \nabla_i \diamond (e_1)$.

Claim $\Gamma \sigma_2 \vdash e_1 : (\sigma_2)F_d$.

Given that $\Gamma \sigma_2 \vdash e_1 : (\sigma_2)F_d$ then $\Gamma \sigma_j \vdash \nabla_j \diamond e_1 : (\sigma_j)F_d$ by TyJudge 4.

We switch the indices when applying the $\delta$, so that $\Gamma \sigma_i \vdash \delta_{ij}(\nabla_j \diamond e_1) : (\sigma_i)F_d$ by TyJudge 5.

Find $\Gamma \sigma_2 \vdash e_1 : (\sigma_1)F_d$ then $\Gamma \sigma_i \vdash \nabla_i \diamond e_1 : (\sigma_i)F_d$ by TyJudge 4.

T(R40) OK

Adding to index environment:

The rewrite rule (R41) has the form $\Sigma(se_1) \Rightarrow s\Sigma e_1$.

$\Gamma \sigma \vdash \sum_{i=1}^{i=n}(s \ast e) : \tau$

case analysis on $\tau$

$\tau = (\sigma)F_d$

Find $\Gamma \sigma \vdash (s$ and $e_1)$

$\Gamma_{\sigma,i:n} \vdash s : (\ast)F_d$(TyInv 11)  $\Gamma_{\sigma,i:n} \vdash e_1 : (\sigma,i:n)F_d$

$\Gamma_{\sigma,i:n} \vdash s \ast e_1 : (\sigma,i:n)F_d$(TyInv 3)

$\Gamma \sigma \vdash (\sum_{i=1}^{i=n}(s \ast e)) : (\sigma)F_d$

Find $\Gamma \sigma \vdash (s \sum_{i=1}^{i=n}(e_1))$

Given that $\Gamma \sigma \vdash e_1 : (\sigma,i:n)F_d$ and $\Gamma \vdash s : ()F_d$

then $\Gamma \sigma \vdash \sum_{i=1}^{i=n}(e_1) : (\sigma)F_d$ by TyJudge 3 and $\Gamma \sigma \vdash s \ast \sum_{i=1}^{i=n}(e_1) : (\sigma)F_d$ by TyJudge 11.

$\tau = (\sigma)F$ similar to above case.

T(R41) OK
\[ \text{value} = \begin{cases} \text{Real}(n) & n \in \mathcal{R} \\ \text{Tensor}[p \cdot x] & \text{index tensor argument } p \text{ using basis values } x \\ \text{Field}(V[\alpha], \nabla[\beta]) & \text{source of data } V, \alpha \text{ and } \beta \\ \text{corresponds to the component of image term and differentiation, respectively} \\ \text{App}(v)[x] & \text{Probe of a field value } v \text{ at tensor value } x \\ E_\alpha & \text{Reduces Levi-Civita tensor} \\ K_{ij} & \text{Reduces Kronecker delta function} \end{cases} \]

Figure 4.3: Value definitions for each EIN expression

4.2 Value Preservation

4.2.1 Value Definition

To show that the rewriting system preserves the semantics of the program, we have to give a dynamic semantics to EIN expressions. We assume a set of values \( v \in \text{VALUE} \) that include tensors, fields\(^1\), images and kernel. Rather than define the meaning of an expression to be a function from indices to values, we include a mapping \( \rho \) from index variables to indices as part of the dynamic environment. We define a dynamic environment to be \( \Psi_\rho \in (\text{INDEXVAR} \xrightarrow{\text{fin}} \mathbb{Z}) \times (\text{VAR} \xrightarrow{\text{fin}} \text{VALUE}) \), where \text{VALUE} \ is the domain of computational values (\textit{e.g.}, tensors, fields, \textit{etc.}). We define the meaning of an EIN expression using a big-step semantics \( \Psi_\rho \vdash e \Downarrow v \). Each EIN expression evaluates to a value. We describe values next and present the full value judgements in Figure 4.2.1.

**Tensors and Fields** Inspired by Equation 1.7 the value of a vector is evaluated as

\[ \Psi_\rho \vdash T_i \Downarrow \text{Tensor}[T \cdot b_i] \]

\(^1\) Note that in this semantics, field values are the reconstructed fields (and their derivatives, \textit{etc.}), not the ideal field being approximated by reconstruction.
The $b_i$ represents the basis value and index $i$ in EIN expressions. The full tensor judgement

\[
\text{ValJudge 2) } \Psi_\rho \vdash T_\alpha \Downarrow \text{Tensor}[T \cdot b_{\alpha 0} \cdot b_{\alpha 1} \ldots b_{\alpha n}]
\]

is used to represent an arbitrary sized tensor. Similarly a field value is created:

\[
\text{ValJudge 3) } \Psi_\rho \vdash F_\alpha \Downarrow \text{Field}(F[b_{\alpha 0} \cdot b_{\alpha 1} \ldots b_{\alpha n}], \nabla[[]])
\]

and convolution has the following judgement

\[
\text{ValJudge 4) } \Psi_\rho \vdash V_\alpha \circledast H^\beta \Downarrow \text{Field}(V[\alpha], \nabla[\beta])
\]

The lift operation is used to lift a tensor to a field level. The value of a lifted term is the value of that term.

\[
\text{ValJudge 5) } \Psi_\rho \vdash e \Downarrow v
\]

\[
\Psi_\rho \vdash \text{lift}(e) \Downarrow v
\]

**Algebraic operators** We support arithmetic operations on and between values. The summation expression can be evaluated with the following judgement:

\[
\frac{\Psi_\rho \vdash e \Downarrow v}{\Psi_\rho \vdash \sum_{i=1}^{n} e \Downarrow \sum_{i=1}^{n} v}
\]

The summation operator is applied to the value $v$. Generally, the judgement for unary operators ($\circ_1 \in \{\sum, \sqrt{}, - |, \kappa, \exp, (\cdot)^n\}$) is as follows:

\[
\text{ValJudge 6) } \frac{\Psi_\rho \vdash e_1 \Downarrow \text{Real}(r_1)}{\Psi_\rho \vdash \circ_1 e_1 \Downarrow \text{Real}(\circ_1 r_1)}
\]
\[ \Psi_\rho \vdash e_1 \downarrow \text{Tensor}[e_1 \cdot b_1] \quad \Psi_\rho \vdash e_1 \downarrow \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) \]

\[ \Psi_\rho \vdash \odot_1 e_1 \downarrow \odot_1 \text{Tensor}[e_1 \cdot b_1] \quad \Psi_\rho \vdash \odot_1 e_1 \downarrow \odot_1 \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) \]

The binary operators \((\odot_2 = + | - | * | / )\) can be applied between values.

\[
\begin{align*}
\text{Val}_{\text{Judge } 7)} & \quad \Psi_\rho \vdash e_1 \downarrow \text{Real}(r_1) \quad \Psi_\rho \vdash e_2 \downarrow \text{Real}(r_2) \\
\text{Ψ}_\rho \vdash (e_1 \odot_2 e_2) \downarrow \text{Real}(r_1 \odot_2 r_2)
\end{align*}
\]

\[
\begin{align*}
\Psi_\rho \vdash e_1 \downarrow \text{Tensor}[e_1 \cdot b_1] & \quad \Psi_\rho \vdash e_2 \downarrow \text{Tensor}[e_2 \cdot b_2] \\
\Psi_\rho \vdash (e_1 \odot_2 e_2) \downarrow \text{Tensor}[e_1 \cdot b_1 \odot_2 \text{Tensor}[e_2 \cdot b_2]
\end{align*}
\]

\[
\begin{align*}
\Psi_\rho \vdash e_1 \downarrow \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) & \quad \Psi_\rho \vdash e_2 \downarrow \text{Field}(e_2[\alpha_2], \nabla[\beta_2]) \\
\Psi_\rho \vdash (e_1 \odot_2 e_2) \downarrow \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) \odot_2 \text{Field}(e_2[\alpha_2], \nabla[\beta_2])
\end{align*}
\]

**Differentiation**  The derivative is applied to an EIN expression.

\[
\begin{align*}
\text{Val}_{\text{Judge } 8)} & \quad \Psi_\rho \vdash e \downarrow \text{Field}(e[\alpha], \nabla[\beta / i]) \\
\Psi_\rho \vdash \frac{\partial}{\partial x_i} \circ e \downarrow \text{Field}(e[\alpha], \nabla[\beta]) \\
\Psi_\rho \vdash \frac{\partial}{\partial x_i} \circ e \downarrow \nabla_i v
\end{align*}
\]

When the expression reduces to a field value \(\text{Field}(e[\alpha], \nabla[\beta])\) then we add the differentiated index \(i\) directly to the field value. Otherwise, we apply a differentiation operator to the value \(v\). The new value \(\nabla_i v\) is subject to the product rule, quotient rule, and other forms of tensor calculus based rewrites [64].

**Probe**  The probe expression \(e@x\) creates a \(\text{App}(v)[x]\) value.

\[
\begin{align*}
\Psi_\rho \vdash e \downarrow v \\
\Psi_\rho \vdash e@x \downarrow \text{App}(v)[x]
\end{align*}
\]

Consider the mathematical definition for \(\sqrt{-}\) operation applied to fields: \(\sqrt{f} = \lambda x. \sqrt{f(x)}\).
Probing the term creates $\sqrt{f(x)} = \sqrt{f(x)}$. Recall, that we generalized the definition for lifted operators in Equation 2.1.

$$P^\uparrow(f) = \lambda x. P(f(x))$$

so then a probe is written as

$$P^\uparrow(e)@x = P(e@x)$$

The following are judgements for composing functions.

\[
\begin{align*}
\Psi_\rho \vdash e_1 \downarrow v_1 & \quad \Psi_\rho \vdash e_2 \downarrow v_2 \quad \odot_2 = + \quad | - | \star / \\
\Psi_\rho \vdash (e_1 \odot_2 e_2)@x \downarrow App(v_1)[x] \odot_2 App(v_2)[x]
\end{align*}
\]

\[
\begin{align*}
\Psi_\rho \vdash e_1 \downarrow v_1 & \quad \Psi_\rho \vdash e_2 \downarrow v_2 \quad \odot_2 = + \quad | - | \star / \\
\Psi_\rho \vdash (e_1 \odot_2 e_2)@x \downarrow App(v_1)[x] \odot_2 App(v_2)[x]
\end{align*}
\]

There are exceptions when the probe term is not a field. If the probed term is a permutation term then the value is simply the value of the permutation term. If the probed term is a lifted tensor term, then the value is the value of that tensor term.

\[
\begin{align*}
\Psi_\rho \vdash e \downarrow v & \quad \Psi_\rho \vdash \delta_{ij} \downarrow v & \quad \Psi_\rho \vdash E_{\alpha} \downarrow v \\
\Psi_\rho \vdash \text{lift}(e)@e \downarrow v & \quad \Psi_\rho \vdash \delta_{ij}@e \downarrow v & \quad \Psi_\rho \vdash E_{\alpha}@e \downarrow v
\end{align*}
\]

**Permutation values** The epsilon and kronecker delta functions are each reduced to a distinct permutation value $(E_{\alpha}, K_{ij})$.

\[
\begin{align*}
\Psi_\rho \vdash \varepsilon_{ijk} \downarrow E_{\alpha} & \quad \Psi_\rho \vdash \delta_{ij} \downarrow K_{ij}
\end{align*}
\]

The value for $\varepsilon_{ijk}$ is subject to properties Equation 1.1. The value for $\delta_{ij}$ is subject to properties Equation 1.2, Equation 1.3, and Equation 1.4. We combine permutation values with tensor values as
\[ K_{ij} \ast \text{Tensor}[T \cdot \beta] \rightarrow \text{Tensor}[T \cdot b_i \cdot b_j \cdot \beta] \quad (4.1) \]

Combine permutation values with field values as

\[ K_{ij} \ast \text{Field}(F[b_j \ldots \alpha], \nabla[\beta]) \rightarrow \text{Field}(F[b_i \cdot b_j \cdot b_j \ldots \alpha], \nabla[\beta]) \quad (4.2) \]

\[ K_{ij} \ast \text{Field}(F[\alpha], \nabla[b_j \ldots \beta]) \rightarrow \text{Field}(F[\alpha], \nabla[b_i \cdot b_j \cdot b_j \ldots \beta]) \quad (4.3) \]

The full set of value judgments are in Figure 4.2.1.
ValJudge 1  \[ \Psi_\rho \vdash c \Downarrow \text{Real}(c) \]

ValJudge 2  \[ \Psi_\rho \vdash T_\alpha \Downarrow \text{Tensor}[T \cdot b_{\alpha_0} \cdot b_{\alpha_1} \ldots b_{\alpha_n}] \]

ValJudge 3  \[ \Psi_\rho \vdash F_\alpha \Downarrow \text{Field}(F[b_{\alpha_0} \cdot b_{\alpha_1} \ldots b_{\alpha_n}], \nabla[]) \]

ValJudge 4  \[ \Psi_\rho \vdash V_\alpha \odot H^\beta \Downarrow \text{Field}(V[\alpha], \nabla[\beta]) \]

ValJudge 5  \[ \Psi_\rho \vdash e \Downarrow v \quad \Psi_\rho \vdash \text{lift}(e) \Downarrow v \]

ValJudge 6  \[ \odot_1 \in \{\sum | \sqrt{\,} | - | \kappa | \exp | (\cdot)^n \} \]
\[ \Psi_\rho \vdash e_1 \Downarrow \text{Real}(r1) \]
\[ \Psi_\rho \vdash e_1 \Downarrow \text{Tensor}[e_1 \cdot b_1] \]
\[ \Psi_\rho \vdash \odot_1 e_1 \Downarrow \text{Real}(\odot_1 r1) \]
\[ \Psi_\rho \vdash \odot_1 e_1 \Downarrow \text{Tensor}[e_1 \cdot b_1] \]
\[ \Psi_\rho \vdash e_1 \Downarrow \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) \]
\[ \Psi_\rho \vdash \odot_1 e_1 \Downarrow \odot_1 \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) \]

ValJudge 7  \[ \odot_2 = + | - | * | / \]
\[ \Psi_\rho \vdash e_1 \Downarrow \text{Real}(r1) \]
\[ \Psi_\rho \vdash e_2 \Downarrow \text{Real}(r2) \]
\[ \Psi_\rho \vdash (e_1 \odot_2 e_2) \Downarrow \text{Real}(r1 \odot_2 r2) \]
\[ \Psi_\rho \vdash e_1 \Downarrow \text{Tensor}[e_1 \cdot b_1] \]
\[ \Psi_\rho \vdash e_2 \Downarrow \text{Tensor}[e_2 \cdot b_2] \]
\[ \Psi_\rho \vdash (e_1 \odot_2 e_2) \Downarrow \text{Tensor}[e_1 \cdot b_1 \odot_2 e_2 \cdot b_2] \]
\[ \Psi_\rho \vdash e_1 \Downarrow \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) \]
\[ \Psi_\rho \vdash e_2 \Downarrow \text{Field}(e_2[\alpha_2], \nabla[\beta_2]) \]
\[ \Psi_\rho \vdash (e_1 \odot_2 e_2) \Downarrow \text{Field}(e_1[\alpha_1], \nabla[\beta_1]) \odot_2 \text{Field}(e_2[\alpha_2], \nabla[\beta_2]) \]

ValJudge 8  \[ \Psi_\rho \vdash e \Downarrow v \]
\[ \Psi_\rho \vdash e \Downarrow \text{Field}(e[\alpha], \nabla[\beta/ i]) \]
\[ \Psi_\rho \vdash \partial_{\partial_i} v \Downarrow \nabla_i v \]
\[ \Psi_\rho \vdash \partial_{\partial_i} \circ e \Downarrow \text{Field}(e[\alpha], \nabla[\beta]) \]

ValJudge 9  \[ \Psi_\rho \vdash e \Downarrow v \]
\[ \Psi_\rho \vdash (\odot_1 e) \Downarrow x \Downarrow \odot_1 \text{App}(v_1)[x] \]
\[ \Psi_\rho \vdash e_1 \Downarrow v_1 \quad \Psi_\rho \vdash e_2 \Downarrow v_2 \quad \odot_2 = + | - | * | / \]
\[ \Psi_\rho \vdash (e_1 \odot_2 e_2) \Downarrow x \Downarrow \text{App}(v_1)[x] \odot_2 \text{App}(v_2)[x] \]

ValJudge 10  \[ \Psi_\rho \vdash \text{lift}(e) \Downarrow v \]
\[ \Psi_\rho \vdash \delta_{ij} \Downarrow v \]
\[ \Psi_\rho \vdash \varepsilon_\alpha \Downarrow v \]
\[ \Psi_\rho \vdash \delta_{ij} \Downarrow v \quad \Psi_\rho \vdash \varepsilon_\alpha \Downarrow v \]
\[ \Psi_\rho \vdash \varepsilon_\alpha \Downarrow v \]

ValJudge 11  \[ \Psi_\rho \vdash \delta_{ij} \Downarrow K_{ij} \quad \Psi_\rho \vdash \varepsilon_\alpha \Downarrow E_\alpha \]

Figure 4.4: Value Judgements for each EIN expression.
4.2.2 Value Preservation Theorem

Our correctness theorem states that our rewrite rules do not change the value of an expression with respect to a dynamic environment, assuming that the expression and dynamic environment are both type-able in the same static environment.

**Theorem 4.2.1 (Value Preservation).** If \( \Gamma \vdash_{\sigma} e : \tau \), \( \Gamma \vdash_{\sigma} \Psi_{\rho} \text{ ok} \), \( e \rightarrow e' \), and \( \Psi_{\rho} \vdash e \Downarrow v \), then \( \Psi_{\rho} \vdash e' \Downarrow v \).

**Proof.** Given a derivation \( d \) of the form \( e \rightarrow e' \) we state \( V(d) \) as a shorthand for the claim that the derivation preserves the value of the expression \( e \). By case analysis and algebraic reasoning provided below are key cases of the proof (Section A.2).

Composes functions:
The rewrite rule (R1) has the form \((e_1 \odot_n e_2)@x \rightarrow (e_1@x) \odot_n (e_2@x)\) where \( \odot_n = \ast | / \).

Claim \((e_1 \odot_n e_2)@x\) evaluates to \(v\)

Assume subterms: \( \Psi_{\rho} \vdash e_1 \Downarrow v_1, \Psi_{\rho} \vdash e_2 \Downarrow v_2 \)

then \( \Psi_{\rho} \vdash (e_1 \odot_n e_2)@x \Downarrow App(v_1)[x] \odot_n App(v_2)[x] \) by (ValJudge 9),

\( \Psi_{\rho} \vdash e_1@x \Downarrow App(v_1)[x] \) by (ValJudge 9), \( \Psi_{\rho} \vdash e_2@x \Downarrow App(v_2)[x] \) by (ValJudge 9),

and \( \Psi_{\rho} \vdash (e_1@x) \odot_n (e_2@x) \Downarrow App(v_1)[x] \odot_n App(v_2)[x] \) by (ValJudge 7)

The last step leads to \((e_1@x) \odot_n (e_2@x) \Downarrow v\)

\( V( \text{ R1} ) \) OK

Chain rule:
The rewrite rule (R6) has the form \( \nabla_i \diamond (e_1 * e_2) \rightarrow e_1(\nabla_i \diamond e_2) + e_2(\nabla_i \diamond e_1) \).

Claim \( \nabla_i \diamond (e_1 * e_2) \) evaluates to \(v\)

Assume subterms: \( \Psi_{\rho} \vdash e_1 \Downarrow v_1, \Psi_{\rho} \vdash e_2 \Downarrow v_2 \)

then \( \Psi_{\rho} \vdash e_1 * e_2 \Downarrow v_1 * v_2 \) by (ValJudge 7), and \( \Psi_{\rho} \vdash \nabla_i \diamond (e_1 * e_2) \Downarrow \nabla_i (v_1 * v_2) \) by (ValJudge 8)

The value of \( v \) is \( v = \nabla_i (v_1 * v_2) \)
\[v = (v_1 \nabla_i v_2) + (v_2 \nabla_i v_1)\] by applying the product rule

Given that \(e_1 \downarrow v_1, e_2 \downarrow v_2\) then \(\Psi_\rho \vdash \nabla_i \diamond e_1 \downarrow \nabla_i v_1, \nabla_i \diamond e_2 \downarrow \nabla_i v_2\) by (ValJudge 8),

\[\Psi_\rho \vdash e_1(\nabla_i \diamond e_2) \downarrow v_1 \nabla_i v_2, e_2(\nabla_i \diamond e_1) \downarrow v_2 \nabla_i v_1\] by (ValJudge 7),

and \(\Psi_\rho \vdash e_1(\nabla_i \diamond e_2) + e_2(\nabla_i \diamond e_1) \downarrow (v_1 \nabla_i v_2) + (v_2 \nabla_i v_1)\) by (ValJudge 7).

The last step leads to \(e_1 \nabla_i \diamond e_2 + e_2 \nabla_i \diamond e_1 \downarrow v\)

\[V(\ R6) \ OK\]

**Applies a trigonometric identity:**

The rewrite rule (R9) has the form \(\nabla_i \diamond (\cosine(e_1)) \Rightarrow (-\sine(e_1)) \ast (\nabla_i \diamond e_1)\).

Claim \(\nabla_i \diamond (\cosine(e_1))\) evaluates to \(v\)

Assume that \(\Psi_\rho \vdash e_1 \downarrow v_1\) then \(\Psi_\rho \vdash \cosine(e_1) \downarrow \cosine(v_1)\) by (ValJudge 6),

and \(\Psi_\rho \vdash \nabla_i \diamond (\cosine(e_1)) \downarrow \nabla_i(\cosine(v_1))\) by (ValJudge 8)

The value of \(v\) is \(\nabla_i(\cosine(v_1))\)

\[v = \nabla_i(\cosine(v_1)) = (-\sine(v_1)) \ast (\nabla_i \diamond v_1)\] by applying trigonometric identity

Since subterm: \(\Psi_\rho \vdash e_1 \downarrow v_1\) then \(\Psi_\rho \vdash \nabla_i \diamond e_1 \downarrow \nabla_i v_1\) by (ValJudge 8),

\[\Psi_\rho \vdash \sine(e_1) \downarrow \sine(v_1)\] by (ValJudge 6), \(\Psi_\rho \vdash -\sine(e_1) \downarrow -\sine(v_1)\) by (ValJudge 6),

and \(\Psi_\rho \vdash (-\sine(e_1)) \ast (\nabla_i \diamond e_1) \downarrow (-\sine(v_1)) \ast (\nabla_i v_1)\) by (ValJudge 7)

The last step leads to \((-\sine(e_1)) \ast (\nabla_i \diamond e_1) \downarrow v\)

\[V(\ R9) \ OK\]

**Flattening:**

The rewrite rule (R24) has the form \(e_1 - 0 \Rightarrow e_1\).

Claim \(e_1 - 0\) evaluates to \(v\)

Assume that \(e_1 \downarrow v'\) then \(\Psi_\rho \vdash e_1 - 0 \downarrow v' - \Real(0)\) by (ValJudge 1, ValJudge 7).

The value of \(v\) is \(v' - \Real(0)\).

By using algebraic reasoning: \(v' - \Real(0) = v'\). Since \(e_1 - 0 \downarrow v\) and \(e_1 - 0 \downarrow v'\) then \(v = v'\)

The last step leads to \(e_1 \downarrow v\)
Simple Algebraic rewrite:

The rewrite rule (R32) has the form $\sqrt{e_1} \ast \sqrt{e_1} \Rightarrow e_1$.

Claim $\sqrt{e_1} \ast \sqrt{e_1}$ evaluates to $v$

Assume that $e_1 \Downarrow v'$

then $\Psi_\rho \vdash \sqrt{e_1} \Downarrow \sqrt{v'}$ by (ValJudge 6), and $\Psi_\rho \vdash \sqrt{e_1} \sqrt{e_1} \Downarrow \sqrt{v'} \sqrt{v'}$ by (ValJudge 7)

The value of $v$ is $\sqrt{v'} \ast \sqrt{v'}$

By using algebraic reasoning to analyze $v$

$v = \sqrt{v'} \ast \sqrt{v'} = v'$ by reduction

The last step leads to $e_1 \Downarrow v$

V(R32) OK

Applies permutation judgement:

The rewrite rule (R33) has the form $E_{ijk} \nabla_{ij} \Diamond e_1 \Rightarrow \text{lift}(0)$.

Claim $E_{ijk} \nabla_{ij} \Diamond e_1$ evaluates to $v$

Assume that $\Psi_\rho \vdash e_1 \Downarrow v_1$.

This type of structure inside a derivative evaluates to a field value.

Therefore, $v_1 = \text{Field}(e_1[\alpha], \nabla[\beta])$.

Given that $\Psi_\rho \vdash e_1 \Downarrow v_1$ and $\Psi_\rho \vdash E_{ijk} \Downarrow E_{ijk}$ by (ValJudge 11)

then $\Psi_\rho \vdash \nabla_{ij} \circ e_1 \Downarrow \text{Field}(e_1[\alpha], \nabla[\beta \cdot b_i \cdot b_j])$ by (ValJudge 8),

and $\Psi_\rho \vdash E_{ijk} \nabla_{ij} \circ e_1 \Downarrow E_{ijk} \ast \text{Field}(e_1[\alpha], \nabla[\beta \cdot b_i \cdot b_j])$ by ((4.1))

The value of $v$ is $E_{ijk} \ast \text{Field}(e_1[\alpha], \nabla[\beta \cdot b_i \cdot b_j])$

By using algebraic reasoning to analyze $v$

$v = E_{ijk} \ast \text{Field}(e_1[\alpha], \nabla[\beta \cdot b_j \cdot b_i])$ by flipping order of differentiation

$v = -E_{jik} \ast \text{Field}(e_1[\alpha], \nabla[\beta \cdot b_j \cdot b_i])$ by anti-cyclic movement of indices Equation 1.1

$v = -E_{nmk} \ast \text{Field}(e_1[\alpha], \nabla[\beta \cdot b_n \cdot b_m])$ by swapping $i \rightarrow m$ and $j \rightarrow n$
\[ v = -E_{ijk} * \text{Field}(e_1[\alpha], \nabla[\beta \cdot b_i \cdot b_j]) \] by swapping \( m \rightarrow j \) and \( n \rightarrow i \)

Given \( e_2 = E_{ijk} \nabla_{ij} \circ e_1, e_2 \downarrow v, \) and \( e_2 \downarrow -v \) then using algebraic reasoning \( v = \text{Real}(0). \)

Lastly, \( \Psi_\rho \vdash \text{lift}(0) \downarrow \text{Real}(0) \) by (ValJudge 1).

The last step leads to \( \text{lift}(0) \downarrow v \)

\[ V(\text{R}33) \, OK \]

**Applying epsilon terms** :

The rewrite rule (R35) has the form \( E_{ijk}E_{ilm} \implies \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}. \)

Claim \( E_{ijk}E_{ilm} \) evaluates to \( v \)

Given that \( E_{ijk} \downarrow E_{ijk} \) and \( E_{pqr} \downarrow E_{pqr} \) then \( E_{ijk}E_{pqr} \downarrow E_{ijk}E_{pqr}. \)

The value of \( v \) is \( E_{ijk}E_{pqr}. \)

Consider the product of two \( E \) expressions as

\[
E_{ijk}E_{pqr} \rightarrow \begin{vmatrix}
K_{ip} & K_{iq} & K_{ir} \\
K_{jp} & K_{jq} & K_{jr} \\
K_{kp} & K_{kq} & K_{kr}
\end{vmatrix}
\]

\[
\rightarrow K_{ip}(K_{jq}K_{kr} - K_{jr}K_{kq}) + K_{iq}(K_{jr}K_{kp} - K_{jp}K_{kr}) + K_{ir}(K_{jp}K_{kq} - K_{jq}K_{kp})
\]

Rewriting so that there is a shared index \( (p = i): \)

\[
\rightarrow K_{ii}K_{jq}K_{kr} - K_{ii}K_{jr}K_{kq} + K_{iq}K_{jr}K_{ki} - K_{iq}K_{ji}K_{kr} + K_{ir}K_{ji}K_{kq} - K_{ir}K_{jq}K_{ki}
\]

Applying Equation 1.4:

\[
\rightarrow 3K_{jq}K_{kr} - 3K_{jr}K_{kq} + K_{iq}K_{jr}K_{ki} - K_{iq}K_{ji}K_{kr} + K_{ir}K_{ji}K_{kq} - K_{ir}K_{jq}K_{ki}
\]

Applying Equation 1.3:

\[
\rightarrow 3K_{jq}K_{kr} - 3K_{jr}K_{kq} + K_{kq}K_{jr} - K_{jq}K_{kr} + K_{jr}K_{kq} - K_{kr}K_{jq}
\]

Reduces to:

\[
\rightarrow K_{jq}K_{kr} - K_{jr}K_{kq}
\]

Match indices to rule \( (q \rightarrow l \) and \( r \rightarrow m) \)

\[
\rightarrow K_{jl}K_{km} - K_{jm}K_{kl}
\]

Getting the value of \( \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}. \)

Given that \( \Psi_\rho \vdash \delta_{jl} \downarrow K_{jl}, \delta_{km} \downarrow K_{km}, \delta_{jm} \downarrow K_{jm}, \delta_{kl} \downarrow K_{kl} \) by (ValJudge 11).
then $\Psi_\rho \vdash \delta_{jl}\delta_{km} \Downarrow K_{jl}K_{km}$ \hspace{1mm} $\delta_{jm}\delta_{kl} \Downarrow K_{jm}K_{kl}$ by (Val\ Judge 7)

and $\Psi_\rho \vdash \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} \Downarrow K_{jl}K_{km} - K_{jm}K_{kl}$ by (Val\ Judge 7)

The last step leads to $\delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} \Downarrow v$

$\forall( R35)$ OK

Applies reduction on basis functions:

The rewrite rule ( R36) has the form $\delta_{ij}T_j \rightarrow T_i$.

Claim $\delta_{ij}T_j$ evaluates to $v$

Given that $\Psi_\rho \vdash T_j \Downarrow \text{Tensor}[T \cdot b_j]$ by Val\ Judge 2 and $\Psi_\rho \vdash \delta_{ij} \Downarrow K_{ij}$ by Val\ Judge 11

then $\Psi_\rho \vdash \delta_{ij}T_j \Downarrow \text{Tensor}[T \cdot b_j \cdot b_i \cdot b_j]$ by Equation 4.1

The value of $v$ is $\text{Tensor}[T \cdot b_j \cdot b_i \cdot b_j]$

By using algebraic reasoning to analyze $v$

$v = \text{Tensor}[T \cdot b_i]$ by reducing value $b_j \cdot b_j$ using Equation 1.5

Lastly, $\Psi_\rho \vdash T_i \Downarrow \text{Tensor}[T \cdot b_i]$ by (Val\ Judge 2)

The last step leads to $T_i \Downarrow v$

$\forall( R36)$ OK

4.3 Termination

4.3.1 Normal Form

After normalization, EIN expressions are in normal form $\mathcal{N}$. Some tensors, constants, and permutation terms that are in normal form include:

$T_{\alpha\cdot} , c \neq 0, \delta_{ij}, \mathcal{E}_{ij}, \text{ and } \mathcal{E}_{ijk}$
The field forms $\mathcal{F}$ include:

$$F_\alpha, V \otimes H, \nabla \diamond F_\alpha$$

All differentiation is applied (via product rule or otherwise) so it only exists if operating on a field term:

$$\nabla \diamond F_\alpha$$

until it is pushed down to the convolution kernel:

$$V \otimes \nabla H$$

The only probed terms are field forms $\mathcal{F}$:

$$F_\alpha \# T, (V \otimes H) \# x, \text{ and } \nabla \diamond F \# x$$

Some unary operations are in normal form, as long as their sub-term $e_1$ is in normal form:

$$\text{sine}(e_1), \text{lift}(e_1), \sqrt{e_1}, \exp(e_1)$$

Other arithmetic operations cannot have a zero constant sub-term (3.4):

$$-e_1, e_1 + e_2, e_1 - e_2, e_1 \ast e_2, \frac{e_1}{e_2}$$

The division structure is subject to algebraic rewrites (3.4). The normal form of the product and summation structure is more restricted in part because of index-based rewrites. Normal form is presented more formally next:

**Normal Form** The following grammar specifies the subset $\mathcal{N}$ of EIN expressions that are in *normal form*:
\[ N ::= A | c \]
\[ A ::= D | G \]
\[ D ::= B | -G \]
\[ G ::= B | D \]
\[ B ::= T \alpha | F | F @ T \alpha | c \neq 0 | \delta_{ij} \mid \epsilon_{ijk} \]
\[ \mid A + A \mid A - A \mid A \ast A \mid \sqrt{N} \mid \sum N \]
\[ \mid \text{exp}(N) \mid \sqrt{N} \mid \text{lift}(N) \mid \kappa(N) \]
\[ F ::= F_\alpha \mid v \odot h \mid \nabla \diamond F_\alpha \]

subject to the following additional restrictions:

1. If \( A \ast A \) term contains the form \( \epsilon_{ijk} \ast \epsilon'_{i'j'k'} \) then the indices \( ijk \) must be disjoint from \( i'j'k' \). Otherwise, if \( i = i' \) then \( \epsilon_{ijk} \ast \epsilon'_{i'j'k'} \) is rewritten to \( \delta_{jj'} \delta_{kk'} - \delta_{jk} \delta_{kj'} \).

2. If \( A \ast A \) term contains the form \( \epsilon_{ijk} \ast A \) and \( A \) has a differentiation component then no two of the indices \( ijk \) may occur in the differentiation component of \( A \). For example, \( \nabla_{jk} \) has indices \( jk \) and so \( \epsilon_{ijk} \ast \nabla_{jk} \) is not in normal form since \( \epsilon_{ijk} \ast \nabla_{jk} \implies 0 \).

3. If \( A \ast A \) term contains the form \( \delta_{ij} \ast A \) then \( j \) may not occur in \( A \). For example, the term \( T_j \) does have index \( j \), expression \( \delta_{ij} \ast T_j \) is not in normal form, and \( \delta_{ij} \ast T_j \) will be rewritten to \( T_i \).

4. If \( A \ast A \) term contains the form \( \sqrt{e_1} \ast \sqrt{e_2} \) then \( e_1 \neq e_2 \).

5. If \( \sum(N) \) term is of the form \( \sum(e_1 \ast e_2) \) then \( e_1 \) is not a scalar \( s \), scalar field \( \varphi \), or constant \( c \). For example, terms \( \sum(s \ast e_2) \) or \( \sum(\varphi \ast e_2) \) are not in normal form and will be rewritten as \( s \sum e_2 \) and \( \varphi \sum e_2 \), respectively.

### 4.3.2 Termination

The last property we demonstrate is that the normalization rewrites will terminate and that the resulting term will be in normal form. Our approach uses the standard technique
of defining a well-founded size size metric $S(e)$ on EIN expressions and showing that the rewrite rules always decrease the size of an expression.

**Size Metric** We define a size metric $S : E \to \mathbb{N}$ inductively on the structure of the grammar in Figure 2.1.

<table>
<thead>
<tr>
<th>EIN expression</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e, T_\alpha, F_\alpha, (v_\beta \otimes h^\mu), \delta_{ij}$</td>
<td>1</td>
</tr>
<tr>
<td>$\mathcal{E}_\alpha$</td>
<td>4</td>
</tr>
<tr>
<td>$\text{lift}(e), \sqrt{e}, -e, \exp(e), e^n, \kappa(e)$</td>
<td>$1 + Se$</td>
</tr>
<tr>
<td>$e_1 + e_2, e_1 - e_2, e_1 * e_2$</td>
<td>$1 + Se_1 + Se_2$</td>
</tr>
<tr>
<td>$\frac{e}{b}$</td>
<td>$2 + Se_1 + Se_2$</td>
</tr>
<tr>
<td>$\Sigma e$</td>
<td>$2 + 2Se$</td>
</tr>
<tr>
<td>$\nabla_\nu \circ e$</td>
<td>$5SeSe$</td>
</tr>
<tr>
<td>$e(x)$</td>
<td>$2Se$</td>
</tr>
</tbody>
</table>

**Lemma 4.3.1.** If $e \Longrightarrow e'$ then $S(e) > S(e') \geq 0$

*Proof.* Given a derivation $d$ of the form $e \longrightarrow e'$ we state $P(d)$ as a shorthand for the claim that the derivation reduces the size of the expression $e$. By case analysis and comparing the size metric provided below are key cases of the proof (Section A.3.1).

**Requires case analysis on the structure of the rewrite rule:**

The rewrite rule (R1) has the form $(e_1 \odot_n e_2)@x \Longrightarrow (e_1@x) \odot_n (e_2@x)$.

Case analysis by specifying operator $\odot_n$

$\odot_n = *$

Since $S(e_1 * e_2)@x = 2 + 2Se_1 + 2Se_2$ and $S(e_1@x) * (e_2@x) = 1 + 2Se_1 + 2Se_2$

then $S(e_1 * e_2)@x > S(e_1@x) * (e_2@x)$

$\odot_n = \bigoplus$

Since $S(e_1 \bigoplus e_2)@x = 4 + 2Se_1 + 2Se_2$ and $Se_1 \bigoplus e_2@x = 2 + 2Se_1 + 2Se_2$

then $S(e_1 \bigoplus e_2)@x > S(e_1 \bigoplus e_2)@x$
P(d)

**Uses size comparison lemma A.3.3:**

The rewrite rule (R9) has the form $\nabla_i \diamond (\cosine(e_1)) \Rightarrow (-\sin(e_1)) * (\nabla_i \diamond e_1)$.

Since $S(\nabla_i \diamond (\cosine(e_1)))=(1 + Se_1)5(1+Se_1)$
and $S((-\sin(e_1)) * (\nabla_i \diamond e_1))=Se_1 * (1 + 5Se_1) + 3$
then $S(\nabla_i \diamond (\cosine(e_1))) > S((-\sin(e_1)) * (\nabla_i \diamond e_1))$ by (Lm A.3.3)

P(d)

**Uses size comparison lemma A.3.2:**

The rewrite rule (R17) has the form $\nabla_i(e_1 \odot e_2) \Rightarrow (\nabla_i e_1) \odot (\nabla_i e_2)$.

Since $S(\nabla_i(e_1 \odot e_2))=(1 + Se_1 + Se_2)5(1+Se_1+Se_2)$
and $S((\nabla_i e_1) \odot (\nabla_i e_2))=Se_15(Se_1) + Se_25(Se_2) + 1$
then $S(\nabla_i(e_1 \odot e_2)) > S((\nabla_i e_1) \odot (\nabla_i e_2))$ by (Lm A.3.2)

P(d)

**Simple Example:**

The rewrite rule (R27) has the form $\frac{e_1 \odot e_2}{e_3} \Rightarrow \frac{e_1}{e_2 \odot e_3}$.

Since $S(\frac{e_1 \odot e_2}{e_3})=4 + Se_1 + Se_2 + Se_3$ and $S(\frac{e_1}{e_2 \odot e_3})=3 + Se_1 + Se_2 + Se_3$
then $S(\frac{e_1 \odot e_2}{e_3}) > S(\frac{e_1}{e_2 \odot e_3})$

P(d)

Thus, the well foundedness of the size metric guarantees that the normalization process terminates. We also want to guarantee that normalization actually produces a normal-form. We define a subset of the EIN expressions that are in normal form by a grammar Section 4.3.1. We then define the terminal expressions as $T = \{ e \mid \not\exists e' \text{ such that } e \Rightarrow e' \}$.

The following two lemmas relate the set of normal forms expressions to the terminal expressions. The first shows that termination implies normal form.
Lemma 4.3.2. If $e \in \mathcal{T}$, then $e \in \mathcal{N}$

Proof. The proof is by examination of the syntax in Figure 2.1. For any syntactic construct, we show that either the term is in normal form, or there is a rewrite rule that applies. We state $Q(x)$ as a shorthand for the claim that if $x$ has terminated and is normal form. Additionally we state $CQ(x)$ if there exists an expression that is not in normal form and has terminated. The following is a sample of a proof by contradiction (full proof is available Section A.3.2).

\[
\begin{align*}
\text{case on structure } x \\
\text{\hspace{1cm} } x=c & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{1cm} } x=T_\alpha & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{1cm} } x=F_\alpha & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{1cm} } x=V_\alpha \otimes H & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{1cm} } x=\delta_{ij} & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{1cm} } x=\mathcal{E}_\alpha & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{1cm} } x=\text{lift}(e_1) & \\
\text{\hspace{1cm} } \text{Prove } Q(x) \text{ by contradiction.} \\
\text{\hspace{2cm} Assume } CQ(x) \\
\text{\hspace{2cm} } c & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{2cm} } T_\alpha & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{2cm} } F_\alpha & \quad \text{Incorrect Type} & Q(x) \\
\text{\hspace{2cm} } e \otimes e & \quad \text{Incorrect Type} & Q(x) \\
\text{\hspace{2cm} } \delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk} & \quad \text{Normal Form} & Q(x) \\
\text{\hspace{2cm} } \text{lift}(e_1) & \quad \text{Incorrect Type} & Q(x) \\
\text{\hspace{2cm} } M(e) & \quad \text{Assume } Q(e) \text{ then } Q(x) \\
\text{\hspace{2cm} } \text{Given } M(e_1) = \sqrt{e_1} | \exp(e_1) | e_1^T | \kappa(e_1) \\
\text{\hspace{2cm} } -e & \quad \text{Assume } Q(e) \text{ then } Q(x) \\
\text{\hspace{2cm} } \frac{\partial}{\partial x_\alpha} \circ e & \quad \text{Incorrect Type} & Q(x)
\end{align*}
\]
The next lemma demonstrates that normal form implies termination.

**Lemma 4.3.3.** If $e \in \mathcal{N}$, then $e \in \mathcal{T}$

*Proof.* We state $M(e)$ as a shorthand for the claim that if $e$ is in normal form then it has terminated. The following is a proof by contradiction. $CM(e)$: There exists an expression $e$ that has not terminated and is in normal form. More precisely, given a derivation $d$ of the form $e \rightarrow e'$, there exists an expression that is the source term $e$ of derivation $d$ therefore not-terminated, and is in normal form. Below are cases of the proof (Section A.3.3).

**R1.** $(e_1 \circ_n e_2)@x \Longrightarrow (e_1@x) \circ_n (e_2@x)$

Let $y = (e_1 \circ_n e_2)@x$ and since $y$ is not in normal form then $M(\ R1)$ OK

**R2.** $(e_0 \circ_2 e_1)@x \Longrightarrow (e_0@x) \circ_2 (e_1@x)$

Let $y = (e_0 \circ_2 e_1)@x$ and since $y$ is not in normal form then $M(\ R2)$ OK

*□*

**Theorem 4.3.4 (Normalization).** For any closed EIN expression $e$ the following two properties hold:

1. there exists an EIN expression $e' \in \mathcal{N}$, such that $e \implies^* e'$, and

2. there is no infinite sequence of rewrites starting with $e$.

In other words, for any expression $e$ we can apply rewrites until termination, at which point we will have reached a normal form expression $e'$. 

93
Proof. The theorem follows from Lemmas 4.3.1, 4.3.2, and 4.3.3 described in Section A.3.

4.4 Discussion

The properties that we have described demonstrate the correctness of the normalization transformations for EIN. Unfortunately, the rewriting system is not confluent (because different pairings of $E$ can be rewritten and produce different normal forms), in our system we apply rules in a standard order, but there may be opportunities for improving performance by tuning the order of rewrites.

While there are still many opportunities for compiler bugs, normalization is the most critical part of compiling tensor-field expressions down to executable code, so these results increase our confidence in the correctness of the compiler. There are other parts of the compiler pipeline for which we hope to prove correctness in the future.
CHAPTER 5

AUTOMATIC TESTING MODEL

Testing a compiler for a high-level mathematical programming-language poses a number of challenges not found in previous work on testing compilers. While it is easy to write down complicated mathematical expressions to feed to the compiler, it is difficult to predict what the correct answer should be. For this reason, manual construction of tests for the Diderot compiler was time consuming and prone to biases (i.e., combinations of operations that were easy for the test author to understand). Furthermore, Diderot is a rich language with many operators, so the space of possible combinations is too large for manual exploration. Thus, as in previous work, it is vital that we build a testing tool that can automatically generate test cases that provide good coverage of the features of the language [49, 20].

There is extensive research in compiler testing. Differential testing relies on comparing various versions and implementations of the compiler [53, 71]. Equivalence Modulo Inputs [47] creates a family of programs that can be used for differential testing. Alone, these approaches did not seem sufficient in the case of Diderot. Most of the substantial transformations that occur during compilation are necessary and not reasonable to disable. Additionally earlier versions of the compiler are much less expressive than the current implementation. Our goal is to validate the correct answer for the new operations in the Diderot language. We chose to develop a ground truth for our test cases by careful construction of data sets and careful choices of points of evaluation.

In this chapter, we present Diderot’s automated testing model, $DATm$ [15]. It is designed to rigorously test the core mathematical parts of the Diderot implementation. $DATm$ combines generation of test programs with generation of synthetic data for which the correct values and properties of Diderot program output is known. For each test program, synthetic data is used to synthesize tensors and tensor fields that are being used in the Diderot program. The solution can then be derived analytically as an operation on polynomials. The generated Diderot program is compiled and run on the test data. $DATm$ compares the result
of the Diderot program with the analytically derived correct answer and if the answers are within an error tolerance, the test passes.

We also demonstrate how the testing model can be extended to automatically create and test different Diderot programs. We apply a metamorphic testing technique to do visual verification on an important class of algorithms. Metamorphic testing is used to evaluate unknown solutions based on some property.

\textit{DATm} can offer a full coverage of a set of operators. Our testing coverage includes common computations that the user is expected to use and uncommon ones that a compiler writer is not likely to test. It has found various bugs in the Diderot compiler and has enabled quick debugging of new operators. It is designed to aid development by supporting quick reproducibility of test cases, providing exhaustive testing (which is especially useful for new operators), and random testing (which necessary for searching the space of more complicated programs). It has provided other unexpected benefits, such as identifying mathematically valid programs that were unnecessarily rejected by the compiler because of artificial limits in the typing rules.

The remainder of the chapter is organized as follows. We first introduce Diderot’s Automated Testing model and describe its implementation in Section 5.1. In Section 5.2 we describe how the model can be extended to automatically test a class of visualization algorithms. Section 5.3 presents the results about both the effectiveness of \textit{DATm} in finding bugs and about \textit{DATm}'s efficiency in generating and running tests Section 5.3. Lastly, we discuss the contributions of the work in Section 5.4.

\section{5.1 Diderot’s Automatic Testing model}

This chapter describes the technique we used to test the new operators added to the Diderot language by introducing two models (Figure 5.2): \textit{DATm} a model that evaluates based on equality and \textit{DAVm} a model that evaluates based on symmetry. We will give an overview of \textit{DATm} in the following section and then provide further details.
5.1.1 Overview

This section will introduce DATm as illustrated in Figure 5.2a and relate it to common terminology. The test requirements are described in the frame and serve as input to the test generator. The testing frame defines several key factors for DATm such as what is being tested and how to search for test cases. The generator creates a test description for each test case. The test description describes an application of operators to arguments with different Diderot types. The test description can also be considered the test input.

The testing process involves creating synthetic data and a test program given the test description. Synthetic data is created for each argument and is represented with a data file and symbolic expression. A Diderot program is generated by using the corresponding synthetic data and list of points. The output of the executed Diderot program is a data file. The analytically derived solution is found by evaluating the symbolic expression at the same series of points as the Diderot test program.

The test output is a nrrd file created by Diderot and a Sympy expression. We expect that both output reduce to a series of numbers. We want to ensure that the Diderot program compiled and we can compare the output. The expected behavior is numerical equality between the output.

5.1.2 Generating test cases

Automatically building test cases is one of the goals of DATm (Figure 5.1). Test cases are based on a subset of the Diderot language. Types $\tau$ includes reals, vectors, matrices, third-order tensors, scalar fields, vector fields, and n-by-n second order tensors fields. Operations include a mix of unary and binary operators applied to tensors and tensor fields. The core of a test case is an application of an operator to expressions. An exhaustive generation of test cases can be found by iterating over the various types and operators in the scope,
\( \tau ::= \text{tensor}[\varsigma] \) tensor with shape \( \varsigma \)

\( \text{field}\#k(d)[\kappa] \) tensor field with continuity \( k \), shape \( \kappa \),
and dimension \( d \) (\( 1 \leq d \leq 3 \))

\( \text{image}(d)[\kappa] \) image data with
shape \( \kappa \), and dimension \( d \) (\( 1 \leq d \leq 3 \))

\( \gamma ::= \tau | \text{field}\#k(d)[\varsigma] \) broader range of shape

\( d_n ::= 2 | 3 \) dimension

\( v_n ::= 2 | 3 | 4 \) extended dimension

\( \varsigma ::= \text{nil'} | v_1, \cdots, v_n \) \( n \leq 3 \) Tensor shape

\( \kappa ::= \text{nil'} | v_1 | d_1, d_1 \) Field shape

operators ::= \( - \), \( \| \cdot \| \), \( @ \), \( \sqrt{\ } \), \( \nabla \), \( \nabla \otimes \), \( \nabla \times \), \( \nabla \bullet \), inverse,

| \( \text{normalize} \), trace, transpose, det,
| \( \sin \), \( \cos \), \( \tan \), \( \arccos \), \( \arcsin \),
| \( [:,0] \), \( [1] \), \( [:,1,:] \), \( [1,0,:] \), \( \cdots \)
| \( + \), \( - \), \( \ast \), \( / \), \( \cdot \), \( : \), \( \times \), \( \otimes \), \( \odot \), | modulate

kernel ::= c4hexic, tent, ctmr, bspln3

Figure 5.1: Subset of Diderot types and operators that can be tested with \( DAT_m \)
then internally type-checking it to filter out the applications that do not make mathematical sense. Test generation is parameterized by the testing frame which is explained in the next section.

5.1.3 Testing Frame

The testing frame is defined by the settings and scope. The settings indicate how to initialize various variables used in testing. The scope describes the subset of types and operators that are being tested.

Settings

The settings initialize the following:

- **Data Creation factors** when creating synthetic data. Synthetic data creation is defined by the order of coefficients linear ($x$), quadratic ($x^2$), and cubic ($x^3$), and the quantity and orientation of samples taken (by randomizing shear and angle). Using fewer samples and creating a non-isotropic grid creates a stronger field reconstruction (a core operation for Diderot) test.

- **Test program details** and restrictions to the scope of test cases. Generated test programs probe a tensor field at a set number of positions and use a specified reconstruction kernel (Figure 5.1). The requirements of the test cases include the number of operators (1-3), and limitations to argument types (such as only tensor fields).

- **Random or exhaustive exploration** of test cases. It is not always practical to run an exhaustive test and create tens of thousands of programs. The developer specifies the probability of a single test program being executed in the testing frame. Random testing does not ensure coverage, but it makes it feasible to explore a larger set of complicated programs (with a varying number of nested operators) in a more manageable amount of time. Experiments with random testing are presented in Section 5.3.2.
Scope

The scope is the set of possible programs that can be tested. It is possible to specify a single test case or target a subset of test cases with command line arguments. The ability to indicate the scope can aid the process of debugging and enable more targeted testing. The scope of the testing falls into three modes:

1. Run all possible test cases described under the testing frame. Every possible combination of operators and arguments types supported by DATm can be created in the exhaustive setting. In the random mode, the test generator will randomly choose the test cases to create.

2. Targeting a group allows us to limit the testing scope to some argument(s). Targeting can be helpful when testing a new operator added to the language by only creating tests that involve that operator.

3. Run a single case, i.e. the addition of two 2-d scalar fields.

5.1.4 Generate Diderot program

DATm supports a scripted instantiation of test program templates for each test program. Type description of arguments $\tau$ are converted to Diderot types. Tensor field values are initialized by loading a nrrd file that was created in the previous step.

\[
\text{field}\#_k(2)[2]F = \text{load}(\text{"F.nrrd"}) \odot c4\text{hexic} ; \\
\text{tensor}[2]T = [\exp T_1, \exp T_2] ; 
\]

Each operator is defined by attributes that allow its scripted printing in Diderot programs. The outer product operator :

\[
\text{op}_{outer} = \{\text{out} : \odot, \text{placement} : \text{middle}, \text{arity} : 2, \text{limit} : \text{None}\} 
\]

indicates that the unicode symbol $\odot$ is placed in the middle of two arguments.
\textbf{field} \#k(2)[2,2]G = \text{inv} (F \otimes T);

The field is probed at multiple points in the field domain. An inside test is imposed to be sure the position is in the field domain. The inside test is necessary when the field parameters such as angle vary

\textbf{if} (\textbf{inside}(F, \text{pos}))\{\ldots \}

The result of computation is the observed value.

\textbf{tensor}[2,2] \text{observed} = G(\text{pos});

Once the Diderot program is written, it is compiled and executed. The resulting nrrd file is converted to a text file and read as observed data.

\textit{5.1.5 Data Creation}

\textit{DATm} offers a scripted generation of synthetic data used in the testing process (test.nrrd and \textit{Sympy expression} on left in Figure 5.2a). The tensor and tensor field arguments in our test cases can be represented by a polynomial expression. Random numbers serve as coefficients to a linear, quadratic, or cubic polynomial expressions. For each argument in a test case we create a representation for Diderot and Python.

\textbf{Nrrd Format}

In a Diderot program, a tensor field is instantiated with a data file (in a nrrd format [68]) and a reconstruction kernel. The file maintains sampling orientation and discrete data points. A Diderot template can be synthesized with arguments that represent a polynomial expression. The template takes samples from the function created from the polynomial expression and saves it to a Diderot nrrd file (test.nrrd on left in Figure 5.2a). The number of samples and their orientation are based on parameters in the testing frame.
Sympy Expression

The operators we are testing are tensor calculus-based operators. Since our operators are based on math we have the opportunity to use Python packages based on math to analytically derive the correct solution. In DATm, we use the sympy package in Python [65]. A symbolic expression can be created from a list of coefficients, and handled easily.

\[ \exp_1 = x^2 + 3x + 4 \text{ and } \exp_2 = 7x - 1 \]

It can be differentiated.

\[ \exp_3 = \nabla_x \exp_1 = 2x + 3 \]

It can be manipulated with a series of tensor operators on and between them

\[ \exp_4 = \exp_3 + \exp_2 = 9x + 2 \]

It can be evaluated at points.

\[ \exp_4(x = 1) = 11 \]

In practice, symbolic expressions are created for each argument in a test case.

5.1.6 Evaluation

The same coefficients that are used to synthesize a Diderot tensor field are also used to create a Sympy expression. The operators that are applied in the generated Diderot test program are also be applied between the Sympy expressions. A tensor field in a Diderot program is probed and a Sympy expression is evaluated at the same series of points The expected test output is then the result of running the Diderot program and the evaluated Sympy expression.

We expect to be able to compare the test output based on equality. If the output is
Figure 5.2: Pipeline of Diderot’s Automated Testing model referred to as (a) $DATm$ or (b) $DAVm$. They evaluate based on numerical accuracy and equality or visual verification and symmetry. Highlighted are the different portions to the pipeline.
within some error tolerance then the test passes. There are three different possible failure modes. The test program can experience a type error “T”, a compilation error “C” or a numerical error “N”. A type error indicates an issue with the Diderot type checker. A type error can occur because the Diderot language did not meet the level of expressivity that was expected. A compilation error could be caused by a mistake in a rewriting step that halted compilation of the program. A numerical error indicates that the test program did compile and execute, but the Diderot output is not comparable to the analytically derived result.

5.1.7 Numerical Instability

In the past, testing tools have evaluated tests by comparing the compilers with older/other versions of the compiler [71], or ask the human user to supply a criterion that could be checked automatically [20]. We choose to instead evaluate based on a ground truth. Still, the evaluation is comparing the output of floating point arithmetic done by the Diderot compiler. The potential rounding errors that can occur from floating point arithmetic are well-known and experienced [24]. In DATm it is possible that the numerical instability of floating point arithmetic results in a false positive.

DATm does take some precautions against doing operations with undefined results. Each operation is tagged with a condition attribute. For instance, the square root operation $\sqrt{e}$ is tagged by a condition that limits the argument to that operation to only positive numbers. When generating a Diderot program DATm generates an if statement to check the conditions of each operation.

```plaintext
if (e > 0) { observed = G(pos); }
```

A field computation is probed a number of times, but if the conditions for the operation are not met then we are not confident in the test and the test is considered invalid. Returning to the running examples an inverse operation involves dividing by the determinant. DATm generates a test on that condition
\[
\text{if} (|\det(F \otimes T)| > \epsilon) \{ \text{observed} = G(\text{pos}); \}
\]

5.1.8 Application details

Adding new operators

Once a new tensor operator is added to Diderot, it can be added to DATm with moderate ease. The process of adding a new operator to DATm compromise of three steps:

- Define operator with arity and output type. Also, initiate attributes placement and limitations, which facilitate the scripting process that creates Diderot test programs.
- Add case to internal type checker
- Add case to apply operator to the polynomial expression(s)

The new operator can then be tested exhaustively with different argument types and in combination of existing operators. Letting the slight cost of adding a new operator to DATm well worth the return.

Uncommon programs

DATm can offer more extensive testing than is expected to be found by a Diderot developer or user. The Diderot programmer might use a limited set of computations that make mathematical sense in a context of a visualization program. In the past the testing and development of Diderot relied on hand-written programs centered on these commonly used combination of computations and arguments. Testing Diderot by developer alone biased the discovery of bugs. The testing framework expands the search for bugs from what was convenient and immediately useful to write to what is possible to write in the Diderot language.

It is worth noting that there are other uncommon field types, that can be exposed to some amount of testing (\(\gamma\) in Figure 5.1). Intermediate steps in the testing process can apply a single operator \(\text{op}_1(\tau_1, \tau_2) \rightarrow \tau_3\) where the result type is outside of our scope (\(i.e.,\))
Figure 5.3: Volume rendering of a symmetric 3-D field using Summation projection (left) and Maximal projection (right) with positions that are sampled to evaluate correctness. We sampled 30 groups of 4 points that were equal distance from the center. Highlighted in red is one of the groups.

\[ \tau_3 \notin \tau, \tau_3 \in \gamma \). Application of a second operator (i.e., \( op_2(op_1(\tau_1, \tau_2)) \rightarrow op_2(\tau_3) \rightarrow \ldots \) tests a more extensive range of operations than can be created directly from templates.

As an example, consider the computation

\[
\begin{align*}
\text{tensor} [3] t; \\
\text{field} #k(1)[2,2] F; \\
\text{field} #k(1)[3,2,2] G = -(t \otimes F);
\end{align*}
\]

Synthetic data can be created for \( t \) and \( F \), but currently not for the \( G \)’s rarely used type \( \text{field} #k(1)[3,2,2] \). Teem [68] could not create a nrrd file of that type. In the process of computing \( t \otimes F \), the compiler creates and tests this unusual field type.

## 5.2 Visualization Verification

This section demonstrates a modest way to do automated visual verification of the Diderot language by using metamorphic testing. Programs written for scientific visualization or image analysis can be more mathematically complicated than what we have tested. It is possible that the complexity could bring to light more bugs. Evaluating results of a visualization program based on the numerical accuracy can be difficult (if not impossible), because sometimes we can’t easily know what the algebraic solution should be. Therefore, each visualization program had required an “eyeball test” in addition to already discussed drawbacks of writing tests by hand. That is, just visually looking at the resulting image. In an effort to include more types of testing, we created Diderot’s Automated Visualization
model DAVm and use a known property to evaluate an unknown result.

### 5.2.1 Concept

To test Diderot, we need to construct a visualization program that can be checked in an automated way. We choose to compute a simple volume renderings of synthetic 3-D fields created by the new operators. Projecting a rotationally 3-D symmetric field, restricted to a spherical domain, should produce a rotationally symmetric 2-D image, regardless of the viewpoint and the field operations involved. DAVm generates Diderot programs to (1) do a volume rendering and (2) a sampling of the output of the volume rendering. Figure 5.3 provides an example of the output from these programs. In the following we describe aspects to DAVm that depend on the visualization concept.

### 5.2.2 Pipeline

DAVm, an extension of DATm, uses much of the same basic code. It was created to generate meaningful tests while keeping the benefits of automated and random testing. Figure 5.2b provides the pipeline for DAVm. DAVm, like DATm, is used to generate test cases, create synthetic data, write Diderot programs, and evaluate each test program. In DAVm, the evaluation of those programs do not use sympy and instead is grounded in symmetry checks.

Generate Test cases

A Diderot program is used as a template for creating test programs. We use a MIP program [40] as the base code for a basic volume rendering program in Diderot. As previously discussed tensor fields are created by the convolution between an image file and a kernel.

```plaintext
field #k(3)[~]F0 = c4hexic ⊗ image("f0.nrrd");
field #k(3)[~]F1 = c4hexic ⊗ image("f1.nrrd");
```

We modify the template to represent the computation for each test cases. The field of interest
is not restricted to a 3-D scalar field created from source files (such as F0 or F1) but some field created as a result of operators between them

\[
\text{field } \#3(3)[]G = (-F0) \ast F1;
\]

the core of the program gets the maximum value along the ray and stores it into the output variable out.

\[
\text{out} = \max(\text{out}, G(\text{pos}));
\]

The example demonstrated how to generate test programs by using a maximal intensity volume rendering template. Another template can be used to do a summation projection volume rendering by using:

\[
\text{out} += \text{out} ;
\]

Figure 5.3 presents images from these programs. The test generator is used to automatically create test cases but has an extra restriction; the result of the computation being tested needs to be a 3-D scalar field, the type of arguments is restricted to 3-D fields, and the data generated to synthesize tensors and tensor fields is symmetrical.

Evaluation

The correct numerical answers for these visualization programs are unknown (the projections would involve potentially unwieldy symbolic integrations) but we can use symmetry arguments to detect major bugs. We test the symmetry of the output by sampling the field at groups of points, at the same distance from the center of the image, and comparing them. Each point in a group is at the same distance from the center. For each point, the x-coordinate is chosen randomly, and the corresponding y-coordinate is chosen using the distance formula. The points are chosen at meaningful positions while maintaining some notion of randomization. We generated a few hundred programs and used an eyeball test to establish a baseline so we can determine when the difference between the value of points is
5.3 Results and Performance for \textit{DATm}

In this section, we present four sets of results from using \textit{DATm}. The first set is from an experiment that measures the impact from changing the type of search (exhaustive or random). \textit{DATm} can be used to create a vast number of programs automatically and do faster regression testing. The second set is from an experiment that applies \textit{DATm} at different snapshots of the compiler. This experiment demonstrates that a lot of bugs were not being caught until we developed \textit{DATm}. The third set is a description of the bugs that were found in the compiler. We provide examples of bugs that would have been especially difficult to find without \textit{DATm}. The fourth set is from a simple experiment that runs \textit{DAVm}. It demonstrates that it is possible to do other types of tests with an extension of the testing model.

5.3.1 Experimental Framework

The experiments were run on an Apple iMac with a 2.7 GHz Intel core i5 processor, 8GB memory, and OS X Yosemite (10.10.5) operating system. The experiments may run different sets of tests by changing some settings in the testing frame, but the following factors in the testing frame are constant. To create synthetic data, quadratic coefficients are used. To create nrrd files for a tensor fields 70 samples are taken and the sampling orientation is not randomized. The generated test program used c4hexic kernels to reconstruct tensor fields and 7 positions to probe the fields. We omit times for doing exhaustive testing with a high number of nested operators in favor of doing random testing in a large search space.
5.3.2 Exhaustive vs. Random Testing

Our experiment demonstrates the ability to do exhaustive and random testing with DATm. It executes the model by initializing the testing frame with different settings. The changes vary the number of test programs that are created and how they are found. We report the impact of changing these variables by recording how long testing takes and number of test cases available.

Our experiment does a single exhaustive search and several random searches for up to 3 nested operators. An exhaustive search attempts all possible test programs. A random search has a certain probability to execute each test program. Table 5.4 records the timing measurement from doing these different searches and varying the number of nested operators. The measurements range from seconds to hours. Table 5.4 records the number of test programs that are created with an exhaustive search. A single operator creates hundreds of programs, two nested operators creates tens of thousands, and three nested operator creates hundreds of thousands of test programs.

The experiment demonstrates that DATm can create a vast number of test programs. By creating a large range of test cases it is possible to find hidden and unique bugs. DATm can create thousands of programs, but it is not feasible to do exhaustive search each time there is a change to the compiler. To enable quicker regression testing it is necessary to also do random testing. The developer can choose to parametrize DATm to apply exhaustive testing or random testing to specified operators.

5.3.3 Progress

To evaluate the effectiveness of DATm we ran the same set of programs on six different snapshots of the compiler. The snapshots of the compiler where taken off the Diderot repository over four months apart, starting from March 2015. To be clear, we did a post evaluation of the state of the compiler using DATm. The testing frame was set to the
Timing to run DATm with given probability.

<table>
<thead>
<tr>
<th>No.of Operators</th>
<th>No. of Programs</th>
<th>0 %</th>
<th>0.5 %</th>
<th>1%</th>
<th>5%</th>
<th>10%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>695</td>
<td>0.25</td>
<td>0.3</td>
<td>0.55</td>
<td>2.6</td>
<td>4.65</td>
<td>32.53</td>
</tr>
<tr>
<td>2</td>
<td>18,819</td>
<td>7.62</td>
<td>14.43</td>
<td>18.35</td>
<td>64.2</td>
<td>121.23</td>
<td>1099.16</td>
</tr>
<tr>
<td>3</td>
<td>495,626</td>
<td>58.83</td>
<td>246.02</td>
<td>393.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 5.4: The following offers measurements from executing DATm with different settings. The settings are (1) the number of operators, and (2) the probability to run a single test case. A 0% probability refers to iterating test cases only, a 100% is an exhaustive test, and the range in-between refer to a random search with a set probability to execute each test case. The figure records the number of test programs that are created with an exhaustive search, and the timing measurement (in minutes) for running DATm with each type of search.

The test results are organized by three different categories, “failed”, “compilation error”, or “passed”. The “failed” description means that there was a clear error because the numerical result was not correct, or there was an error when executing the program. The “compilation error” descriptions indicates that the programs that did not compile because there was an error at compile time. The compilation errors can include type errors from testing operations that were not part of the language syntax at the time or from errors elsewhere inside the compiler. The experiment measures the number of tests programs that fall into these different categories. Figure 5.5 provides the results from the experiment.

Over time, the number of passing tests is inconsistent, rather than constantly improving. It is possible that new errors may have been introduced during development and not caught. Bugs were unknown and unfixed. The test does not indicate the number of bugs and we expect that many of the tests fail due to the same compiler bug. Various compilation errors may be due to earlier versions of the language not fully supporting all the functionality. The experiment shows that development of Diderot rapidly changed once DATm was introduced. At the latest data point, Diderot did not fail any of the tests.
Figure 5.5: Results when running DATm over time. Categories “OK”, “Fail”, “Did not compile”, and “NA” indicates a program passed, failed, it did not compile, or was thrown out, because conditions set by the operators were not met, respectively. The vertical line marks when DATm was introduced.

5.3.4 Bugs

DATm has found various bugs in the Diderot compiler when it was developed. As previously discussed in Section 5.1.6, there are three different failures modes: type error (T), compile error (C), and numerical error (N). DATm found seven compile time errors, five numerical errors, and eight type errors. While there might not be many bugs, some of the compile and type errors were caused by complicated mistakes in the compiler. The numerical bugs would have been difficult to find without DATm. The type error bugs account for missing features and unexpected restrictions in the language. Figure 5.6 is a list of bugs found with DATm. The following provides a description of some of the bugs and provides motivation for extensive testing.

Unique Bugs

DATm discovered bugs that could only arise with a unique combination of operators. Bugs of this nature are unlikely to be found by a Diderot user and difficult to identify in the code alone due to its complexity. Following is a list of examples of these type of bugs. Each with a description of the bug and what makes it unique.
<table>
<thead>
<tr>
<th>Nested</th>
<th>Mode</th>
<th>Cause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>O</td>
<td>Restriction on trace</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>O</td>
<td>Restriction on transpose</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>O</td>
<td>Restriction on slice</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>O</td>
<td>Layout in using computations inline</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>E</td>
<td>Restriction on determinant</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>E</td>
<td>Restriction on modulate</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>E</td>
<td>Missing $\times$ between tensor and field</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>E</td>
<td>Generality $\otimes$ between tensor and field</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>O</td>
<td>Design in EIN IR indices</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ex. $(U \times V)[1]$</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>R</td>
<td>Rewrite Parameter Id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ex. $\nabla(T + F)$</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>R</td>
<td>Wrapping summation in rewrite rule</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>R</td>
<td>Check dimension in Optimization</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>R</td>
<td>Translating to vector code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vectorize EIN without variable index</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>R</td>
<td>optimization not generalized</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>R</td>
<td>redundant indices in a single term</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ex. $\text{trace}(\text{modulate}(-U,V))$</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>R</td>
<td>Rewrite field reconstruction indices</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ex. transpose($\nabla \otimes V$)</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>R</td>
<td>Data accessed incorrectly</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reverse order for second order F</td>
</tr>
<tr>
<td>1</td>
<td>N</td>
<td>O</td>
<td>Algebraic error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>creating EIN operator. Ex. $|\varphi|$</td>
</tr>
<tr>
<td>1</td>
<td>N</td>
<td>O</td>
<td>creating the concatenation operator</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>R</td>
<td>determinant(concat(F,G))</td>
</tr>
</tbody>
</table>

Figure 5.6: The list of bugs are categorized by the number of nested operators needed to discover the bug, the failure mood, and the cause of the bug. There are three different failures modes; type error $T$, compile error $C$, and numerical error $N$. The cause of the error could be because of missing features $E$, a transformation rewrite $R$, or other $O$
Ex1. Numerical error caused by complicated transformation  This bug was an example of a rare numerical error in the output, which was exposed by computing the transpose of the Jacobian

\[
\text{field}^k(3)[3,3] G = \text{transpose}(\nabla \otimes (V));
\]

The Diderot compiler must generate code to map derivatives from image-index space to world space. A subtle error arises in tracking the variable index that represents the shape of the tensor field. Because of the use of transpose, the index is swapped with another, which gives rise to the error. It is the unique combination of two operators (transpose of a differentiated vector field) that triggers the bug.

Ex2. Bug exposed by testing nested operators  Programs with nested operators reach deeper parts of the Diderot compiler. The computations reach parts of the compiler that are designed to optimize complicated expressions. The trace of the modulate between a negation of A and B can be computed as

\[
G = \text{trace}(\text{modulate}(-A, B));
\]

The application of the trace operator on modulate is okay. It is the use of a third operator that triggers an algorithm to be applied. The computation raised a compilation error because the case was not handled correctly by that algorithm.

Ex3. Design issue in EIN IR  This bug emphasizes the need to test every operator in combination with each other. The cross product operator \( \times \) and slice operation \([1]\) are seemingly independent operators that worked fine on their own. The cross product operation multiplies two arguments by using indices and a permutation symbol, while the slice operation slices a tensor (or field) on some axis. When they were combined in a Diderot test program:

\[
(U \times V)[1];
\]
it created a compilation error, because the necessary rewrites could not be represented. The computation exposed a design issue in the IR of the compiler.

Ex4. Parameter ids  A careless error occurs when we compute the differentiation of the subtraction of a tensor and tensor field.

\[
\begin{align*}
\text{field}^k(d)[d']F; \\
\text{tensor}[d']T; \\
\text{field}^k(d)[d']G = T - F; \\
\text{field}^k(d)[d',d]J = \nabla \otimes G;
\end{align*}
\]

The differentiation of a constant \( T \) is zero, and while the tensor expression goes to zero, the tensor argument remains. The parameter ids need to be reset, but were not done correctly.

Language expressivity

An unexpected benefit of implementing \( DATm \) was the gut-check to Diderot’s advertised expressivity. Previous work offered a type specification for the Diderot language. As an example, consider the type specification for the trace operation on fields:

\[
\text{trace} : \text{field}^k(d)[i,i] \rightarrow \text{field}^k(d)[]
\]

But in actuality, the implementation provided trace with the following type:

\[
\text{trace} : \text{field}^k(d)[d,d] \rightarrow \text{field}^k(d)[]
\]

The type restriction did not cause numerically incorrect results, but instead ambiguously restrained the use of the operator.

One of the design goals of Diderot is for every tensor operator to be supported on both tensors and tensor fields with full generality. \( DATm \) was able to find situations where the implementation did not meet full generality. We have found cases of missing expressivity where the types were overly constrained (as with trace) and where combinations of fields and tensors were not allowed. While these test failures are not, strictly speaking, bugs in the implementation, it is worth noting their existence, since fixing them makes Diderot a more
<table>
<thead>
<tr>
<th>No. of Operators</th>
<th>No. of Programs</th>
<th>Time given probability.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 %</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>.01</td>
</tr>
<tr>
<td>2</td>
<td>216</td>
<td>3.13</td>
</tr>
<tr>
<td>3</td>
<td>3,151</td>
<td>81.63</td>
</tr>
</tbody>
</table>

Figure 5.7: Results from running DAVm. The time is given in minutes from an exhaustive and random search.

complete language.

5.3.5 Visualization Results

In this section, we present the results of running DAVm. The experiment created test programs based on the MIP template. To sample the result, we created 30 groups of 4 points equal distance from the center. The right-most image in Figure 5.3 is an example of the output from metamorphic testing. In the image, the points sampled are imposed on the volume rendering of the test program. The experiment measures the the number of programs and time it took to test those programs. The results are in Figure 5.7.

DAVm is a prototype to illustrate DATm can be used for other types of testing. DAVm did not find any errors in Diderot, but it was created after Diderot was tested extensively with DATm. Testing with DAVm offers a few drawbacks. The testing concept only applies to 3-d scalar fields and that restricts the type of test cases that can be generated. Additionally it takes longer to execute a visualization test program. As a result, DAVm does not create nearly as many tests as DATm. The experiment does demonstrate that it is possible to extend DATm to do automated testing with visual verification.

5.4 Discussion

Automation DATm offers many benefits to testing Diderot. Testing offers more extensive coverage for a set of operators than was otherwise available. It creates thousands of unique programs. A magnitude unmanageable to write by hand. This includes common ones a
developer might quickly write to test the language and unlikely ones that still deserve and require examination.

**Debugging**  *DATm* is designed to find bugs in the Diderot compiler, but it can also aid in the process of debugging by tracking failed tests and allowing reproducibility. It is possible to make copies of every program that fails, but this could easily eat up computer space. In lieu of that, we use labels to target/limit the testing scope. The labeling aids the developer in the process of debugging by identifying tests that fail and enables the developer to test them again after making changes to the compiler.

**Hidden Bugs**  *DATm* has successfully found bugs in the Diderot language. The sources of error range from type checking to being entangled in complicated transformations. Some of these bugs were more difficult to find because they only arise with a unique combination of operators (Section 5.3.4). The unique nature to these bugs makes them harder for a developer to find and automated exhaustive testing more essential.

**Types of Test Programs** An earlier section introduced a way to do visual verification on programs with unknown algebraic solution. Our application of *DAVm* demonstrated how volume rendering templates can be used to generate test programs. The templates can be slightly altered to do a different computations as long as the operations involved maintain symmetry. We believe that it is possible to build other types of visualization test programs by using templates.

**Evaluation**  We have presented *DATm*, an automated testing tool for a high level language. *DATm* provides a practical way to test Diderot by creating thousands of tests and evaluates the results based on a ground truth. We have described the details in implementing *DATm* as well as an extension to create other types of tests. We have provided examples of bugs *DATm* was able to find and have made the language more reliable and correct. There are
other types of improvements we can make towards the testing process, such as including other types of tests strategies (see future work Section 9.1.1).
CHAPTER 6
EXTENSION

Computational scientists compute solutions to systems of partial differential equations (PDEs) on large finite meshes using numerical techniques, such as the finite element method (FEM). They can be used to describe complex phenomena like turbulent fluid flow. The Diderot language does not know how to solve PDEs, or represent FEM fields, but it does have the syntax to visualize fields. Our goal is to connect Diderot to the world of finite element mesh. The work in this chapter takes a step towards using Diderot to visualize FEM fields. With the work in this chapter we hope to use Diderot to help debug visualizations of FEM fields and enable more interesting visualizations.

Solving PDES, and visualizing PDEs require two different techniques and entirely different code. To simplify the process, scientists may turn to standard visualization tools to analyze their images. The problem with that, is that there is not a universal solution to accurately visualize every PDE. For one, visualizing finite element data created with higher order elements and a small number of cells can lead to images that do not accurately represent the original solution. We offer Diderot as tool to use in these instances.

We want the user to be able to use visualization programs enabled by Diderot on fields created by FEM. Expecting users to transition from visualization toolkits to writing in a new programming languages and developing an expertise in scientific visualization is a big ask. A background in scientific visualization should not limit someone from using the visualization algorithms that can be written in Diderot. Our goal is to be able to augment any existing Diderot program (written for discrete data) and apply it to FEM data with minimal changes to the program. That way Diderot could compile programs to extract interesting visualization features from FEM data (Section 7.2).

In this chapter we present our results from visualizing FEM data with Diderot. We enabled communication between Diderot and an outside source. We describe our approach and present the results. Using Diderot we simply sample a FEM field and we initialize a
template to do volume rendering of a FEM field. Our work demonstrates a modest step
towards visualizing FEM data with Diderot. In addition it provides motivation for future
work.

The work in this chapter extended the Diderot language so a user can visualize fields
created by a different domain. Section 6.1 offers some background about this domain. Sec-
tion 6.2 provides a motivating example for visualizing finite element data. Section 6.3 de-
scribes the implementation details to our approach. Section 6.4 demonstrates an application
of our approach and then visually verifies this work. We end with a discussion Section 6.5.

6.1 Background

The Finite Element Method [8] gives a general framework that can provide a solution to
differential equations. FEM fields are created from a solving a PDE on a finite element
mesh, which involves discretizing the domain into small finite elements and using a set
of basis functions (derived from the mesh) to span the domain space. These fields are
approximate numeral solutions to PDEs. Some of the related work are discussed in more
detail in Chapter 8. In this section we more closely define FEM fields and software used to
create them.

FEM Fields  We concisely describe the details to creating FEM fields that are relevant to
this chapter. There are a variety of different meshes that are built-in or could be created by
outside tools. Often, we use a unit square mesh.

\[ m = \text{UnitSquareMesh}(2, 2) \]

This creates a 2x2 mesh of a square. Each smaller square is divided into two triangles. The
mesh is one of the arguments when defining a function space.

\[ V = \text{FunctionSpace}(m, "P", K) \]

"P" refers to a family of finite element spaces (others include "DP", "RT", and "BDM").
The basis functions are linear when $K=1$ and cubic when $K=3$.

A FEM field can be created from solving a PDE, but it can also be generated by interpolating an analytically defined expression. In the following chapter when we wish to create simple examples we choose to generate fields.

**Software**  
Software created in this area attempts to translate high-level math to computer code. Building on FEM aims to automate the discretization of differential equations, given a variational form of a PDE and provide an approximate solution. Optimizing this work is pursued by many groups including the *FEniCS Project* [26] and *Firedrake* [58].

The FEniCS [26, 50] project is an automated system to find solutions for partial differential equations using the finite element method (FEM). It enables users to employ a wide range of discretization to a variety of PDEs. On the surface it uses the Unified Form Language (UFL) [2], a domain-specific language to represent weak formulations of partial differential equations. UFL does not provide the problem solving environment, instead it creates an abstract representation that is used by form compilers to generate low-level code, like The FEniCS Form compiler (*FFC*) [3]. FFC can implement tensor reduction for finite element assembly [45] and aims to accept input from any multilinear variational form and any finite element to generate efficient code.

Firedrake [58] is a system like FEniCS that is used to solve PDEs. In addition to UFL, it uses a modified version of FFC [3] (now using TSFC), *FIAT* [44] (now using FINAT), a *PyOP2* interface [59], and *COFFEE* [51]. FFC is the FEniCS form compiler for generation of low-level C kernels from UFL forms. FIAT is the finite element automatic tabulator. It presents an abstract description of elements and has a wide range of finite element families. PyOP2 provides a framework for carrying out parallel computations on unstructured meshes. The COFFEE compiler optimizes abstract syntax trees generated by FFC.
### 6.2 Creating and Visualizing FEM data

The strategy to solve PDEs is very different from the algorithms used to visualize the result. Toolkits are commonly used to visualize the solutions created by FEM software. Firedrake uses a VTK file format for its visualization output. The format only supports linear and quadratic data. Firedrake takes the output and writes the output to a linear file format. Paraview might then accurately assume linear basis functions to represent the Firedrake output even though the original solution was created with higher-order elements. As a result the image may not accurately represent the PDE solution.

In this section we present a simple example of how a bug can occur in this circumstance. We create a field using higher-order data with a small number of cells. We describe how to create FEM data, and then we present the results of using Paraview and Diderot side by side. The implementation process that enables the communication between Diderot and Firedrake is introduced in more detail in Section 6.3.

**Creating FEM data** In the following example we use UFL to interpolate an expression over a function space $F$. To start we define a polynomial expression $x^2(1-x)$

\[
\text{exp} = \text{"x[0]*x[0]*(1-x[0])"}
\]

and a unit square mesh ($m$).

\[
m = \text{UnitSquareMesh}(2, 2)
\]

The function space $V$ is defined by a mesh ($m$), the family of finite element spaces ($P$), and the order of the polynomial ($K$).

\[
V = \text{FunctionSpace}(m, "P", K)
\]

Instead of solving a PDE, we will generate a field ($f$) by interpolating an analytically defined expression given the function space ($V$).

---

1. The images in this chapter are created when Firedrake did an $L^2$ projection, but now Firedrake uses interpolation to generate linear output.
The expression (exp) is composed with linear basis functions (when K=1) and a cubic basis function (when K=3) using a unit square mesh (m).

**Visualizing FEM data** Given a higher-order polynomial expression we can also assume that linear functions will not correctly be able to represent it. On the other hand, cubic functions should be able to offer a reasonable approximation for this problem. Additionally given expression is \( x^2(1 - x) \), we expect the maximal point to be between \( x=0 \) and \( x=1 \) at \( x=\frac{2}{3} \).

We chose to visualize the expression several different ways in order to provide a means of comparison in Figure 6.1. In the following grayscale images, the maximal points are indicated by the brighter spots. We gather the following results:

- Mathematica and Diderot, on the left, offer baseline for the correct solution. We defined a function \( F(x) = x^2(1 - x) \) where \( x \in [0,1] \). WolframAlpha can quickly and easily graph the results. In Diderot we synthesized a field by taking samples of a function defined by \( F \) and saved the result to `out.nrrd`. Then a second Diderot program was used to sample `out.nrrd` and visualize the result.
• The two left images use Firedrake data created with linear elements (K=1). Regardless of the visualization strategy we expect the images to be an inaccurate representation of the solution.

• The two images on the right use Firedrake data created with cubic elements (K=3). The image created with Paraview is incorrect \(^2\), and the image created with Diderot is correct.

When the results do not represent the field it can be difficult to understand and use visualizations to debug. From the user’s perspective the issue could be with the user’s UFL code, Firedrake’s evaluation, or the visualization program. Our goal is to provide Diderot as an alternative tool that can be used in these instances.

### 6.3 Our Approach

We worked in collaboration with the Firedrake team [58] at the Imperial College of London. Firedrake provides the field and Diderot visualizes the results. The process is enabled by communication between Diderot and Firedrake. This section explains the approach in more detail.

**FEM data** A field is created using FEM data with a Firedrake program. The field can be the result of solving a PDE or interpolating an expression over a function space. The FEM community has a different understanding of fields than the Diderot syntax currently expresses (we discuss design ideas in future work Chapter 9).

**Diderot program** The core of visualization programs is independent of the source of data, but Diderot was restricted to regular imaging grids. We addressed this limitation by allowing

---

2. As a note subdividing the field would create a more accurate solution but the context of the problem required creating a field with higher-order elements and a small number of cells.
the Diderot compiler to define a field that is defined by other kinds of data. The Diderot user could use a single line in the Diderot program to indicate the new kind of field data.

```plaintext
input fem #0(2)[g;
field #0(2)[F = toField(g)
```

As a result, a visualization program written for regular imaging grids could be easily changed so it is applied to the result of a PDE solution. We have created a few Diderot programs that have been compiled to a library.

**Firedrake program**  The Diderot program provides the framework to sample fields and do volume rendering of 3-D fields. A Firedrake program makes a call to the Diderot library. A field reference is used to initialize a call to the Diderot library.

```plaintext
res = 200 # resolution
step = 0.01 # step size
diderot.mip(file_name, f, res, res, step)
```

It initializes visualization parameters (such as resolution and step size) and provides a pointer to the field.

**Point Evaluation**  Firedrake can evaluate Functions at arbitrary physical points [32]. This feature is particularly useful for the work in this chapter. It determines which cell to look at. The change in coordinates from a reference element to one being computed involves the calculation of the jacobian matrix, its determinant, and its inverse.

Diderot does not know how to evaluate a FEM field at a given position. For each inside test, probe, and gradient operation the field has to be evaluated at a position. [32].

```plaintext
if (InsideFEM(glob->gv_m0, l_pos_4)) {
    evaluate(glob->gv_m0, l_pos_4, &l_t_6);
}
```

The output code will use Firedrake’s point evaluation functions for a given field and position.
Evaluate Visualization  We could attempt to validate images created by Diderot. According to the Principle of Representation Invariance the data layout should not change the results [43]. When possible, we can represent the data in two ways. The data is represented either as a discrete image field convoluted with a kernel or a FEM field created by Firedrake. If both fields are visualized using the same Diderot program, then we can expect that the visualization of these fields to be the same.

6.4 Demonstration

We provide two examples to demonstrate the applications of the work. In the first example, a field is created by interpolating an expression over an indicated function space and visualized with a volume rendering program. In the second example, we provide a classic example of a PDE and simply sample the result.

6.4.1 Communication and evaluating visualization

Diderot visualizes the result by using a MIP program. MIP or maximum intensity projection is a minimal volume visualization tool for 3-d scalar images.

We create a cube mesh (m) of length 2 on each side.

\[
m = \text{CubeMesh}(1, 1, 1, 2)
\]

The function space \( V \) is defined by a mesh (m), reference element (P), and uses cubic polynomials.

\[
V = \text{FunctionSpace}(m, \text{"P"}, 3)
\]

The expression creates a symmetrical sphere centered at \([1, 1, 1]\) and shifted by 0.5.

\[
\text{exp} = \text{"0.5 - ((1 - x)^2 + (1 - y)^2 + (1 - z)^2)"}
\]

Field (f) is defined by interpolating the expression given the function space (V).

\[
f = \text{Function}(V).\text{interpolate}(\text{Expression}(\text{exp}))
\]
We make a call to Diderot library by passing the field and initializing the resolution and step size.

```
res = 200 # resolution
step = 0.01 # step size
diderot.mip(file_name, f, res, res, step)
```

Diderot is used to do a volume rendering of this 3-d field by setting up a camera and doing a MIP. Diderot allows a user to define a field by some external source. Here is the augmented Diderot code for the Diderot program:

```
vec3 camAt = [1, 1, 1]; // position camera looks at
input fem #0(3)[] g;
field #0(3)[] F = toField(g);
...
if (!inSphere || |pos-camAt|< 1){out = max(out, F(pos));}
```

The Diderot program will generate code that will communicate to Firedrake’s point evaluation capability to reduce F(pos).

**Results**  We visualize the FEM field using a Diderot program. We visualized the field by varying the expression created by Firedrake, where the camera in the Diderot program is pointed, and how the data is included in the visualization program. We try these different variations to mimic how a user might use an existing Diderot program (or template) to visualize their data.

We provide the results in Figure 6.2. We can set the camera to look at the center of the data (center and right column) or off-center (left column). We also created fields with the Firedrake expression centered (two top rows) and off-centered (third row).

The red box indicates the setting where the Diderot camera and firedrake expression are both centered and results is as expected (symmetrical sphere). The other images in the figure are subject to potential bugs, because the function space defined by Firedrake does not match the image space probed by Diderot. This example illustrates the care that is needed when setting up the Firedrake Diderot pipeline.
Figure 6.2: Creates sphere \( \text{shift-}((c-x)^2 + (c-y)^2 + (c-z)^2)) \) with Firedrake. Included are examples of bugs. The centered image is correct.

**Evaluate Visualization**  The data representation, either by FEM field, or discrete field, should not change the visual representation (according to the Principle of Representation Invariance). We compare the output for the field created by Firedrake and the field created by Diderot in Figure 6.3. As can be expected, the image created by Firedrake is comparable to that of Diderot.
6.4.2 PDE Example

The Helmholtz problem is a symmetric problem and a classic example of a PDE. Consider the Helmholtz equation on a unit square $\Omega$ with boundary $\Gamma$.

$$-\nabla^2 u + u = f$$

$$\nabla u \cdot \vec{n} = 0 \text{ on } \Gamma.$$  

The solution to the equation is some function $u \in V$ for some suitable function space $V$ that satisfies both equations 3. After transforming the equation into weak form, applying a test function $V$, and integrating over the domain we get the variational problem. 4.

$$\int_{\Omega} \nabla u \cdot \nabla v + uv dx = \int_{\Omega} vf dx \quad (6.1)$$

We choose function $f$

$$f = (1.0 + 8.0 \pi^2) \cos(2\pi x) \cos(2\pi y) \quad (6.2)$$

3. The example details and code are provided by Firedrake
4. The approach to solve PDEs is described in more detail in [8]
The PDE is solved using Firedrake. We present the following code written in UFL. We omit some details and instead provide comments for clarity.

```python
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", k)
u = TrialFunction(V)
v = TestFunction(V)
f = Function(V)
f = //Interpolate the expression (6.2)
a = //Represents left-hand side of (6.1)
L = //Represents right-hand side of (6.1)
u = Function(V)
solve(a == L, u, solver_parameters={"ksp_type": 'cg'})
```

The original code uses linear elements (k=1), but we choose to use linear and cubic elements. We illustrate the results using Paraview and Diderot.

```python
# Paraview output
File("helmholtz.pvd") << u
# Call to Diderot
res=100
stepSize=0.01
type=1 # creates nrrd file
vis_diderot.basic_d2s_sample(namenrrd,u, res, res, stepSize, type)
```

The field is visualized with Paraview and with Diderot using linear (K=1) and cubic (K=3) data. The results are in Figure 6.4 along with a difference image to compare the Diderot results. The data with higher-order elements and visualized with Diderot is the most refined. There is a smoothness captured with the higher-order data that is not in the linear data.
6.5 Discussion

The Finite element community has difficulty visualizing certain fields. The communication pipeline allowed us to demonstrate Diderot’s ability to visualize FEM data by sampling the data and doing volume renderings. We have shown that Diderot could be used to correctly visualize fields created with higher order elements and a small number of cells. We have also shown a way to evaluate the results for simple test cases by comparing the result with an external data representation.

It would be beneficial if a Firedrake user could use Diderot with little hassle. We proposed that we could ease this transition by providing a Diderot template (or existing Diderot programs) that can be initialized by Firedrake code. However, as we have shown in example (Section 6.4.1) this can still lead to errors. It takes care, on behalf of the user, to be sure that a Diderot program is set-up to correctly visualize the Firedrake data.

We have successfully established communication between Firedrake and Diderot but it is just a step towards what is possible to do. Our work provides motivation to further develop Diderot and incorporate it with the FEM community. Ideas for future work are described in Chapter 9.
CHAPTER 7
APPLICATIONS

We have discussed the contribution of our work in individual chapters. We have measured the impact of applying implementation techniques inside the compiler in Section 3.3. We have reported the results from applying our testing models in Section 5.3. We have demonstrated the use of Diderot to visualize fields in another domain in Section 6.2.

In this chapter we hope to present applications of the dissertation by connecting the work. In Section 7.1 we describe how this work streamlines the process of adding new operators to the Diderot language. In Section 7.2 we describe some of the visualization programs that benefit from writing code at a high level and depend on our implementation to compile their programs.

7.1 Streamline the process of adding to the Diderot language

Our work has made it easier and faster to add it to the Diderot language. In previous chapters we have defined new EIN operators and presented bugs our testing model has found, but we have not shown the full pipeline of adding something new to the surface language. When we add something new to the language we try to leverage the expressive EIN IR (Chapter 2), the generic implementation (Chapter 3), and the robust testing model (Chapter 5).

In the following section we define a new operation we want to add to the surface language, concat. We present a new EIN operator that can represent the computation. We describe our testing process.

**Goal** Currently a user can slice tensor fields to get components of them. Consider the example

```plaintext
field#k(d)[d1,d2] A;
field#k(d)[d1,d2] B;
field#k(d)[d1] F = A[:,0];
field#k(d)[d1] G = B[:,1];
```
It should be possible to build new tensor fields on the surface level by concatenating components together, much like how tensors are concated together \([a, b]\).

\[
\text{\textbf{field}} \#k(d) [d_1, d_1] H = \text{concat} [F, G];
\]

We illustrate the structure of \(H\) below.

\[
H = \begin{bmatrix} F_0 & F_1 \\ G_0 & G_1 \end{bmatrix}
\]

**Design and implementation** We can use EIN expressions as building blocks to represent the concat computation. Perhaps each field term is represented by an EIN expression and it is enabled with a delta function

\[
H = \lambda F, G. \langle F_j \delta_{0i} + G_j \delta_{1i} \rangle_{i:2,j:2}(F,G)
\]

After normalization the new EIN operator would be

\[
H = \lambda A, B. \langle A_j \delta_{0i} + B_j \delta_{1i} + H_{1:3} \rangle_{i:3}(A,B).
\]

In the compiler we choose to create generic versions of EIN operator that can be instantiated to certain types.

\[
\lambda F, G. \langle F_{\alpha} \delta_{0i} + G_{\alpha} \delta_{1i} \rangle_{i:2\alpha}(F,G) \quad \lambda F, G, H. \langle F_{\alpha} \delta_{0i} + G_{\alpha} \delta_{1i} + H_{\alpha} \delta_{2i} \rangle_{i:3\alpha}(F,G,H)
\]

Since we are solely using existing EIN expressions to represent this computation, no new code needs to be added to the compiler.

**Testing** Once we successfully added a new operator to the surface language it is natural to write some test programs by hand. The hand-written tests were successful, but we know it is not a rigorous enough approach. We can test it more thoroughly by using the automated
testing model, \textit{DATm}. We added the concat operator to \textit{DATm} and used targeted testing to only generate test cases that use the concat operator. It created about 126 test programs that use the concat operator in 7 minutes.

We found bugs in our implementation and describe it in the following text. A numerical error arose when computing the determinant of the concatenation of a field.

\begin{verbatim}
field\#k(d)[2]F,G;
field\#k(d)[H = det(concat(F,G));
\end{verbatim}

Our normalization system applies index-based rewrites to reduce EIN expressions. A specific index rewrite is applied when the index in the delta term matches an index in tensor (or field) term. The rewrite checked if two indices were equal and did not distinguish between variable and constant indices. It is a mathematical mistake to reduce constant indices because they are not equivalent to variable indices. We were able to fix the bug and test it again.

\section*{7.2 Visualization Applications}

Scientists extract features to better understand inherent properties of their data. These features can be based on individual pixels, detection of regions with specific shape, time, and transformations of the data [38]. The work in this dissertation is designed to make it possible for a programmer to illustrate features in their data with Diderot.

\textbf{Visualization and Image-Analysis Programs} Chapter 2 describes the design and implementation of a new IR EIN for field reconstruction and other operations. The work makes it easier to develop algorithms that rely heavily upon higher-order operations. This section will describe three visualization features [41] to illustrate the impact and application of designing and implementing the EIN IR.

For scalar fields representing material properties over some scanned region (as with CT or MRI data), the boundaries between materials are important features, typically detected on a per-sample basis with edge detection. To visualize the continuous boundary as a connected
feature of the spatial domain, we can use one of the original definitions of edge detection by Canny [10]: the locations in a scalar field $F$ where the gradient magnitude $|\nabla F|$ is maximal with respect to motion along the gradient direction $\nabla F / |\nabla F|$ [10]. One could equivalently state that the gradient of the gradient magnitude $\nabla |\nabla F|$ is orthogonal (zero dot-product) to the gradient direction. The computation amounts to showing the zero-crossings of a derived scalar field $C$ expressed directly in Diderot as

$$\text{field} \#2(3)[] C = -\nabla \left( |\nabla F| \right) \bullet \nabla F / |\nabla F|;$$

in terms of the scalar field $\text{field} \#4(3)[] F$.

Vector fields arise in the analysis of fluid flow; properties of the derivatives of the vector field characterize important features (like vortices) in the flow. The curl $\nabla \times V$, for example, indicates the axis direction and magnitude of local rotation. One definition of vortices identifies them with places where the flow direction $V / |V|$ aligns with the curl direction $\nabla \times V / |\nabla \times V|$ [23].

**Normalized helicity** measures the angle between these directions:

$$\text{field} \#3(3)[] H = (V / |V|) \bullet (\nabla \times V / |\nabla \times V|);$$

in terms of the vector field $\text{field} \#4(3)[3] V$

Material properties like diffusivity and conductivity vary locally not just in magnitude and orientation but also in directional sensitivity, so they are modeled with second-order tensor fields. Visualizing the structure of tensor fields typically depends on measuring various tensor invariants, such as anisotropy: the magnitude of directional dependence. Neuroscientists study the architecture of human brain white matter with diffusion tensor fields computed from MRI [6]. A popular measure of diffusion anisotropy, “fractional anisotropy” can be directly expressed in Diderot as:

$$\text{field} \#4(3)[3,3] E = T - \text{trace}(T) \ast \text{idemity}[T] / 3;$$
$$\text{field} \#4(3)[] A = \sqrt{3.0 / 2.0} \ast |E| / |T|$$

which measures the magnitude of the purely anisotropic deviatoric tensor relative to the tensor $T$ itself.
Subsequent visualization or analysis of the fields so defined will typically require differentiation, such as the first derivatives needed for shading renderings of isocontours, or the second derivatives needed for extracting ridge and valley features. Generating expressions for $\nabla H$, and $\nabla A$ by hand is cumbersome and error-prone, whereas EIN allows Diderot to easily handles these, and even second derivatives like $\nabla \otimes \nabla A$.

**Compilation of Tensor Calculus**  The richer language inspired a new generation of Diderot programs. Unfortunately the programs suffered from a space blow-up issue and the programs could not be used. To address this size issue we developed compilation techniques (Chapter 3) to reduce to size of the programs and to enable compilation. The following are description of some of the visualization features that face this compilation issue when they were first created but can now be visualized with Diderot.

Curvature of a surface is defined by the relationship between small positional changes on the surface and changes in the surface normal [42]. The curvature transfer function will color more or less based on curvature. At each point we locally measure quantities that map via transfer function to optical quantities. The first-order differentiation can enhance clarity and produce effective renderings. It cannot be used to draw a line with consistent thickness. We can find the crest lines by taking the Hessian of these principles (fourth derivative overall). **Crest Lines** are places were the surface curvature is maximal along the curvature direction [48].

Crease surfaces are two-dimensional manifolds along which a scalar field assumes a local maximum (ridge) or a local minimum (valley) in a constrained space [28, 66]. Ridges are defined by an extremum with a local neighborhood of points. A point is an extreme point if the gradient of the field at that point is zero. The ordered eigenvalues ($\lambda_1 > \lambda_2 > \lambda_3$) are solutions to the Hessian of the function. A positive eigenvalue corresponds to convexity and indicates a valley surface and a negative eigenvalue corresponds to concavity and indicates a ridge surface. When determining a ridge surfaces the approximation is only as meaningful
as the difference between $\lambda_2$ and $\lambda_3$. 
CHAPTER 8
RELATED WORK

In this section, we discuss related work organized by area.

Visualization tools and languages  There are a variety of domain-specific languages and frameworks that provide similar features to those that are supported in Diderot. Shadie is a DSL for direct-volume rendering applications that is targeted at GPUs [36]. It is restricted to volume renderings and built-in functions. Scout is a DSL that extends the data-parallel programming model with shapes — regions of voxels in the image data — to accelerate visualization tasks on GPUs [52]. It is designed for algorithms that do computations over discrete voxels, such as stencil algorithms, instead of a continuous tensor field. Delite is a framework for implementing embedded parallel DSLs on heterogeneous processors [9, 11]. The host language for Delite is Scala. Vivaldi is a DSL that supports parallel volume rendering applications on heterogeneous systems [17]. It has a fixed volume rendering vocabulary and does not have the flexible notation that Diderot provides. ViSlang is a system to develop and integrate DSLs for visualization [61].

Einstein Index Notation  EIN (Chapter 2) is inspired by Einstein Index Notation, which is a concise written notation for tensor calculus invented by Albert Einstein [29]. Einstein index notation, sometimes called the summation convention, can be used to represent a wide array of physical quantities and algorithms in scientific computing [1, 5, 18, 25, 62, 63]. Various designers study the ambiguities and limitations of the notation to extend its uses on paper and develop grammar and semantics for implementation. A part of the ambiguity in index notation is related to the implicit summation. Therefore, various things are done to suppress summation [1]. These include using notation to differentiate between types of indices, using a no sum-operator [5], and between indices that repeat exactly twice [25, 63]. EIN notation uses an explicit summation symbol leading to more book-keeping but allowing
us to express explicit boundaries for diverse operations.

**Design and optimizations techniques** Domain-specific languages can offer several benefits. The syntax and type system can be designed to meet the practice and expectation of domain experts. The compiler can leverage common domain-specific traits. The programming model can abstract away from hardware and operating system decisions. By doing so, the end-user can write code that looks like the domain and let the system focus on generating high-performance code. There are various domain-specific languages that can provide a link between the mathematical algorithms and programming. We describe our contributions to the Diderot implementation in Chapter 2, and Chapter 3. This section focuses on four domain-specific compilers that are more closely related to our work (Spiral [57], TCE [46], COFFEE [51], and UFL [2]).

The tensor contraction engine (*TCE*) created by Hartono, Albert et. al represents quantum chemistry in a high-level Mathematica style language [21, 35, 46]. The class of computations are multi-dimensional summations over products of several arrays. They have a large number of nested loops and an explosively large parameter search space. The calculations can require larger space than available physical memory. To address this issue TCE developed algebraic transformations to reduce operation counts and their framework balances between loop fusion and memory usage. Like TCE, the Diderot language supports summations of products over several arrays. In addition to that we support a high level of expressiveness between tensors and tensor fields and field differentiation.

*Spiral* is a DSL created for digital signal processing [33, 54, 57]. Its design encapsulates significant mathematical knowledge of algorithms used in digital signal processing. Their work addresses the goal of doing the right transformation at the right level of abstraction. Its implementation uses IR *SPL* to represent the signal processing language, $\Sigma SPL$ to express loops and index mapping, and *C-IR* for code level optimization. $\Sigma SPL$ does loop merging and create complicated terms that are simplified with a set of rewriting rules [34].

139
EIN and ΣSPL both use a summation expression to illustrate indices, but their computations are expressed with algorithms, while EIN notation aims to expresses general tensor calculus.

The Unified Form Language (UFL) is a domain-specific language for representing weak formulations of partial differential equations [2]. UFL is most closely similar to EIN. At its core we both aim to support tensor algebra, high-level expressions with domain-drive abstraction, and offer differentiation (automated vs. symbolic). The difference in the development in our IR exists because of the difference in our domains, UFL is a language for expressing variational statements of PDEs and Diderot is a language for scientific visualization and image analysis. UFL creates an abstract representation that is used by several form compilers to generate low-level code. Since that, UFL avoids optimizations that a form compiler might want to leverage and instead sticks to a set of “safe and local” simplifications. Diderot controls the entire pipeline from surface language to code generation, and so it does have the opportunity to do optimal rewriting at a higher level.

COFFEE is a domain-specific compiler for local assembly kernels, an operation key to finding numerical solutions to partial differential equations [51]. COFFEE computes the contribution of a single cell in a discretized domain to the linear system to approximate a PDE. The entire computation is discretized into a larger number of cells, making the time to compute this computation important. As input COFFEE sees a scalarized tensor expression tree, and applies optimizations, like loop-invariant code motion, vector-register tiling, and expression splitting, on a low-level. Our IR applies optimizations at a higher level to exploit the mathematical properties of the computations on higher-order tensors before flattening

**Evaluating a Visualization** Verifiable visualization allows us to apply a verification process to visualization algorithms. Instead of real-world datasets one uses test cases with manufactured solutions. The manufactured solutions could be created in a way to predict result of algorithm with its implementation when evaluating a known model problem. Etiene et al. derives formulas for the expected order of accuracy of isosurface features and compare
them to experimentally observed results in order to provide confidence in behavior [30].

We use the idea of verification [4, 69] as a guiding metric for testing. To directly quote Etiene et al. “Verification is the process of assessing software correctness and numerical accuracy of the solution to a given mathematical model.” [69]. The measure of correctness for computations written in the Diderot language, is based on how accurately the output of the Diderot program represents the mathematical equivalent of the computations (Chapter 5).

Kindlmann and Scheideggar introduce three algebraic design principles: Principle of visual-data correspondence, unambiguous data depiction, and representation invariance [43]. The first two principles ensure that the data changes are well-matched with visual changes, and that the changes are informative. The Principle of Representation Invariance states that a visualization is invariant with respect to changes in data representation. If a change is visible, then that change is a hallucinator. We use the design concepts to compare two different sources of data in Chapter 6.

**Domain-Specific Testing** There are a variety of domain-specific languages and frameworks that provide similar features that are supported in Diderot, such as previously mentioned Vivaldi and ViSlang. There is no published work on testing (automated or otherwise) for these DSLs.

The importance of testing domain-specific languages has been discussed previously [60, 70]. Wu et al. introduces a framework, **DUFT**, to generate unit tests engines for DSLs [70] by adding a layer of DSL unit testing on top of existing general-purpose language tools and debugging services. **DUFT** tests DSL programs, but not the DSL implementation. **DATm** automatically tests the DSL implementation Chapter 5.

Ratiu et al. tested **mbeddr**, a set domain-specific languages on top of C built with HetBrains MPS language workbench [60]. Language developers define assertions from the specification of the DSL, and how it would be translated to the target language. Unlike **DATm**, **mbeddr** generates random models that do not pass type-checking rules and do not
reveal bugs. \textit{DATm} is proposing a model for verification. We are looking at a mathematical model and use that reasoning for test synthesis.

There is work on testing specific properties on general purpose languages. \textit{Herbie} is a tool can be used automatically to improve accuracy in floating point arithmetic [56]. The work introduces a method to evaluate the average accuracy of a floating-point expression, and localize the source of rounding error. It randomly samples inputs, generates rewriting candidates, and discovers rewrites to improve accuracy. Lindig [49] automatically tested consistency of C compilers, specifically C calling conventions, using randomly generated programs. The tool generates the types of functions and checked that the parameters were received. \textit{DATm} creates programs that are semantically more expressive.

\textbf{Types of Testing} \textit{Randomized differential testing} (RDT) is way of testing by examining two comparable systems [53, 71, 12, 24]. When the results differ (or one crashes), then there is a test case for a potential bug. RDT is a widely used method for testing compilers in practice. \textit{Csmith} [71] is a tool that can generate random C programs with the goal of finding deep optimization bugs. The programs are expressive and contain complex code. Similar to \textit{DATm}, Csmith effectively looks for deep optimization bugs in an atypical combination of language features. Unlike \textit{DATm}, Csmith evaluates results by comparing various C compilers, and so has “no ground truth.”

\textit{Metamorphic testing} involves evaluation of an unknown result based on some known property. Donaldson [24] \textit{et al.} applies metamorphic testing to graphic compilers by using value injections. When comparing images, they note that small differences in rendered images can occur even when there is no compiler bug. \textit{Mettoc} creates a family of programs and compares them using an equivalence relation [67]. When running \textit{DAVm}, while we did not know the numerical output of the visualization programs generated, we knew some property of the the result (symmetry).

Property-based testing has been developed before. \textit{Palka et al.} generates random and
type correct programs for the Glasgow Haskell compiler [55]. The output of optimized and unoptimized versions of the compiler are compared. *QuickCheck* [20] is a widely used testing tool that allows Haskell programmers to test properties of a program. It is an embedded language for writing properties. The type checker creates test cases that satisfy a condition. The test case generator is limited to a number of candidate test cases. Chen *et al.* [12] compares various compiler testing techniques. Besides RDT, they use “different optimization levels” (DOL) where they compare the output for comparable compilers at different optimization levels for the same program. They found it was effective at finding optimization-related type bugs.

**Choosing test cases** There is extensive work on how to create and choose test cases. McKeeman [53] describes test case reduction and test quality with differential testing. When generating test cases *EasyCheck* [19] focuses on traversal strategy. Bernardy, et. al [7] present a scheme that leads to reduction in the number of needed test cases. In addition to fixing types, they also fix some parameters passed to functions, effectively avoiding meaningless tests. *Feat* ([27]) focuses on functional enumeration. It exhaustively enumerates on possible data types. *Test set diversity metric* [31] is applied to ensure a diverse set of test cases. They use information distance (regardless of datatype) when automatically creating a diverse set of tests. Csmith [71] generates C programs minimizing unknown behavior, Palka [55] randomly generated lambda terms, and Lindig [49] tested C calling conventions.

**Finite Element Method and software** See Section 6.1
CHAPTER 9

CONCLUSION

The work in this dissertation has been both fundamental to the development of Diderot and important. We have illustrated a way to represent a language based on a high-level of math. We have presented our implementation techniques to solve compilation issues. We have provided a way to apply automated testing to a high-level DSL based on a ground truth.

The design and implementation of the EIN IR has increased the programmability of the Diderot language. We have made the compiler more robust with our implementation techniques and correctness proofs. We have also made our new features more reliable by developing an automated testing model for the language. We describe some ideas for future work in the following section.

9.1 Future Work

9.1.1 Correctness and Testing

Testing Coverage The subset of the Diderot language that is being tested is clearly described. DATm is testing the fundamental computations and types of the language, while DAVm puts those computations in the core part of a visualization program. We test various combinations of the operators and arguments in the exhaustive setting to imitate what can be expressed in the language. What is not clear is how many lines of the Diderot compiler are being tested. There is currently nothing implemented to mark or measure unused paths.

Smarter Test generation DATm generates small Diderot programs (for hefty tensor computations), and there is not much need for minimization or reducing the program size. There is a need to evaluate test case distribution when implementing random search in a deep test space. To make it feasible to do a larger search for test cases, DATm could do
smarter test generation by evaluating test cases based on existing cases or applying some size metric [31, 7].

**Clear Bug Reporting** There is currently not a way to automatically distinguish between the various types of compiler errors. A single bug could cause multiple test failures. It could be possible for the Diderot developer to design the compiler error messages, so that DATm could read them and then categorize the test cases. The bug log would then be organized by errors and its test cases.

**Types of Testing** It has been valuable to have an exhaustive approach to generating test programs since testing yielded interesting rare bugs, but an application of different types of testing could complement our testing process. While it is not helpful to test against earlier versions of Diderot (due to extensive language developments), we could possibly create a family of programs and do some variation of differential testing [53]. In the future, it would be interesting to evaluate the effectiveness of different testing approaches on the Diderot compiler [12].

**Parallel Testing** The time it takes to run a large numbers of tests is a limiting factor in the usefulness of the tool. DATm takes from 2–5 seconds per test (depending on the test’s complexity), which limits its use to either long runs or very sparse random testing. Fortunately, it should be fairly easy to run multiple tests in parallel using the parallelism of Unix processes on multicore servers or workstations.

### 9.1.2 Design and Implementation

**New language features** The work in this dissertation has made it easier to add operators to the surface language (Section 7.1). We implemented a limited version of an operation to build tensor fields. Our representation is limited to two or three tensor field arguments to
build a tensor field and is lacking generality. We need to create a new structure that accepts
n number of arguments with a mix of tensors and field components.

There are perhaps other operators that have not yet been explored. One example includes
the various built-in math functions and the more complicated eigensystem. Applications for
these operators in a visualization program can help drive the implementation for these new
features.

**Increase sharing**  The compilation issue has been largely solved, but there are other pos-
sible approaches to this problem. For one, we could redesign EIN with sharing in mind.
Secondly, we could make the sharing process visible during normalization. This would be
effective when differentiating and doing rewrites that clearly make replicate code but has
the potential to hide other common computations.

### 9.1.3 FEM and Diderot

**Debugging FEM data with Diderot**  The work on visualizing FEM data opens the door
to more useful applications. Diderot could potentially be used to visually debug and validate
fields created by FEM and possibly reveal hidden details in data created with higher-order
elements. We have shown an example of using Diderot to correctly visualize a field created
by Firedrake but have not explored its full potential.

**Mature visualization for FEM data**  We have yet to create mature visualizations with
our approach to visualize FEM fields. Partly, because the EIN IR (discussed in Chapter 2)
cannot yet represent FEM data. Additionally as far as we know, Firedrake does not offer
point evaluation for more complicated field differentiation. Once this patch for communica-
tion exists we should be able to do mature visualizations for FEM data using Diderot.

The point evaluation is also limited. It is not supported for some niche problems such as
extruded meshes, manifolds, and mixed element meshes. These structures are more compli-
cated than what is currently available. For instance, when considering a manifold we would
need to consider how it is represented and what we want to know about it.

**Better communication between Diderot and FEM** When a Diderot program is evaluating a tensor field it makes a call to Firedrake’s point evaluation functions. Each function call creates multiple tensor operations in order to do the right transformations and find the right cell. These operations can be similar to previous calls, leading to redundant and expensive computations. If we represent these steps in EIN then it is possible that Diderot can catch these redundant computations and the entire process is less expensive.

**Describing a FEM field** The syntax could be better designed to represent a fem field with two new datatypes; a mesh and a fptr. A mesh could note the reference element (elem) and polynomial order (o).

```plaintext
input string elem;
input int o;
input mesh(o) m = (elem, o);
```

There may be certain restrictions applied in the implementation. For one, the continuity of the field created (p) might be distinct from the number of derivatives available (q) by callback mechanisms.

```plaintext
input fptr#p.q(d)[σ] f;
```

The field will then be composed of both.

```plaintext
field#q(d)[σ] F = toField(m, f)
```

The syntax is more informative for the Diderot user.

### 9.1.4 Index notation on the surface language

Being able to support an index-like notation directly in Diderot could be beneficial to quick prototyping and debugging. Writing directly in index notation allows the developer to specify computations that may be difficult to replicate with existing surface level operators. It can
help a developer create intricate test cases to more rigorously test the implementation that involves the EIN IR.

**Writing directly in EIN IR** Writing directly in the EIN IR would have low implementation costs. The translation process would convert the surface level notation to the EIN IR and there would be no other new code to add to the compiler.

We will use notation $t\{\alpha\}$ to indicate the EIN term for tensor variable $t$ has indices initialized with $\alpha$. Consider, adding two permutations of the same tensor:

```c
  tensor[2,3,3] t;
  tensor[2,3,3] out = t{abc}+t{acb};
```

A direct mapping of the variable $t$ in term $t\{abc\}$ can create EIN expression $T_{ijk}$ and a permutation of the variable in $t\{acb\}$ creates term $T_{ikj}$ in the EIN operator:

$$
out = \lambda(T)(T_{ijk} + T_{ikj})_{i:a,j:b,k:c}(t)
$$

The translation from surface level to EIN would require some unique type checking to be sure the EIN indices are bound to appropriate dimensions. Consider permuting $g$

```c
  tensor[3,2,3] g;
  tensor[2,3,3] out1 = g{abc}; //NOT ok creates [3,2,3]
  tensor[2,3,3] out2 = g{cab}; //NOT ok creates [3,3,2]
  tensor[2,3,3] out3 = g{bac}; //ok
```

As you can see, not all permutations of $g$ can be added. Only the term $g\{b,a,c\}$ creates the right output shape $[2,3,3]$. The user input requires some unique type checking and perhaps some user-friendly error messages.

**Set of new rules** The translation process would require imposing certain restrictions on the syntax to clear up ambiguity. Such as the use of an implicit summation instead of supporting point-wise multiplication (as the EIN IR does). Consider writing several product operations between two tensors ($T$ and $M$) with code

148
tensor [d\_1, \ d\_2, \ d\_2] \ T;
tensor [d\_2, \ d\_2] \ M;
tensor [d\_1, d\_2, d\_2] \ inner = T\{abd\}*M\{dc\}; \ // inner product
tensor [d\_1] \ double = T\{abc\}*M\{bc\}; \ // double dot product
tensor [d\_1] \ augmented = T\{abc\}*M\{cb\}; \ // augmented product

The translation should recognize the redundant indices in the product term and create a summation operator.

\[ inner = \lambda(T, M) \left( \sum_{l} T_{ijl} \ast M_{lk} \right) \] \quad (T, M)

\[ double = \lambda(T, M) \left( \sum_{jk} T_{ijk} \ast M_{jk} \right) \] \quad (T, M)

\[ augmented = \lambda(T, M) \left( \sum_{jk} T_{ijk} \ast M_{kj} \right) \] \quad (T, M)

This type of translation should be clear to the user and compiler.

**Indicating covariant and contravariant indices** There is a distinction between index notation as defined by traditional Einstein index notation and what is represented by the EIN IR. Traditional Einstein index notation is widely used in textbooks and could be the most intuitive way for a user to write in index notation. The notation typically notes covariant and contravariant indices with an upper and lower index $M^{\mu}_{\nu}$. Diderot’s coordinate system assumes an orthonormal basis and so the distinction between indices does not matter. In the future, we may wish to make that distinction in the surface language.

\begin{align*}
tensor[\mu; \nu] \ M;
\end{align*}

There is currently no support to distinguish between covariant and contravariant indices in the compiler. It is unclear if we need to create a full translation from source language to code around the IR to be sure the mathematical value is maintained.
REFERENCES


APPENDIX A
PROOFS

A.1 Type Preservation Proof

Proof. The following is a proof for Theorem 4.1.1

Given a derivation \(d\) of the form \(e \rightarrow e'\) we state \(T(d)\) as a shorthand for the claim that the derivation preserves the type of the expression \(e\). For each rule, the structure of the left-hand-side term determines the last typing rule(s) that apply in the derivation of \(\Gamma\sigma \vdash e : \tau\). We then apply a standard inversion lemma and derive the type of the right-hand-side of the rewrite. The proof demonstrates that \(\forall d. T(d)\).

Case on structure of \(d\)

We will do a case analysis on the structure on the left-hand-side where \(\odot_n = \{*,/\}\).

First we will prove \(T(d)\) for \(\odot_n = *\) then \(\odot_n = /\). \(\odot_n = *\)

Find \(\Gamma\sigma \vdash ((e_1 \odot_n e_2)@x)\)

This type of structure inside a probe operation results in a tensor type.

\[(e_1 \odot_n e_2)@x\]

Find \(\Gamma\sigma \vdash (e_1 \text{ and } e_2)\)

\[
\frac{
\Gamma\sigma \vdash e_1 : (\sigma)F_d, \Gamma\sigma \vdash e_2 : (\sigma)F_d(T_{\text{Inv 11}})}{
\Gamma\sigma \vdash e_1 \ast e_2 : (\sigma)F_d(T_{\text{Inv 7}})}
\]

\[
\frac{
\Gamma\sigma \vdash (e_1 \ast e_2)@x : (\sigma)T
\Gamma\sigma \vdash ((e_1 @x)*(e_2 @x))
\Gamma\sigma \vdash e_1, e_2 : (\sigma)F_d,
\text{ then } \Gamma\sigma \vdash e_1@x, e_2@x : (\sigma)T \text{ by Ty_{Judge 7}, and } \Gamma\sigma \vdash e_1@x \ast e_2@x : (\sigma)T \text{ by Ty_{Judge 11}}
\} \Gamma\sigma \vdash (e_1@x)\odot_n (e_2@x)
\]

This type of structure inside a probe operation results in a tensor type.
\((e_1 \odot_n e_2)@x\)

Find \(\Gamma \sigma \vdash (e_1 \text{ and } e_2)\)

\[
\Gamma \sigma \vdash e_1 : (\sigma) \mathcal{F}_d, \Gamma \sigma \vdash e_2 : () \mathcal{F}_d (\text{Ty}_{\text{Inv}} 12)
\]

\[
\Gamma \sigma \vdash \left(\frac{e_1}{e_2}\right) : (\sigma) \mathcal{F}_d (\text{Ty}_{\text{Inv}} 7)
\]

\[
\Gamma \sigma \vdash \left(\frac{e_1@x}{e_2@x}\right) : (\sigma) \mathcal{T}
\]

Find \(\Gamma \sigma \vdash \left(\frac{e_1@x}{e_2@x}\right)\)

Given that \(\Gamma \sigma \vdash e_1 : (\sigma) \mathcal{F}_d, \Gamma \vdash e_2 : () \mathcal{F}_d\) then \(\Gamma \sigma \vdash e_1@x : (\sigma) \mathcal{T}\) by Ty_{Judge} 7,

\[
\Gamma \vdash e_2@x : () \mathcal{T} \text{ by Ty}_{\text{Judge}} 7, \text{ and } \Gamma \sigma \vdash \frac{e_1@x}{e_2@x} : (\sigma) \mathcal{T} \text{ by Ty}_{\text{Judge}} 12.
\]

T( R1 for \(\odot_n = /\))

T( R1) OK

R2. \((e_0 \odot_2 e_1)@x \Rightarrow (e_0@x) \odot_2 (e_1@x)\)

\(\odot_2 = + | -\)

Find \(\Gamma \sigma \vdash ((e_1 \odot_2 e_2)@x)\)

This type of structure inside a probe operation results in a tensor type.

\((e_0 \odot_2 e_1)@x\)

Find \(\Gamma \sigma \vdash (e_1 \text{ and } e_2)\)

\[
\Gamma \sigma \vdash e_1, e_2 : (\sigma) \mathcal{F}_d (\text{Ty}_{\text{Inv}} 10)
\]

\[
\Gamma \sigma \vdash e_1 \odot_2 e_2 : (\sigma) \mathcal{F}_d (\text{Ty}_{\text{Inv}} 7)
\]

\[
\Gamma \sigma \vdash (e_1 \odot_2 e_2)@x : (\sigma) \mathcal{T}
\]

Find \(\Gamma \sigma \vdash ((e_1@x) \odot_2 (e_2@x))\)

Given that \(\Gamma \sigma \vdash e_1, e_2 : (\sigma) \mathcal{F}_d\)

then \(\Gamma \sigma \vdash e_1@x, e_2@x : (\sigma) \mathcal{T}\) by Ty_{Judge} 7 and \(\Gamma \sigma \vdash e_1@x \odot_2 e_2@x : (\sigma) \mathcal{T}\) by Ty_{Judge} 10

R3. \((\odot_1 e_1)@x \Rightarrow \odot_1 (e_1@x)\)

We will do a case analysis on the structure on the left-hand-side where \(\odot_1 = \{|-M(.)|\}\).

First we will prove \(T(d)\) for \(\odot_1 = -\) then \(\odot_1 = M(.)\). case \(\odot_1 = -\),
Find $\Gamma_{\sigma} \vdash ((-e_1)@x)$

This type of structure inside a probe operation results in a tensor type.

$(\odot_{1}e_1)@x$

Find $\Gamma_{\sigma} \vdash (e_1)$

\[
\begin{align*}
\Gamma_{\sigma} & \vdash e_1 : (\sigma)F_d(Ty_{\text{Inv} 10}) \\
\Gamma_{\sigma} & \vdash -e_1 : (\sigma)F_d(Ty_{\text{Inv} 7})
\end{align*}
\]

\[
\Gamma_{\sigma} \vdash (-e_1)@x : (\sigma)T
\]

Find $\Gamma_{\sigma} \vdash (-e_1@x)$

Given that $\Gamma_{\sigma} \vdash e_1 : (\sigma)F_d$ then $\Gamma_{\sigma} \vdash e_1@x : (\sigma)T$ by $Ty_{\text{Judge} 7}$ and $\Gamma_{\sigma} \vdash -e_1@x : (\sigma)T$

by $Ty_{\text{Judge} 10}$

\[
T(\text{ R3 for } \odot_{1} = -)
\]

$\odot_{1} = M(e_1)$

Note: $M(e_1) = \sqrt{e_1} \mid \kappa(e_1) \mid \exp(e_1) \mid e^n$

Find $\Gamma_{\sigma} \vdash ((M(e_1))@x)$

This type of structure inside a probe operation results in a tensor type.

$(\odot_{1}e_1)@x$

Find $\Gamma_{\sigma} \vdash (e_1)$

\[
\begin{align*}
\Gamma_{\sigma} & \vdash e_1 : (\sigma)F_d(Ty_{\text{Inv} 9}) \\
\Gamma_{\sigma} & \vdash M(e_1) : (\sigma)F_d(Ty_{\text{Inv} 7})
\end{align*}
\]

\[
\Gamma_{\sigma} \vdash M(e_1)@x : (\sigma)T
\]

Find $\Gamma_{\sigma} \vdash (M(e_1@x))$

Given that $\Gamma_{\sigma} \vdash e_1 : (\sigma)F_d$

\[
\text{then } \Gamma_{\sigma} \vdash e_1@x : (\sigma)T \text{ by } Ty_{\text{Judge} 7} \text{ and } \Gamma_{\sigma} \vdash M(e_1@x) : (\sigma)T \text{ by } Ty_{\text{Judge} 9}
\]

T(\text{ R3 for } \odot_{1} = M)

T(\text{ R3}) \text{ OK}

\[
R4. (\sum_{i=1}^{n} e_1)@x \implies \sum_{i=1}^{n} (e_1@x)
\]

158
We will do a case analysis on the structure on the left-hand-side where \( \chi = \{ \text{lift}(e_1) | \delta_{ij} | \mathcal{E}_\alpha \} \).

First we will prove \( T(d) \) for \( \chi = \text{lift}(e_1) \) then \( \chi = \delta_{ij} | \mathcal{E}_\alpha \).

**Case **\( \chi = \text{lift}(e_1) \)

Find \( \Gamma \vdash ((\chi(e_1))@x) \)

This type of structure inside a probe operation results in a tensor type.

(\( \chi@x \))

Find \( \Gamma \vdash (e_1) \)

\[ \Gamma \vdash e_1 : (\sigma)\mathcal{F}_d(Ty_{\text{Inv~8}}) \]

\[ \Gamma \vdash (\text{lift}(e_1)) : (\sigma)\mathcal{F}_d(Ty_{\text{Inv~7}}) \]

\[ \Gamma \vdash (\text{lift}(e_1))@x : (\sigma)\mathcal{T} \]

Find \( \Gamma \vdash (\text{lift}(e_1@x)) \)

Given that \( \Gamma \vdash e_1 : (\sigma)\mathcal{F}_d \)

then \( \Gamma \vdash e_1@x : (\sigma)\mathcal{T} \) by \( Ty_{\text{Judge~7}} \) and \( \Gamma \vdash \text{lift}(e_1@x) : (\sigma)\mathcal{T} \) by \( Ty_{\text{Judge~8}} \)

\( T( \text{R5 where } \chi = \text{lift}(e_1) ) \)

For the case \( \chi = \delta_{ij} | \mathcal{E}_\alpha \)

Given that \( \Gamma \vdash \chi : \tau \) then \( \Gamma \vdash \chi@x : \tau \) by \( Ty_{\text{Judge~7}} \)

\( T( \text{R5 where } \chi = \delta_{ij} | \mathcal{E}_\alpha ) \)

\( T( \text{R5} ) \text{ OK} \)

\( R6. \nabla_i \circ (e_1 \ast e_2) \implies e_1(\nabla_i \circ e_2) + e_2(\nabla_i \circ e_1) \)

Included in the earlier prose.

\( T( \text{R6} ) \text{ OK} \)

\( R7. \nabla_i \circ \left( \frac{e_1}{e_2} \right) \implies \frac{(\nabla_i \circ e_1)e_2 - e_1(\nabla_i \circ e_2)}{e_2^2} \)
Included in the earlier prose.

R8. $\nabla_i \diamond (\sqrt{e_1}) \Rightarrow \text{lift}(1/2) \ast \frac{\nabla_i \circ e_1}{\sqrt{e_1}}$

Find $\Gamma \vdash (\nabla_i \diamond (\sqrt{e_1}))$

This type of structure inside a derivative operation results in a field type
and the $\sqrt{e_1}$ term results in a scalar.

Claim: $\Gamma \vdash \sqrt{e_1} : () \mathcal{F}_{d}$ then $\Gamma_i \vdash \nabla_i \diamond (\sqrt{e_1}) : (i) \mathcal{F}_{d}$ by TyJudge 4

Find $\Gamma \vdash (e_1)$

\[
\begin{align*}
\Gamma & \vdash e_1 : () \mathcal{F}_{d} (\text{TyInv 9}) \\
\Gamma & \vdash \sqrt{e_1} : () \mathcal{F}_{d} (\text{TyInv 4})
\end{align*}
\]

$\Gamma \vdash \nabla_i \diamond (\sqrt{e_1}) : (\sigma) \mathcal{F}_{d}$ and $\sigma = \{d\}$ (Claim)

Given that $\Gamma \vdash e_1 : () \mathcal{F}_{d}$

then $\Gamma_d \vdash \nabla_{i:d} \diamond e_1 : (i) \mathcal{F}_{d} (\text{TyJudge 4})$ and $\Gamma \vdash \sqrt{e_1} : () \mathcal{F}_{d} (\text{TyJudge 9})$

Additionally, $\Gamma \vdash \text{lift}(-) : (\sigma) \mathcal{F}_{d}$ by (TyJudge 8)

Find $\Gamma \vdash (\text{lift}(1/2) \ast \frac{\nabla_i \circ e_1}{\sqrt{e_1}})$

Given that $\Gamma \vdash \sqrt{e_1} : () \mathcal{F}_{d}$ and $\Gamma_d \vdash \nabla_{i:d} \diamond e_1 : (i) \mathcal{F}_{d}$

then $\Gamma_d \vdash \frac{\nabla_{i:d} \circ e_1}{\sqrt{e_1}} : (i) \mathcal{F}_{d}$ by TyJudge 12 and $\Gamma_d \vdash \text{lift}(1/2) \ast \frac{\nabla_i \circ e_1}{\sqrt{e_1}} : (i) \mathcal{F}_{d}$ by TyJudge 11

R9. $\nabla_i \diamond (\text{cosine}(e_1)) \Rightarrow (-\text{sine}(e_1)) \ast (\nabla_i \circ e_1)$

Find $\Gamma \vdash (\nabla_i \diamond (\text{cosine}(e_1)))$

This type of structure inside a derivative operation results in a field type
and the $\text{cosine}(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \text{cosine}(e_1) : () \mathcal{F}_{d}$ then $\Gamma_i \vdash \nabla_i \diamond (\text{cosine}(e_1)) : (i) \mathcal{F}_{d}$ by TyJudge 4

Find $\Gamma \vdash (e_1)$

\[
\begin{align*}
\Gamma & \vdash e_1 : () \mathcal{F}_{d} (\text{TyInv 9}) \\
\Gamma & \vdash \text{cosine}(e_1) : () \mathcal{F}_{d}
\end{align*}
\]

Find $\Gamma \vdash ((-\text{sine}(e_1)) \ast (\nabla_i \circ e_1))$

Given that $\Gamma \vdash e_1 : () \mathcal{F}_{d}$ then $\Gamma_d \vdash \nabla_i \circ e_1 : (i) \mathcal{F}_{d}$ by TyJudge 4, $\Gamma \vdash \text{sine}(e_1) : () \mathcal{F}_{d}$ by
TyJudge 9,

\[ \Gamma \vdash -\text{sine}(e_1) : ()F_d \text{ by TyJudge 10, and } \Gamma_d \vdash (-\text{sine}(e_1)) \ast (\nabla_i \diamond e_1) : (i)F_d \text{ by TyJudge 11} \]

T( R9) OK

R10. \( \nabla_i \diamond (\text{sine}(e_1)) \implies (\text{cosine}(e_1)) \ast (\nabla_i \diamond e_1) \)

Included in the earlier prose.

T( R10) OK

R11. \( \nabla_i \diamond (\text{tangent}(e_1)) \implies \frac{\nabla_i \diamond e_1}{\text{cosine}(e_1) \ast \text{cosine}(e_1)} \)

Find \( \Gamma_\sigma \vdash (\nabla_i \diamond (\text{tangent}(e_1))) \)

This type of structure inside a derivative operation results in a field type

and the \( \text{tangent}(e_1) \) term results in a scalar.

Claim: \( \Gamma \vdash \text{tangent}(e_1) : ()F_d \text{ then } \Gamma_i \vdash \nabla_i \diamond (\text{tangent}(e_1)) : (i)F_d \text{ by TyJudge 4} \)

Find \( \Gamma_\sigma \vdash (e_1) \)

\[ \Gamma \vdash e_1 : ()F_d \text{ (TyInv 9)} \]

\[ \Gamma \vdash \text{tangent}(e_1) : ()F_d \]

Find \( \Gamma_\sigma \vdash (\frac{\nabla_i \diamond e_1}{\text{cosine}(e_1) \ast \text{cosine}(e_1)}) \)

Given that \( \Gamma \vdash e_1 : ()F_d \)

then \( \Gamma_d \vdash \nabla_i \diamond e_1 : (i)F_d \text{ by TyJudge 4, } \Gamma \vdash \text{cosine}(e_1) \ast \text{cosine}(e_1) : ()T \text{ by TyJudge 9, TyJudge 11, and } \Gamma \vdash \frac{\nabla_i \diamond e_1}{\text{cosine}(e_1) \ast \text{cosine}(e_1)} : ()T \text{ by TyJudge 12} \)

T( R11) OK

R12. \( \nabla_i \diamond (\text{arccosine}(e_1)) \implies \left( \frac{-\text{lift}(1)}{\sqrt{\text{lift}(1) - (e \ast e)}} \right) \ast (\nabla_i \diamond e_1) \)

Similar approach to R13

T( R12) OK

R13. \( \nabla_i \diamond (\text{arcsine}(e_1)) \implies \left( \frac{\text{lift}(1)}{\sqrt{\text{lift}(1) - (e \ast e)}} \right) \ast (\nabla_i \diamond e_1) \)
Find $\Gamma \vdash (\nabla_i \circ (\text{arcsine}(e_1)))$

This type of structure inside a derivative operation results in a field type

and the $\text{arcsine}(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \text{arcsine}(e_1) : (\mathcal{F}_d$ then $\Gamma_i \vdash \nabla_i \circ (\text{arcsine}(e_1)) : (i)\mathcal{F}_d$ by $\text{TyJudge}$ 4

Find $\Gamma \vdash (e_1)$

$\Gamma \vdash e_1 : (\mathcal{F}_d(\text{TyInv} 9))$

$\Gamma \vdash \text{arcsine}(e_1) : (\mathcal{F}_d$)

Since $\Gamma \vdash e_1 : (\mathcal{F}_d$ then $\Gamma_i \vdash \nabla_i \circ e_1 : (i)\mathcal{F}_d$ by $\text{TyJudge}$ 4

Find $\Gamma \vdash (\text{lift}(1))$

$\Gamma \vdash \text{lift}(1) : (\sigma)\mathcal{F}_d(\text{TyJudge } 8 )$

Find $\Gamma \vdash ((\frac{\text{lift}(1)}{\sqrt{\text{lift}(1)-(e*e)}}) \ast (\nabla_i \circ e_1))$

Given that $\Gamma \vdash e_1 : (\mathcal{F}_d$ then $\Gamma \vdash e_1 \ast e_1 : (\mathcal{F}_d$ by $\text{TyJudge}$ 11,

$\Gamma \vdash \text{lift}(1)-(e_1 \ast e_1) : (\mathcal{F}_d$ by $\text{TyJudge}$ 10, $\Gamma \vdash \sqrt{\text{lift}(1)-(e_1 \ast e_1)} : (\mathcal{F}_d$ by $\text{TyJudge}$ 9,

$\Gamma \vdash \frac{\text{lift}(1)}{\sqrt{\text{lift}(1)-(e_1 \ast e_1)}} : (\mathcal{F}_d$ by $\text{TyJudge}$ 12, and $\Gamma \vdash \frac{\text{lift}(1)}{\sqrt{\text{lift}(1)-(e_1 \ast e_1)}} \ast (\nabla_i \circ e_1) : (i)\mathcal{F}_d$

by $\text{TyJudge}$ 11  

R14.$\nabla_i \circ (\text{arctangent}(e_1)) \implies \frac{\text{lift}(1)}{\sqrt{\text{lift}(1)+(e_1 \ast e_1)}} \ast (\nabla_i \circ e_1)$

Similar approach to  R13

T(  R14)  OK

R15.$\nabla_i \circ (\text{exp}(e_1)) \implies \text{exp}(e_1) \ast (\nabla_i \circ e_1)$

Find $\Gamma \vdash (\nabla_i \circ (\text{exp}(e_1)))$

This type of structure inside a derivative operation results in a field type

and the $\text{exp}(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \text{exp}(e_1) : (\mathcal{F}_d$ then $\Gamma_i \vdash \nabla_i \circ (\text{exp}(e_1)) : (i)\mathcal{F}_d$ by $\text{TyJudge}$ 4

Find $\Gamma \vdash (e_1)$

162
\[
\begin{align*}
\Gamma \vdash e_1 : (\mathcal{F}_d(\text{Ty}_{\text{Inv}} 9)) \\
\Gamma \vdash \exp(e_1) : () \mathcal{F}_d
\end{align*}
\]

Find \( \Gamma_\sigma \vdash (\exp(e_1) \ast (\nabla_i \circ e_1)) \)

Given that \( \Gamma \vdash e_1 : (\mathcal{F}_d) \) then \( \Gamma_d \vdash \nabla_i \circ e_1 : (i) \mathcal{F}_d \) by (Ty_Judge 4),
\[
\Gamma \vdash \exp(e_1) : (\mathcal{F}_d) \text{ by (Ty}_\text{Judge} 9), \text{ and } \Gamma_d \vdash \exp(e_1) \ast (\nabla_i \circ e_1) : (i) \mathcal{F}_d \text{ by (Ty}_\text{Judge} 11)
\]

T( R15) OK

R16.\( \nabla_i \circ (\exp^n) \implies \text{lift}(n) \ast \exp^{n-1} \ast (\nabla_i \circ e_1) \)

This type of structure inside a derivative operation results in a field type and the \( e_1^n \) term results in a scalar.

Claim: \( \Gamma \vdash e_1^n : (\mathcal{F}_d) \) then \( \Gamma_i \vdash \nabla_i \circ (e_1^n) : (i) \mathcal{F}_d \) by Ty_Judge 4

Find \( \Gamma_\sigma \vdash (e_1) \)
\[
\begin{align*}
\Gamma \vdash e_1 : (\mathcal{F}_d), \Gamma \vdash n : () \mathcal{T} \text{ and } \sigma = \{d\}(\text{Ty}_{\text{Inv}} 9) \\
\Gamma_\sigma \vdash (e_1^n) : (\sigma \backslash d) \mathcal{F}_d(\text{Ty}_{\text{Inv}} 4)
\end{align*}
\]
\[
\Gamma_\sigma \vdash \nabla_i : (e_1^n) : (\sigma) \mathcal{F}_d
\]

Given that \( \Gamma \vdash e_1 : (\mathcal{F}_d) \) then \( \Gamma_d \vdash \nabla_i \circ e_1 : (i) \mathcal{F}_d \) by (Ty_Judge 4).

Given that \( \Gamma \vdash e_1 : (\mathcal{F}_d), \Gamma \vdash n : () \mathcal{T} \)

then \( \Gamma \vdash \text{lift}(n) : (\mathcal{F}_d) \text{ by (Ty}_\text{Judge} 8) \) and \( \Gamma \vdash e_1^{n-1} : (\mathcal{F}_d) \text{ by (Ty}_\text{Judge} 9).\)

Given that \( \Gamma \vdash e_1^{n-1} : (\mathcal{F}_d) \) and \( \Gamma_d \vdash \nabla_i \circ e_1 : (i) \mathcal{F}_d \)

then \( \Gamma_d \vdash \text{lift}(n) \ast e_1^{n-1} \ast (\nabla_i \circ e_1) : (i) \mathcal{F}_d \text{ by (Ty}_\text{Judge} 11).\)

T( R16) OK

R17.\( \nabla_i (e_1 \circ e_2) \implies (\nabla_i e_1) \circ (\nabla_i e_2) \)

Find \( \Gamma_\sigma \vdash (\nabla_i (e_1 \circ e_2)) \)

This type of structure inside a derivative operation results in a field type.

Given the subterm: \( \Gamma_\sigma / i \vdash e_1 \circ e_2 : (\sigma / i) \mathcal{F}_d \)

then by Ty_Judge 4 we know it’s derivative \( \Gamma_\sigma \vdash \nabla_i \circ (e_1 \circ e_2) : (\sigma) \mathcal{F}_d \)
Find $\Gamma \vdash (\tau(e_1) \text{ and } \tau(e_2))$

$\Gamma_{\sigma \setminus d} \vdash e_1, e_2 : (\sigma \setminus d) F_d (\text{Ty}_{\text{Inv}} 10)$

$\Gamma_{\sigma \setminus d} \vdash e_1 \odot e_2 : (\sigma \setminus d) F_d (\text{Ty}_{\text{Inv}} 4)$

$\Gamma_{\sigma} \vdash \nabla_i (e_1 \odot e_2) : (\sigma) F_d$

Find $\Gamma_{\sigma} \vdash ((\nabla_i e_1) \odot (\nabla_i e_2))$

Given that $\Gamma_{\sigma} \vdash e_1, e_2 : (\sigma \setminus d) F_d$

then $\Gamma_{\sigma} \vdash \nabla_i e_1 : (\sigma) F_d$ by (Ty$_{\text{Judge}} 4$) and $\Gamma_{\sigma} \vdash (\nabla_i e_1) \odot (\nabla_i e_2) : (\sigma) F_d$ by (Ty$_{\text{Judge}} 10$).

T( R17) OK

R18. $\nabla_i (-e_1) \implies - (\nabla_i \odot e_1)$

Find $\Gamma_{\sigma} \vdash (\nabla_i (-e_1))$

This type of structure inside a derivative operation results in a field type.

Given the subterm: $\Gamma_{\sigma / \ i} \vdash -e_1 : (\sigma / \ i) F_d$
then by Ty$_{\text{Judge}} 4$ we know it’s derivative $\Gamma_{\sigma} \vdash \nabla_i \odot (-e_1) : (\sigma) F_d$

Find $\Gamma_{\sigma} \vdash (e_1)$

$\Gamma_{\sigma \setminus d} \vdash e_1 : (\sigma \setminus d) F_d (\text{Ty}_{\text{Inv}} 10)$

$\Gamma_{\sigma \setminus d} \vdash -e_1 : (\sigma \setminus d) F_d (\text{Ty}_{\text{Inv}} 4)$

$\Gamma_{\sigma} \vdash \nabla_i (-e_1) : (\sigma) F_d$

Find $\Gamma_{\sigma} \vdash (- (\nabla_i \odot e_1))$

Given that $\Gamma_{\sigma} \vdash e_1 : (\sigma \setminus d) F_d$

then $\Gamma_{\sigma} \vdash \nabla_i e_1 : (\sigma) F_d$ by (Ty$_{\text{Judge}} 4$) and $\Gamma_{\sigma} \vdash (\nabla_i e_1) \odot (\nabla_i e_2) : (\sigma) F_d$ by (Ty$_{\text{Judge}} 10$)

T( R18) OK

R19. $\nabla \odot \sum_{v:1}^{u=n} e_1 \implies \sum_{v:1}^{u=n} (\nabla \odot e_1)$

This type of structure inside a derivative operation results in a field type.

Given the subterm: $\Gamma_{\sigma / \ i} \vdash \sum_{v:1}^{u=n} : (\sigma / \ i) F_d$
then by Ty$_{\text{Judge}} 4$ we know it’s derivative $\Gamma_{\sigma} \vdash \nabla_i \odot (\sum_{v:1}^{u=n}) : (\sigma) F_d$
Find $\Gamma \sigma \vdash (e_1)$

$\Gamma_{\sigma \setminus d,v : n} \vdash e_1 : (\sigma \setminus d,v : n)F_d (Ty_{\text{Inv}} 3)$

$\Gamma_{\sigma \setminus d} \vdash \left( \sum_{v : 1}^{u=n} e_1 \right) : (\sigma)F_d (Ty_{\text{Inv}} 4)$

$\Gamma \sigma \vdash \nabla_{i,d}(\sum_{v : 1}^{u=n} e_1) : (\sigma)F_d$

Find $\Gamma \sigma \vdash (\sum_{v : 1}^{u=n} (\nabla \circ e_1))$

Given that $\Gamma \sigma \vdash e_1 : (\sigma \setminus d,v : n)F_d$

then $\Gamma \sigma \vdash \nabla_{i,d}(e_1) : (\sigma,v : n)F_d$ by ($Ty_{\text{Judge}} 4$) and $\Gamma \sigma \vdash \sum_{v : 1}^{u=n}(\nabla_{i,d}(e_1)) : (\sigma)F_d$ by ($Ty_{\text{Judge}} 3$)

$T(\text{R19})$ OK

R20. $\nabla \circ \chi \Longrightarrow 0$

This type of structure inside a derivative operation results in a field type.

Given the subterm: $\Gamma_{\sigma / i} \vdash \nabla \circ \chi : (\sigma / i)F_d$

then by $Ty_{\text{Judge}} 4$ we know it’s derivative $\Gamma \sigma \vdash \nabla_{i} \circ (\nabla \circ \chi) : (\sigma)F_d$

Lastly, $\Gamma \sigma \vdash 0 : (\sigma)F_d$ by ($Ty_{\text{Judge}} 8$).

$T(\text{R20})$ OK

R21. $\nabla i \circ (V_\alpha \otimes H^u) \Longrightarrow (V_\alpha \otimes h^{iu})$

Given $\Gamma \sigma \vdash V_\alpha \otimes H^u : (\sigma / i)F_d$ by ($Ty_{\text{Judge}} 2$) then $\Gamma \sigma \vdash \nabla i \circ (V_\alpha \otimes H^u) : (\sigma)F_d$ by ($Ty_{\text{Judge}} 4$).

Lastly, $\Gamma \sigma \vdash (V_\alpha \otimes h^{iu}) : (\sigma)F_d$ by ($Ty_{\text{Judge}} 2$).

$T(\text{R21})$ OK

R22. $- - e_1 \Longrightarrow e_1$

Find $\Gamma \sigma \vdash ( - - e_1)$

Assign generic type $\Gamma \sigma \vdash e_1 : \tau$

Find $\Gamma \sigma \vdash (e_1)$

165
\[
\Gamma_\sigma \vdash -e_1 : \tau(Ty_{Inv} 10)
\]
\[
\Gamma_\sigma \vdash -e_1 : \tau(Ty_{Inv} 10)
\]
\[
\Gamma_\sigma \vdash e_1 : \tau
\]

T( R22) OK

R23. \(-0 \rightarrow 0\)
Find \(\Gamma_\sigma \vdash (-0)\)
Assign generic type \(\Gamma_\sigma \vdash -0 : \tau\)
Find \(\Gamma_\sigma \vdash (0)\)
\[
\Gamma_\sigma \vdash 0 : \tau(Ty_{Inv} 10)
\]
\[
\Gamma_\sigma \vdash -0 : \tau
\]
T( R23) OK

R24. \(e_1 - 0 \rightarrow e_1\)
Find \(\Gamma_\sigma \vdash (e_1 - 0)\)
Assign generic type \(\Gamma_\sigma \vdash e_1 - 0 : \tau\)

\text{Case analysis on } \tau
\tau = (\sigma)T
\[
\Gamma_\sigma \vdash e - 0 : (\sigma)T
\]
\[
\Gamma_\sigma \vdash 0 : (\sigma)T \text{ by } (Ty_{Judge} 1)
\]
T( R24)
\tau = (\sigma)F_d

Not permitted
T( R24) OK

R25. \(0 - e_1 \rightarrow -e_1\)
Similar approach to R24
T( R25) OK

R26. $\frac{e_1}{e_1} \rightarrow 0$
Similar approach to R24
T( R26) OK

R27. $\frac{e_1}{e_2} \rightarrow \frac{e_1}{e_2e_3}$
Included in the earlier prose.

R28. $\frac{e_1}{e_3} \rightarrow \frac{e_1e_3}{e_2}$
Similar approach to R27
T( R28) OK

R29. $\frac{e_1}{e_3} \rightarrow \frac{e_1e_4}{e_2e_3}$

$\Gamma_{\sigma} \vdash \frac{e_1}{e_3} : \tau$

case analysis on $\tau$

$\tau = (\sigma)T$

Find $\Gamma_{\sigma} \vdash (e_1, e_2, e_3, \text{ and } e_4)$

$\Gamma_{\sigma} \vdash e_1 : (\sigma)T$

$\Gamma_{\sigma} \vdash e_2, e_3, e_4 : ()T(T_{Inv} \ 12)$

$\Gamma_{\sigma} \vdash \frac{e_1}{e_2} : (\sigma)T$ and $\frac{e_3}{e_4} : ()T(T_{Inv} \ 12)$

$\Gamma_{\sigma} \vdash \frac{e_1}{e_2} : (\sigma)T$

Find $\Gamma_{\sigma} \vdash (\frac{e_1e_4}{e_2e_3})$

Given that $\Gamma_{\sigma} \vdash e_1 : (\sigma)T$ and $\Gamma \vdash e_2, e_3, e_4 : ()T$ then $\Gamma_{\sigma} \vdash e_1 * e_4 : (\sigma)T$ by (TyJudge 11),

$\Gamma \vdash e_2 * e_3 : ()T$ by (TyJudge 11), and $\Gamma_{\sigma} \vdash \frac{e_1e_4}{e_2e_3} : (\sigma)T$ by (TyJudge 12).
\[ T(\text{ R29 for } \tau = (\sigma)T) \]
\[ \tau = (\sigma)\mathcal{F}_d \text{ similar to above case} \]

\[ T(\text{ R29}) \text{ OK} \]

\[ R30.0 + e_1, e_1 + 0 \Rightarrow e_1 \]
Similar approach to \text{ R24}

\[ T(\text{ R30}) \text{ OK} \]

\[ R31.0e, e_0 \Rightarrow 0 \]
Similar approach to \text{ R24}

\[ T(\text{ R31}) \text{ OK} \]

\[ R32.\sqrt{(e_1)} * \sqrt{(e_1)} \Rightarrow e_1 \]
Assign generic type \( \Gamma \sigma \vdash \sqrt{(e_1)} * \sqrt{(e_1)} : \tau \)

Find \( \Gamma \sigma \vdash (e_1) \)
\[
\begin{align*}
\Gamma \sigma \vdash e_1 : \tau & \quad \text{(TyInv 9)} \\
\Gamma \sigma \vdash \sqrt{e_1} : \tau & \quad \text{(TyInv 11)} \\
\Gamma \sigma \vdash \sqrt{e_1} * \sqrt{e_1} : \tau & \quad \text{ (TyInv 11)}
\end{align*}
\]

\[ T(\text{ R32}) \text{ OK} \]

\[ R33.\mathcal{E}_{ijk} \nabla_{ij} \circ e_1 \Rightarrow \text{ lift}(0) \]
Similar approach to \text{ R34}

\[ T(\text{ R33}) \text{ OK} \]

\[ R34.\mathcal{E}_{ijk}(V_{\alpha} \otimes h^{jk}) \Rightarrow \text{ lift}(0) \]
Given \( \Gamma \sigma \vdash V_{\alpha} \otimes h^{jk} : (\sigma)\mathcal{F}_d \) by TyJudge 2 then \( \Gamma \sigma \vdash \epsilon_{ijk}V_{\alpha} \otimes h^{jk} : (\sigma)\mathcal{F}_d \) by TyJudge 6.

Lastly, \( \Gamma \sigma \vdash \text{ lift}(0) : (\sigma)\mathcal{F}_d \) by TyJudge 8
R35. $E_{ijk}E_{ilm} \implies \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$

We know $\Gamma_{\sigma} \vdash E_{ijk}E_{ilm} : (\sigma)T$ by TyJudge 6.

Given $\Gamma_{\sigma} \vdash \delta_{jl}\delta_{km} : (\sigma)T$ by TyJudge 5 then $\Gamma_{\sigma} \vdash \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} : (\sigma)T$ by TyJudge 10.

T( R35) OK

R36. $\delta_{ij}T_j \implies T_i$

Find $\Gamma_{\sigma} \vdash (\delta_{ij}T_j)$

Given $\Gamma_{\sigma} \vdash T_j : (\sigma)T$ and $\sigma = \{j\}$ by (TyJudge 1)

then $\Gamma_{\sigma} \vdash \delta_{ij}(T_j) : (\sigma)T$ and $\sigma = \{i\}$ by (TyJudge 5) by (TyJudge 5)

Find $\Gamma_{\sigma} \vdash (T_i)$

$\Gamma_{\sigma} \vdash T_i : (\sigma)F_d$ and $\sigma = \{i\}$ by (TyJudge 1)

T( R36)

T( R36) OK

R37. $\delta_{ij}F_j \implies F_i$

Similar approach to R36

T( R37) OK

R38. $\delta_{ij}V \otimes H^\delta_{cij} \implies V \otimes H^\delta_{ci}$

Given $\Gamma_{\sigma} \vdash V \otimes H^\delta_{cij} : (\sigma)F_d$ and $\sigma = \{j\}$ by (TyJudge 2)

then $\Gamma_{\sigma} \vdash \delta_{ij}(V \otimes H^\delta_{cij}) : (\sigma)F_d$ and $\sigma = \{i\}$ by (TyJudge 5)

$\Gamma_{\sigma} \vdash V \otimes H^\delta_{ci} : (\sigma)F_d$ and $\sigma = \{i\}$ by (TyJudge 2)

T( R38)

T( R38) OK
R39. \( \delta_{ij} V \otimes H^{\delta_{ij}}(x) \implies V \otimes H^{\delta_{ij}}(x) \)

Similar approach to R38

T( R39) OK

R40. \( \delta_{ij} \nabla_j \diamond e_1 \implies \nabla_i \diamond (e_1) \)

Included in the earlier prose.

R41. \( \Sigma(se_1) \implies se_1 \)

Included in the earlier prose.

R42. \( \nabla_\alpha \diamond \nabla_\beta \diamond e_1 \implies \nabla_{\beta\alpha} \diamond e_1 \)

This type of structure inside a derivative operation results in a field type.

Claim: \( \Gamma_{\sigma/\alpha\beta} \vdash e_1 : (\sigma/\alpha\beta)\mathcal{F}_d \)

then \( \Gamma_{\sigma} \vdash (\nabla_\beta \diamond e_1) : (\sigma/\alpha)\mathcal{F}_d \) by TyJudge 4 , and \( \Gamma_{\sigma} \vdash (\nabla_\alpha \diamond \nabla_\beta \diamond e_1) : (\sigma)\mathcal{F}_d \) by TyJudge 4

Given that \( \Gamma_{\sigma} \vdash e : \sigma/\alpha\beta \) then \( \Gamma_{\sigma} \vdash \nabla_{\beta\alpha}e : (\sigma)\mathcal{F}_d \) by TyJudge 4

T( R42) OK

T(d) Lemma 4.1.1

A.2 Value Preservation Proof

Proof. The following is a proof for Theorem 4.2.1

Given a derivation \( d \) of the form \( e \rightarrow e' \) we state \( V(d) \) as a shorthand for the claim that the derivation preserves the value of the expression \( e \). The proof demonstrates that \( \forall d.V(d) \).

Case on structure of \( d \)
R1. \((e_1 \circ_n e_2)@x \implies (e_1@x) \circ_n (e_2@x)\)

Included in the earlier prose.

R2. \((e_0 \circ_2 e_1)@x \implies (e_0@x) \circ_2 (e_1@x)\)

Similar approach to  R1

V( R2) OK

R3. \((\circ_1 e_1)@x \implies \circ_1(e_1@x)\)

where \(\circ_1 \in \{\sqrt{}, -, \kappa, \exp, (\cdot)^n, \Sigma\}\)

Claim \((\circ_1 e_1)@x \) evaluates to \(v\)

Assume that \(\Psi \rho \vdash e_1 \downarrow v_1\)

then \(\Psi \rho \vdash (\circ_1 e)@x \downarrow \circ_1 App(v_1)[x] \) by (ValJudge 9),

and \(\Psi \rho \vdash \circ_1(e_1@x) \downarrow \circ_1 App(v_1)[x] \) by (ValJudge 6).

The last step leads to \(\circ_1(e_1@x) \downarrow v\)

V( R3) OK

R4. \((\sum_{i=1}^{i=n} e_1)@x \implies \sum_{i=1}^{i=n}(e_1@x)\)

Similar approach to  R3

V( R4) OK

R5. \((\chi)@x \implies \chi\)

Claim \((\chi)@x \) evaluates to \(v\)

Assume that \(\chi \downarrow v'\)

where \(\chi = \text{lift}(\cdot) | \delta | \mathcal{E}\)

\(\Psi \rho \vdash (\chi)@x \downarrow v' \) by (ValJudge 10) which leads to \(v = v'\)

The last step leads to \(\chi \downarrow v\)

V( R5) OK
\[ R6. \nabla_i (e_1 \ast e_2) \implies e_1 (\nabla_i e_2) + e_2 (\nabla_i e_1) \]

Included in the earlier prose.

\[ R7. \nabla_i \left( \frac{e_1}{e_2} \right) \implies \frac{(\nabla_i e_1) e_2 - e_1 (\nabla_i e_2)}{e_2^2} \]

Claim \( \nabla_i \left( \frac{e_1}{e_2} \right) \) evaluates to \( v \)

Assume subterms: \( \Psi \rho \vdash e_1 \downarrow v_1, \Psi \rho \vdash e_2 \downarrow v_2 \)

then \( \Psi \rho \vdash \frac{e_1}{e_2} \downarrow \frac{v_1}{v_2} \) by (Val\_Judge 7) and \( \Psi \rho \vdash \nabla_i \left( \frac{e_1}{e_2} \right) \downarrow \nabla_i \left( \frac{v_1}{v_2} \right) \) by (Val\_Judge 8)

The value of \( v \) is \( v = \nabla_i \left( \frac{v_1}{v_2} \right) \)

The last step leads to \( \frac{(\nabla_i e_1) e_2 - e_1 (\nabla_i e_2)}{e_2^2} \downarrow v \)

R\( \text{V( R7) OK} \)

\[ R8. \nabla_i \left( \sqrt{e_1} \right) \implies \text{lift}(1/2) \ast \frac{\sqrt{\nabla_i e_1}}{\sqrt{e_1}} \]

Claim \( \nabla_i \left( \sqrt{e_1} \right) \) evaluates to \( v \)

Assume that \( \Psi \rho \vdash e_1 \downarrow v_1 \)

then \( \Psi \rho \vdash \sqrt{e_1} \downarrow \) by (Val\_Judge 6), and \( \Psi \rho \vdash \nabla_i \left( \sqrt{e_1} \right) \downarrow \nabla_i \left( \sqrt{v_1} \right) \) by (Val\_Judge 8)

The value of \( v \) is \( v = \nabla_i \left( \sqrt{v_1} \right) \)

The value of \( v \) is \( v = \frac{\nabla_i v_1}{\text{Real}(2) \sqrt{v_1}} \) by applying the chain rule

Given that \( e_1 \downarrow v_1 \) and \( \Psi \rho \vdash \text{lift}(1/2) \downarrow \text{Real}(1/2) \) by (Val\_Judge 1, Val\_Judge 5)

then \( \Psi \rho \vdash \nabla_i \ast e \downarrow \nabla_i v_1 \) by (Val\_Judge 8), \( \Psi \rho \vdash \sqrt{e} \downarrow \sqrt{v_1} \) by (Val\_Judge 6),
\[ \Psi_\rho \vdash \nabla_i \psi \downarrow \frac{\nabla_i v}{\sqrt{v}} \text{ by (ValJudge 7)} ; \text{ and } \Psi_\rho \vdash \text{lift} \left( \frac{1}{2} \right) \ast \frac{\nabla_i \psi}{\sqrt{v}} \downarrow \frac{\nabla_i v}{\text{Real}(2)\sqrt{v}} \text{ by (ValJudge 7)}. \]

The last step leads to \( \text{lift} \left( \frac{1}{2} \right) \ast \frac{\nabla_i \psi}{\sqrt{v}} \downarrow v \)

\[ V(\ R8) \, \text{OK} \]

\[ R9. \nabla_i \diamond \text{(cosine}(e_1)) \implies (\text{sine}(e_1)) \ast (\nabla_i \diamond e_1) \]

Included in the earlier prose.

\[ R10. \nabla_i \diamond (\text{sine}(e_1)) \implies (\text{cosine}(e_1)) \ast (\nabla_i \diamond e_1) \]

Similar approach to \( R9 \)

\[ V(\ R10) \, \text{OK} \]

\[ R11. \nabla_i \diamond (\text{tangent}(e_1)) \implies \frac{\nabla_i \psi}{\text{cosine}(e_1) \ast \text{cosine}(e_1)} \]

Similar approach to \( R9 \)

\[ V(\ R11) \, \text{OK} \]

\[ R12. \nabla_i \diamond (\text{arccosine}(e_1)) \implies \left( \frac{-\text{lift}(1)}{\sqrt{\text{lift}(1)-(\ast e)}} \right) \ast (\nabla_i \diamond e_1) \]

Similar approach to \( R9 \)

\[ V(\ R12) \, \text{OK} \]

\[ R13. \nabla_i \diamond (\text{arcsine}(e_1)) \implies \left( \frac{\text{lift}(1)}{\sqrt{\text{lift}(1)-(\ast e)}} \right) \ast (\nabla_i \diamond e_1) \]

Similar approach to \( R9 \)

\[ V(\ R13) \, \text{OK} \]

\[ R14. \nabla_i \diamond (\text{arctangent}(e_1)) \implies \frac{\text{lift}(1)}{\text{lift}(1)+(e_1 \ast e)} \ast (\nabla_i \diamond e_1) \]

Similar approach to \( R9 \)

\[ V(\ R14) \, \text{OK} \]
R15. $\nabla_i \circ (\exp(e_1)) \Longrightarrow \exp(e_1) * (\nabla_i \circ e_1)$

Claim $\nabla_i \circ (\exp(e_1))$ evaluates to $v$

Assume that $\Psi_\rho \vdash e_1 \Downarrow v_1$

then $\Psi_\rho \vdash \exp(e_1) \Downarrow \exp(v_1)$ by (ValJudge 6)

and then $\Psi_\rho \vdash \nabla_i \circ (\exp(e_1)) \Downarrow \nabla_i(\exp(v_1))$ by (ValJudge 8),

The value of $v$ is $\nabla_i(\exp(v_1))$

$v = \exp(v_1) * (\nabla_i v_1)$ by applying identity

Given that $e_1 \Downarrow v_1$ and $\Psi_\rho \vdash \exp(e_1) \Downarrow exp(v_1)$ by (ValJudge 6),

then $\Psi_\rho \vdash \nabla_i \circ e_1 \Downarrow \nabla_i v_1$ by (ValJudge 8),

and $\Psi_\rho \vdash \exp(e_1) * (\nabla_i \circ e_1) \Downarrow exp(v_1) * \nabla_i v_1$ by (ValJudge 7).

The last step leads to $\exp(e_1) * (\nabla_i \circ e_1) \Downarrow v$

V( R15) OK

R16. $\nabla_i \circ (e_1^n) \Longrightarrow \text{lift}(n) * e_1^{n-1} * (\nabla_i \circ e_1)$

Claim $\nabla_i \circ (e_1^n)$ evaluates to $v$

Assume that $\Psi_\rho \vdash e_1 \Downarrow v_1$

then $\Psi_\rho \vdash e_1^n \Downarrow v_1^n$ by (ValJudge 6), and $\Psi_\rho \vdash \nabla_i \circ (e_1^n) \Downarrow \nabla_i(v_1^n)$ by (ValJudge 8)

The value of $v$ is $\nabla_i(v_1^n)$

$v = \text{Real}(n) * e_1^{n-1} * (\nabla_i v)$ by Applying identity

Given that $e_1 \Downarrow v_1$ then $\Psi_\rho \vdash \text{lift}(n) \Downarrow \text{Real}(n)$ by (ValJudge 1, ValJudge 5),

$\Psi_\rho \vdash e_1^{n-1} \Downarrow e_1^{n-1}$ by (ValJudge 6), $\Psi_\rho \vdash \nabla_i \circ e_1 \Downarrow \nabla_i v_1$ by (ValJudge 8),

and $\Psi_\rho \vdash \text{lift}(n) * e_1^{n-1} * (\nabla_i \circ e_1) \Downarrow \text{Real}(n)e_1^{n-1}(\nabla_i v_1)$ by (ValJudge 7).

The last step leads to $\text{lift}(n) * e_1^{n-1} * (\nabla_i \circ e_1) \Downarrow v$

V( R16) OK

R17. $\nabla_i(e_1 \odot e_2) \Longrightarrow (\nabla_i e_1) \odot (\nabla_i e_2)$
where \( \odot_2 = + | - \)

Claim \( \nabla_i(e_1 \odot e_2) \) evaluates to \( v \)

Assume subterms: \( \Psi \vdash e_1 \downarrow v_1, \Psi \vdash e_2 \downarrow v_2 \)

then \( \Psi \vdash (e_1 \odot_2 e_2) \downarrow v_1 \odot_2 v_2 \) by (ValJudge 7),

and \( \Psi \vdash \nabla_i \circ (e_1 \odot_2 e_2) \downarrow \nabla_i(v_1 \odot_2 v_2) \) by (ValJudge 8).

The value of \( v \) is \( \nabla_i(v_1 \odot_2 v_2) \)

\[ v = \nabla_i v_1 \odot_2 \nabla_i v_2 \] by distributing differentiation

Given that \( e_1 \downarrow v_1, e_2 \downarrow v_2 \) then \( \Psi \vdash \nabla_i \circ e_1 \downarrow \nabla_i v_1, \nabla_i \circ e_2 \downarrow \nabla_i v_2 \) by (ValJudge 8) ,

and \( \Psi \vdash (\nabla_i e_1) \circ (\nabla_i e_2) \downarrow \nabla_i v_1 \odot_2 \nabla_i v_2 \) by (ValJudge 7).

The last step leads to \( (\nabla_i e_1) \circ (\nabla_i e_2) \downarrow v \)

\( \mathbb{V} (\text{R17}) \) OK

R18. \( \nabla_i(-e_1) \implies -(\nabla_i \circ e_1) \)

Claim \( \nabla_i(-e_1) \) evaluates to \( v \)

Assume that \( \Psi \vdash e_1 \downarrow v_1 \)

then \( \Psi \vdash -e_1 \downarrow -v_1 \) by (ValJudge 6) , and \( \Psi \vdash \nabla_i \circ (-e_1) \downarrow \nabla_i(-v_1) \) by (ValJudge 8)

The value of \( v \) is \( \nabla_i(-v_1) \)

\[ v = -(\nabla_i(v_1)) \] by Distribute differentiation

Given that \( e_1 \downarrow v_1 \) then \( \Psi \vdash \nabla_i \circ e_1 \downarrow \nabla_i v_1 \) by (ValJudge 8) ,

\( \Psi \vdash - (\nabla_i \circ e_1) \downarrow -(\nabla_i v_1) \) by (ValJudge 6)

The last step leads to \( - (\nabla_i \circ e_1) \downarrow v \)

\( \mathbb{V}(\text{R18}) \) OK

R19. \( \nabla \circ \sum_{v:1}^{v=n} e_1 \implies \sum_{v:1}^{v=n}(\nabla \circ e_1) \)

Similar approach to R18

\( \mathbb{V}(\text{R19}) \) OK
R20. $\nabla \diamond \chi \Rightarrow 0$

Claim $\nabla \diamond$ evaluates to $v$

Assume $\chi \Downarrow v_1$ then $\Psi_\rho \vdash \nabla_i \diamond (\chi) \Downarrow \nabla_i v_1$ by (ValJudge 8)

$v_1 = Real(0)$ by applying differentiation to a constant

Lastly, $\Psi_\rho \vdash 0 \Downarrow Real(0)$ by (ValJudge 1, ValJudge 5)

The last step leads to $0 \Downarrow v$

V(R20) OK

R21. $\nabla_i \diamond (V_\alpha \otimes H^\nu) \Rightarrow (V_\alpha \otimes h^{iv})$

Claim $\nabla_i \diamond (V_\alpha \otimes H^\nu)$ evaluates to $v$

Since $\Psi_\rho \vdash V_\alpha \otimes H^\nu \Downarrow Field(V[\alpha], \nabla[\nu])$ by (ValJudge 4)

then $\Psi_\rho \vdash \nabla_i \diamond (V_\alpha \otimes H^\nu) \Downarrow Field(V[\alpha], \nabla[i\nu])$ by (ValJudge 8)

The value of $v$ is $Field(V[\alpha], \nabla[i\nu])$

Lastly, $\Psi_\rho \vdash V_\alpha \otimes h^{iv} \Downarrow Field(V[\alpha], \nabla[i\nu])$ by ValJudge 4

The last step leads to $(V_\alpha \otimes h^{iv}) \Downarrow v$

V(R21) OK

R22. $- - e_1 \Rightarrow e_1$

Claim $- - e_1$ evaluates to $v$

Assume that $e_1 \Downarrow v'$ then $\Psi_\rho \vdash - e_1 \Downarrow - v'$ by (ValJudge 6), and $\Psi_\rho \vdash - - e_1 \Downarrow - - v'$ by (ValJudge 6) The value of $v$ is $- - v'$.

By using algebraic reasoning: $- - v' = v'$. Since $- - e_1 \Downarrow v$ and $- - e_1 \Downarrow v'$ then $v = v'$.

The last step leads to $e_1 \Downarrow v$

V(R22) OK

R23. $-0 \Rightarrow 0$

Claim $-0$ evaluates to $v$
then $\Psi_\rho \vdash 0 \Downarrow \text{Real}(0)$ by (ValJudge 1), and $\Psi_\rho \vdash -0 \Downarrow \text{Real}(-0)$ by (ValJudge 6).

The value of $v$ is $\text{Real}(-0)$.

By using algebraic reasoning: $\text{Real}(-0) = \text{Real}(0)$.

The last step leads to $0 \Downarrow v$

V( R23) OK

R24. $e_1 - 0 \implies e_1$

Included in the earlier prose.

R25. $0 - e_1 \implies -e_1$

Claim $0 - e_1$ evaluates to $v$

Assume that $- e_1 \Downarrow v'$ then $\Psi_\rho \vdash 0 - e_1 \Downarrow \text{Real}(0) + v'$ by (ValJudge 1, ValJudge 7).

The value of $v$ is $\text{Real}(0) + v'$. By using algebraic reasoning: $\text{Real}(0) + v' = v'$. Since $0 - e_1 \Downarrow v$ and $0 - e_1 \Downarrow v'$ then $v = v'$

The last step leads to $- e_1 \Downarrow v$

V( R25) OK

R26. $\frac{0}{e_1} \implies 0$

Assume that $e_1 \Downarrow \text{Real}(v_2)$ then $\Psi_\rho \vdash \frac{0}{e_1} \Downarrow \text{Real}(\frac{0}{v_2})$ by (ValJudge 1, ValJudge 7).

The value of $v$ is $\text{Real}(\frac{0}{v_2})$. By using algebraic reasoning: $\text{Real}(\frac{0}{v_2}) = \text{Real}(0)$

Lastly, $\Psi_\rho \vdash 0 \Downarrow \text{Real}(0)$ by (ValJudge 1)

The last step leads to $0 \Downarrow v$

V( R26) OK

R27. $\frac{e_1}{e_3} \implies \frac{e_1}{e_2 e_3}$

Claim $\frac{e_1}{e_3}$ evaluates to $v$

Assume that $\frac{e_1}{e_2 e_3} \Downarrow v', e_1 \Downarrow v_1, e_2 \Downarrow v_2, e_3 \Downarrow v_3$. 

177
then \( \Psi_\rho \vdash \frac{e_1}{e_2} \downarrow \frac{v_1}{v_2} \) by (Val\_Judge 7) and \( \Psi_\rho \vdash \frac{e_2}{e_3} \downarrow \frac{v_2}{v_3} \) by (Val\_Judge 7).

Given that \( e_1 \downarrow v_1 \) \( e_2 \downarrow v_2 \) \( e_3 \downarrow v_3 \)

then \( \Psi_\rho \vdash e_2 e_3 \downarrow v_2 * v_3 \) by Val\_Judge 7 and \( \Psi_\rho \vdash \frac{e_1}{e_2 e_3} \downarrow \frac{v_1}{v_2 * v_3} \) by Val\_Judge 7.

The value of \( v \) is \( \frac{v_1}{v_2 * v_3} \). By using algebraic reasoning: \( v' = \frac{v_1}{v_2 * v_3} = \frac{v_1}{v_3} = v \).

The last step leads to \( \frac{e_1}{e_2 e_3} \downarrow v \)

\( V(\text{R27}) \) OK

\[ R28. \frac{e_1}{e_2} \Rightarrow \frac{e_1 e_3}{e_2} \]

Similar approach to \( \text{R27} \)

\( V(\text{R28}) \) OK

\[ R29. \frac{e_1}{e_2} \Rightarrow \frac{e_1 e_4}{e_2 e_3} \]

Similar approach to \( \text{R27} \)

\( V(\text{R29}) \) OK

\[ R30. 0 + e_1, e_1 + 0 \Rightarrow e_1 \]

Claim \( 0 + e_1, e_1 + 0 \) evaluates to \( v \)

Assume that \( e_1 \downarrow v' \) then \( \Psi_\rho \vdash e_1 + 0 \downarrow v' + \text{Real}(0) \) by (Val\_Judge 1, Val\_Judge 7).

By using algebraic reasoning \( v' + \text{Real}(0) = v' \)

The last step leads to \( e_1 \downarrow v \)

\( V(\text{R30}) \) OK

\[ R31. 0, e_0 \Rightarrow 0 \]

Similar approach to \( \text{R26} \)

\( V(\text{R31}) \) OK
R32. $\sqrt{(e_1)} \ast \sqrt{(e_1)} \implies e_1$

Included in the earlier prose.

R33. $\mathcal{E}_{ijk} \nabla_{ij} \circ e_1 \implies \text{lift}(0)$

Included in the earlier prose.

R34. $\mathcal{E}_{ijk}(V_{ij} \otimes h^{jk}) \implies \text{lift}(0)$

Similar approach to R33

$\forall ( R34) \ OK$

R35. $\mathcal{E}_{ijk} \mathcal{E}_{ilm} \implies \delta_{ji}\delta_{km} - \delta_{jm}\delta_{kl}$

Included in the earlier prose.

R36. $\delta_{ij} T_j \implies T_i$

Included in the earlier prose.

R37. $\delta_{ij} F_j \implies F_i$

Claim $\delta_{ij} F_j$ evaluates to $v$

Given that $\Psi_\rho \vdash F_j \downarrow \text{Field}(F[b_j], \nabla[])$ by ValJudge 3 and $\Psi_\rho \vdash \delta_{ij} \downarrow K_{ij}$ by ValJudge 11

then $\Psi_\rho \vdash \delta_{ij} F_j \downarrow \text{Field}(F[b_j \cdot b_i \cdot b_j], \nabla[])$ by Equation 4.2

The value of $v$ is $\text{Field}(F[b_j \cdot b_i \cdot b_j], \nabla[])$

By using algebraic reasoning to analyze $v$

$v = \text{Field}(F[b_i], \nabla[])$ by reducing value $b_j \cdot b_j$ using Equation 1.5

Lastly, $\Psi_\rho \vdash F_i \downarrow \text{Field}(F[b_i], \nabla[])$ by (ValJudge 3)

The last step leads to $F_i \downarrow v$

$\forall ( R37) \ OK$

R38. $\delta_{ij} V \otimes H^{\delta_{ij}} \implies V \otimes H^{\delta_{ci}}$

179
Claim $\delta_{ij} V \otimes H^{\delta_{cj}}$ evaluates to $v$

Given that $\Psi_\rho \vdash V \otimes H^{\delta_{cj}} \downarrow \text{Field}(F[], \nabla[b_j])$ by ValJudge 3 and $\Psi_\rho \vdash \delta_{ij} \downarrow K_{ij}$ by ValJudge 11

then $\Psi_\rho \vdash \delta_{ij} V \otimes H^{\delta_{cj}} \downarrow \text{Field}(F[], \nabla[b_j \cdot b_i \cdot b_j])$ by Equation 4.2

The value of $v$ is $\text{Field}(F[], \nabla[b_j \cdot b_i \cdot b_j])$.

By using algebraic reasoning to analyze $v$

$v = \text{Field}(F[], \nabla[b_i])$ by reducing value $b_j \cdot b_j$ using Equation 1.5

Lastly, $\Psi_\rho \vdash V \otimes H^{\delta_{ci}} \downarrow \text{Field}(F[], \nabla[b_i])$ by (ValJudge 3)

The last step leads to $V \otimes H^{\delta_{ci}} \downarrow v$

V( R38) OK

R39. $\delta_{ij} V \otimes H^{\delta_{cj}}(x) \implies V \otimes H^{\delta_{ci}}(x)$

Claim $\delta_{ij} V \otimes H^{\delta_{cj}}(x)$ evaluates to $v$

Assume that $V \otimes H^{\delta_{ci}}(x) \downarrow v'$

Define partial value $\delta_{ij} V \otimes H^{\delta_{cj}} \downarrow v_1$

Given that $\Psi_\rho \vdash V \otimes H^{\delta_{cj}} \downarrow \text{Field}(F[], \nabla[b_j])$ by ValJudge 3 and $\Psi_\rho \vdash \delta_{ij} \downarrow K_{ij}$ by ValJudge 11

then $\Psi_\rho \vdash \delta_{ij} V \otimes H^{\delta_{cj}} \downarrow \text{Field}(F[], \nabla[b_j \cdot b_i \cdot b_j])$ by Equation 4.2

The value of $v_1$ is $\text{Field}(F[], \nabla[b_j \cdot b_i \cdot b_j])$.

By using algebraic reasoning to analyze $v$

$v = \text{Field}(F[], \nabla[b_i])$ by reducing value $b_j \cdot b_j$ using Equation 1.5

Given that $\delta_{ij} V \otimes H^{\delta_{cj}} \downarrow \text{Field}(F[], \nabla[b_i])$

then $\Psi_\rho \vdash \delta_{ij} V \otimes H^{\delta_{cj}}(x) \downarrow \text{App}(\text{Field}(F[], \nabla[b_i]))[x]$ by (ValJudge 9)

Given that $\Psi_\rho \vdash V \otimes H^{\delta_{ci}} \downarrow \text{Field}(F[], \nabla[b_i])$ by ValJudge 3

then $\Psi_\rho \vdash V \otimes H^{\delta_{ci}}(x) \downarrow \text{App}(\text{Field}(F[], \nabla[b_i]))[x]$ by (ValJudge 9)

The last step leads to $V \otimes H^{\delta_{ci}}(x) \downarrow v$

V( R39) OK

180
R40. $\delta_{ij} \nabla_j \circ e_1 \implies \nabla_i \circ (e_1)$

Claim $\delta_{ij} \nabla_j \circ (e_1)$ evaluates to $v$

Assume that $\Psi_{\rho} \vdash e_1 \downarrow v_1$.

This type of structure inside a derivative evaluates to a field value.

Therefore, $v_1 = Field(e_1[\alpha], \nabla[\beta])$.

Given that $\Psi_{\rho} \vdash \delta_{ij} \downarrow K_{ij}$ by (ValJudge 11) and $\Psi_{\rho} \vdash \nabla_j \circ e_1 \downarrow Field(e_1[\alpha], \nabla[\beta \cdot b_j])$ by ValJudge 3

then $\Psi_{\rho} \vdash \delta_{ij} \nabla_j \circ (e_1) \downarrow Field(e_1[\alpha], \nabla[\beta \cdot b_j \cdot b_i \cdot b_j])$ by Equation 4.3

The value of $v$ is $Field(e_1[\alpha], \nabla[\beta \cdot b_j])$.

$v = Field(e_1[\alpha], \nabla[\beta \cdot b_i])$ by reducing value $b_j \cdot b_j$ using Equation 1.5

Lastly, $\Psi_{\rho} \vdash \nabla_i (e) \downarrow Field(e_1[\alpha], \nabla[\beta])$ by (ValJudge 8)

The last step leads to $\nabla_i \circ (e_1) \downarrow v$

V( R40) OK

R41. $\Sigma(s e_1) \implies s \Sigma e_1$

Claim $\Sigma(s e_1)$ evaluates to $v$

Assume that $s \downarrow v_s$ and $e_1 \downarrow v_e$

then $\Psi_{\rho} \vdash s \cdot e_1 \downarrow v_s \cdot v_e$ by (ValJudge 7) and $\Psi_{\rho} \vdash \Sigma(s e_1) \downarrow \Sigma(v_s \cdot v_e)$ by (ValJudge 6)

The value of $v$ is $\Sigma(v_s \cdot v_e)$

$v = v_s \cdot \Sigma(v_e)$ by moving scalar outside summation

Given that $s \downarrow v_s$ and $e \downarrow v_e$

then $\Psi_{\rho} \vdash \Sigma e \downarrow \Sigma v_e$ by (ValJudge 6) and $\Psi_{\rho} \vdash s \Sigma e_1 \downarrow v_s \cdot \Sigma v_e$ by (ValJudge 7)

The last step leads to $s \Sigma e_1 \downarrow v$

V( R41) OK

R42. $\nabla_{\alpha} \circ \nabla_{\beta} \circ e_1 \implies \nabla_{\beta \alpha} \circ e_1$

181
Claim $\nabla_\alpha \diamond \nabla_\beta \diamond e_1$ evaluates to $v$

Assume that $\Psi_\rho \vdash e_1 \Downarrow v_1$

then $\Psi_\rho \vdash \nabla_\beta \diamond e_1 \Downarrow \nabla_\beta v_1$ by $(\text{ValJudge} 8)$ and $\Psi_\rho \vdash \nabla_\alpha \diamond \nabla_\beta \diamond e_1 \Downarrow \nabla_\alpha \nabla_\beta v_1$ by $(\text{ValJudge} 8)$.

The value of $v$ is $\nabla_\alpha \nabla_\beta v_1$.

$v = \nabla_{\alpha\beta} v_1$ by rewriting to compact representation

$v = \nabla_{\beta\alpha} v_1$ by switching order of differentiation

Given that $e_1 \Downarrow v_1$ then $\Psi_\rho \vdash \nabla_{\beta\alpha} \diamond e_1 \Downarrow \nabla_{\beta\alpha} v_1$ by $(\text{ValJudge} 8)$

The last step leads to $\nabla_{\beta\alpha} \diamond e_1 \Downarrow v$

$V(\text{ R42})$ OK

A.3 Termination

A.3.1 Size reduction

If $e \implies e'$ then $S(e) > S(e') \geq 0$ (Lemma 4.3.1). The following are a few helpful lemmas that will be referred to in the proof.

Lemma A.3.1. $5^{(1+x)} > (16 + 5^x)$

$5^x > 4. \quad \text{Given } x \geq 1$

$4 \times 5^x > 16 \quad \text{Multiply by 4}$

$5 \times 5^x - 5^x > 16 \quad \text{Refactor left side}$

$5 \times 5^x > (16 + 5^x) \quad \text{Add } 5^x$

$5^{(1+x)} > (16 + 5^x) \quad \text{Rewritten}$

Lemma A.3.2. $5^{(S e_1 + S e_2)} > 5^{(S e_1)} > 4.$

Lemma A.3.3. $(1 + S e_1)5^{(1+S e_1)} > S e_1(16 + 5^{S e_1}) + 20$
\[
5(1 + S e_1) > 16 + 5S e_1
\]

Lemma A.3.1

\[
S e_15(1 + S e_1) > S e_1(16 + 5S e_1)
\]

Multiply by \( S e_1 \)

\[
S e_15(1 + S e_1) + 5(1 + S e_1) > S e_1(16 + 5S e_1) + 5(1 + S e_1)
\]

Add \( 5(1 + S e_1) \)

\[
(1 + S e_1)5(1 + S e_1) > S e_1(16 + 5S e_1) + 5 * 5S e_1 > S e_1(16 + 5S e_1) + 20
\]

(Lemma A.3.2)

Proof. The following is a proof for Lemma 4.3.1

Given a derivation \( d \) of the form \( e \rightarrow e' \) we state \( P(d) \) as a shorthand for the claim that the derivation reduces the size of the expression \( e \). By case analysis and comparing the size metric provided. This proof does a case analysis to show \( \forall d \in \text{Deriv}.P(d) \). Case on structure of \( d \)

R1. \((e_1 \odot_n e_2)@x \implies (e_1@x) \odot_n (e_2@x)\)

Included in the earlier prose.

R2. \((e_0 \odot_2 e_1)@x \implies (e_0@x) \odot_2 (e_1@x)\)

The possible values of \( d \) are \((e_1 + e_2)@x\) or \((e_1 - e_2)@x\).

Since \( S((e_0 \odot_2 e_1)@x) = 2 + 2S e_1 + 2S e_2 \) and \( S((e_0@x) \odot_2 (e_1@x)) = 1 + 2S e_1 + 2S e_2 \)

then \( S(e_0 \odot_2 e_1)@x > S(e_0@x) \odot_2 (e_1@x)\),

\( P(d) \)

R3. \((\odot_1 e_1)@x \implies \odot_1 (e_1@x)\)

The possible values of \( d \) are \( \sqrt{e}@x, (-e)@x, (ke)@x, (\exp(e))@x, (e^n)@x \)

Since \( S((\odot_1 e_1)@x) = 2 + 2S e_1 \) and \( S(\odot_1 (e_1@x)) = 1 + 2S e_1 \)

then \( S(\odot_1 e_1)@x > S\odot_1 (e_1@x)\),

\( P(d) \)

R4. \((\sum_{i=1}^{i=n} e_1)@x \implies \sum_{i=1}^{i=n} (e_1@x)\)

Since \( S((\sum_{i=1}^{i=n} e_1)@x) = 4 + 4S e_1 \) and \( S(\sum_{i=1}^{i=n} (e_1@x)) = 2 + 4S e_1 \)
then $S(\sum_{i=1}^{n} e_i @ x) > S(\sum_{i=1}^{n} (e_i @ x))$

P(d)

R5. $(\chi) @ x \Rightarrow \chi$

Since $S((\chi) @ x) = 2S(\chi)$ and $S(\chi) = S(\chi)$ then $S((\chi) @ x) > S(\chi)$

P(d)

R6. $\nabla_i \circ (e_1 * e_2) \Rightarrow e_1(\nabla_i \circ e_2) + e_2(\nabla_i \circ e_1)$

$S(\nabla_i \circ (e_1 * e_2)) = s_1 + s_2 + s_3$

where $s_1 = S(e_1 * S^{1+Se_1+Se_2}$, $s_2 = S(e_2 * S^{1+Se_1+Se_2}$, and $s_3 = S^{1+Se_1+Se_2}$,

$S(e_1 \nabla_i \circ e_2 + e_2 \nabla_i \circ e_1) = t_1 + t_2 + t_3$

where $t_1 = S(e_1(5^{Se_1} + 1)$, $t_2 = S(e_2(5^{Se_1} + 1)$, and $t_3 = 3$

Given $4 * S^{1+Se_1} > 1$ then

$\rightarrow 5 * 5^{Se_1} > 5^{Se_1} + 1$ by adding $5^{Se_1}$

$\rightarrow 5^{1+Se_1+Se_2} > 5^{Se_1} + 1$ by refactoring

$\rightarrow S(e_1 * 5^{1+Se_1+Se_2} > S(e_1(5^{Se_1} + 1)$ by multiplying by $S(e_1$

$\rightarrow S(e_2 * 5^{1+Se_1+Se_2} > S(e_2(5^{Se_1} + 1)$ by multiplying by $S(e_2$

and so $s_1 > t_1, s_2 > t_2$

Lastly, $5^{1+Se_1+Se_2} > 3$ (Lm A.3.2) and so $s_3 > t_3$

Finally, $S\nabla_i \circ (e_1 * e_2) > S(e_1 \nabla_i \circ e_2 + e_2 \nabla_i \circ e_1$

P(d)

R7. $\nabla_i \circ (e_1 \over e_2) \Rightarrow \frac{(\nabla_i \circ e_1)e_2 - e_1(\nabla_i \circ e_2)}{e_2^2}$

$S(\nabla_i \circ (e_1 \over e_2)) = s_1 + s_2 + s_3$

where $s_1 = S(e_1 5^{2+Se_1+Se_2}$, $s_2 = S(e_2 5^{2+Se_1+Se_2}$, and $s_3 = 2 * 5^{2+Se_1+Se_2}$

$S(\frac{(\nabla_i \circ e_1)e_2 - e_1(\nabla_i \circ e_2)}{e_2^2}) = t_1 + t_2 + t_3$

where $t_1 = S(e_1(1 + 5^{Se_1})$, $t_2 = S(e_2(3 + 5^{Se_2})$, and $t_3 = 6$
Given $5^{2+S_e1+S_e2} > (1 + 5^{S_e1})(\text{Lm A.3.1})$

then $S_e15^{2+S_e1+S_e2} > S_e1(1 + 5^{S_e1})$ by multiplying by $S_e1$

so $s_1 > t_1, s_2 > t_2$

Given $5^{1+S_e1+S_e2} > 5^{S_e2} + 3$ (Lm A.3.1)

then $2 * 5^{1+S_e1+S_e2} > 2 * 5^{S_e2} + 6$ by multiplying by 2

so $s_3 > t_3$

$S_{\text{source}}(d) > S_{\text{target}}(d)$

P(d)

R8. $\nabla_i \diamond (\sqrt{e_1}) \Rightarrow \text{lift}(1/2) \ast \frac{\nabla_i \diamond e_1}{\sqrt{e_1}}$

Since $S(\nabla_i \diamond (\sqrt{e_1})) = (1 + S_e1)5^{(1+S_e1)}$ and $S(\text{lift}(1/2) \ast \frac{\nabla_i \diamond e_1}{\sqrt{e_1}}) = S_e1(1 + 5^{S_e1}) + 6$

then $S(\nabla_i \diamond (\sqrt{e_1})) > S(\text{lift}(1/2) \ast \frac{\nabla_i \diamond e_1}{\sqrt{e_1}})$ (Lm A.3.3)

P(d)

R9. $\nabla_i \diamond (\cos(e_1)) \Rightarrow (-\sin(e_1)) \ast (\nabla_i \diamond e_1)$

Included in the earlier prose.

R10. $\nabla_i \diamond (\sin(e_1)) \Rightarrow (\cos(e_1)) \ast (\nabla_i \diamond e_1)$

Since $S(\nabla_i \diamond (\sin(e_1))) = (1 + S_e1)5^{(1+S_e1)}$

and $S((\cos(e_1)) \ast (\nabla_i \diamond e_1)) = S_e1(1 + 5^{S_e1}) + 2$

then $S(\nabla_i \diamond (\sin(e_1))) > S((\cos(e_1)) \ast (\nabla_i \diamond e_1))$ (Lm A.3.3)

P(d)

R11. $\nabla_i \diamond (\tan(e_1)) \Rightarrow \frac{\nabla_i \diamond e_1}{\cos(e_1) \ast \cos(e_1)}$

Since $S(\nabla_i \diamond (\tan(e_1))) = (1 + S_e1)5^{(1+S_e1)}$

and $S(\frac{\nabla_i \diamond e_1}{\cos(e_1) \ast \cos(e_1)}) = S_e1(5^{S_e1} + 2) + 5$

then $S(\nabla_i \diamond (\tan(e_1))) > S(\frac{\nabla_i \diamond e_1}{\cos(e_1) \ast \cos(e_1)})$ (Lm A.3.3)
P(d)

R12. \( \nabla_i \diamond (\arccosine(e_1)) \Rightarrow (\frac{-\text{lift}(1)}{\sqrt{(\text{lift}(1)-(e+e)}}) \ast (\nabla_i \diamond e_1) \)

Since \( S(\nabla_i \diamond (\arccosine(e_1)))=(1 + S e_1)^5(1 + S e_1) \),

and \( S((\frac{-\text{lift}(1)}{\sqrt{(\text{lift}(1)-(e+e)}}) \ast (\nabla_i \diamond e_1))=S e_1(2 + 5S e_1) + 11 \),

then \( S(\nabla_i \diamond (\arccosine(e_1))) > S((\frac{-\text{lift}(1)}{\sqrt{(\text{lift}(1)-(e+e)}}) \ast (\nabla_i \diamond e_1)) \) (Lm A.3.3)

P(d)

R13. \( \nabla_i \diamond (\arcsine(e_1)) \Rightarrow (\frac{\text{lift}(1)}{\sqrt{(\text{lift}(1)-(e+e)}}) \ast (\nabla_i \diamond e_1) \)

Since \( S(\nabla_i \diamond (\arcsine(e_1)))=(1 + S e_1)^5(1 + S e_1) \),

and \( S((\frac{\text{lift}(1)}{\sqrt{(\text{lift}(1)-(e+e)}}) \ast (\nabla_i \diamond e_1))=S e_1(2 + 5S e_1) + 10 \),

then \( S(\nabla_i \diamond (\arcsine(e_1))) > S((\frac{\text{lift}(1)}{\sqrt{(\text{lift}(1)-(e+e)}}) \ast (\nabla_i \diamond e_1)) \) (Lm A.3.3)

P(d)

R14. \( \nabla_i \diamond (\arctangent(e_1)) \Rightarrow \frac{\text{lift}(1)}{\text{lift}(1)+(e+e+e)} \ast (\nabla_i \diamond e_1) \)

Since \( S(\nabla_i \diamond (\arctangent(e_1)))=(1 + S e_1)^5(1 + S e_1) \),

and \( S((\frac{1}{\text{lift}(1)+(e+e+e)} \ast (\nabla_i \diamond e_1))=S e_1(2 + 5S e_1) + 9 \),

then \( S(\nabla_i \diamond (\arctangent(e_1))) > S((\frac{1}{\text{lift}(1)+(e+e+e)} \ast (\nabla_i \diamond e_1)) \) (Lm A.3.3)

P(d)

R15. \( \nabla_i \diamond (\exp(e_1)) \Rightarrow \exp(e_1) \ast (\nabla_i \diamond e_1) \)

Since \( S(\nabla_i \diamond (\exp(e_1)))=(1 + S e_1)^5(1 + S e_1) \) and \( S(\exp(e_1) \ast (\nabla_i \diamond e_1))=S e_1(1 + 5S e_1) + 2 \),

then \( S(\nabla_i \diamond (\exp(e_1))) > S(\exp(e_1) \ast (\nabla_i \diamond e_1)) \) (Lm A.3.3)

P(d)

R16. \( \nabla_i \diamond (e_1^n) \Rightarrow \text{lift}(n) \ast e_1^{n-1} \ast (\nabla_i \diamond e_1) \)

Since \( S(\nabla_i \diamond (e_1^n))=(1 + S e_1)^5(1 + S e_1) \) and \( S(\text{lift}(n) \ast e_1^{n-1} \ast (\nabla_i \diamond e_1))=5 + S e_1(1 + 5S e_1) \)
then $S(\nabla_i (e_1^n)) > S(lift(n) * e_1^{n-1} * (\nabla_i \diamond e_1))$ (Lm A.3.3)

P(d)

R17. $\nabla_i(e_1 \odot e_2) \implies (\nabla_i e_1) \odot (\nabla_i e_2)$

Included in the earlier prose.

R18. $\nabla_i(-e_1) \implies -(\nabla_i \diamond e_1)$

Since $S(\nabla_i(-e_1)) = 5^1 + S e_1 (1 + S e_1)$ and $S(-(\nabla_i \diamond e_1)) = 1 + S e_1 5 S e_1$

then $S(\nabla_i(-e_1)) > S(-(\nabla_i \diamond e_1))$ (Lm A.3.1)

P(d)

R19. $\nabla \odot \sum_{v:1}^{v=n} e_1 \implies \sum_{v:1}^{v=n} (\nabla \diamond e_1)$

Since $S(\nabla \odot \sum_{v:1}^{v=n} e_1) = (2 + 2 S e_1) \cdot 5^2 + 2 S e_1$ and $S(\sum_{v:1}^{v=n} (\nabla \diamond e_1)) = 2 + 2 S e_1 5 S e_1$

then $S(\nabla \odot \sum_{v:1}^{v=n} e_1) > S(\sum_{v:1}^{v=n} (\nabla \diamond e_1))$

P(d)

R20. $\nabla \odot \chi \implies 0$

Since $S(\nabla \diamond) = S \chi 5 S \chi$ and $S(0) = 2$ then $S(\nabla \diamond) > S(0)$ (Lm A.3.2)

P(d)

R21. $\nabla_i \odot (V_\alpha \otimes H^\nu) \implies (V_\alpha \otimes h^{i\nu})$

Since $S(\nabla_i \odot (V_\alpha \otimes H^\nu)) = 5$ and $S((V_\alpha \otimes h^{i\nu})) = 1$ then $S(\nabla_i \odot (V_\alpha \otimes H^\nu)) > S((V_\alpha \otimes h^{i\nu}))$

P(d)

R22. $-e_1 \implies e_1$

Since $S(-e_1) = 2 + S e_1$ and $S(e_1) = S e_1$ then $S(-e_1) > S(e_1)$

P(d)
R23. $-0 \implies 0$
Since $S(-0)=2$ and $S(0)=1$ then $S(-0) > S(0)$
P(d)

R24. $e_1 - 0 \implies e_1$
Since $S(e_1 - 0)=2 + S e_1$ and $S(e_1)=S e_1$ then $S(e_1 - 0) > S(e_1)$
P(d)

R25. $0 - e_1 \implies -e_1$
Similar approach to R24
P( R25) OK

R26. $\frac{0}{e_1} \implies 0$
Since $S(\frac{0}{e_1})=3 + S e_1$ and $S(0)=1$ then $S(\frac{0}{e_1}) > S(0)$
P(d)

R27. $\frac{e_1}{e_2} \implies \frac{e_1}{e_2 e_3}$
Included in the earlier prose.

R28. $\frac{e_1}{e_2} \implies \frac{e_1 e_3}{e_2}$
Similar approach to R27
P( R28) OK

R29. $\frac{e_1}{e_2} \implies \frac{e_1 e_4}{e_2 e_3}$
Since $S(\frac{e_1}{e_2})=6 + S e_1 + S e_2 + S e_3$ and $S(\frac{e_1 e_4}{e_2 e_3})=4 + S e_1 + S e_2 + S e_3$
then $S(\frac{e_1}{e_2}) > S(\frac{e_1 e_4}{e_2 e_3})$
P(d)

R30.
\[ e_1, e_1 + 0 \implies e_1 \]
Since \( S(0 + e_1, e_1 + 0) = 2 + Se_1 \) and \( Se_1 = Se_1 \) then \( S(0 + e_1, e_1 + 0) > S(e_1) \)
P(d)

R31.\( e, e_0 \implies 0 \)
Similar approach to R30
P( R31) OK

R32.
\[ \sqrt{(e_1)} \ast \sqrt{(e_1)} \implies e_1 \]
Since \( S(\sqrt{(e_1)} \ast \sqrt{(e_1)}) = 3 + 2Se_1 \) and \( Se_1 = Se_1 \) then \( S(\sqrt{(e_1)} \ast \sqrt{(e_1)}) > S(e_1) \)
P(d)

R33.
\[ E_{ijk} \nabla_{ij} \ast e_1 \implies \text{lift}(0) \]
Since \( S(E_{ijk} \nabla_{ij} \ast e_1) = 5 + 5Se_1 \) and \( S(\text{lift}(0)) = 2 \) then \( S(E_{ijk} \nabla_{ij} \ast e_1) > S(\text{lift}(0)) \)
P(d)

R34.
\[ E_{ijk}(V_\alpha \otimes h^{jk}) \implies \text{lift}(0) \]
Since \( S(E_{ijk}(V_\alpha \otimes h^{jk})) = 6 \) and \( S(\text{lift}(0)) = 2 \) then \( S(E_{ijk}(V_\alpha \otimes h^{jk})) > S(\text{lift}(0)) \)
P(d)

R35.
\[ E_{ijk} \mathcal{E}_{ilm} \implies \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} \]
Since \( S(E_{ijk} \mathcal{E}_{ilm}) = 9 \) and \( S(\delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}) = 7 \) then \( S(E_{ijk} \mathcal{E}_{ilm}) > S(\delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}) \)
P(d)

R36.
\[ \delta_{ij}T_j \implies T_i \]
Since \( S(\delta_{ij}T_j) = 3 \) and \( S(T_i) = 1 \) then \( S(\delta_{ij}T_j) > S(T_i) \)
\( P(d) \)

R37. \( \delta_{ij} F_j \implies F_i \)

Similar approach to R36

\( P(\text{R37}) \text{ OK} \)

R38. \( \delta_{ij} V \otimes H^{\delta_{cj}} \implies V \otimes H^{\delta_{ci}} \)

Similar approach to R36

\( P(\text{R38}) \text{ OK} \)

R39. \( \delta_{ij} V \otimes H^{\delta_{cj}(x)} \implies V \otimes H^{\delta_{ci}(x)} \)

Since \( S(\delta_{ij} V \otimes H^{\delta_{cj}(x)}) = 4 \) and \( S(V \otimes H^{\delta_{ci}(x)}) = 2 \) then \( S(\delta_{ij} V \otimes H^{\delta_{cj}(x)}) > S(V \otimes H^{\delta_{ci}(x)}) \)

\( P(d) \)

R40. \( \delta_{ij} \nabla_j \diamond e_1 \implies \nabla_i \diamond (e_1) \)

Since \( S(\delta_{ij} \nabla_j \diamond (e_1)) = 2 + S e_1 5 S e_1 \) and \( S(\nabla_i \diamond (e_1)) = S e_1 5 S e_1 \) then \( S(\delta_{ij} \nabla_j \diamond (e_1)) > S(\nabla_i \diamond (e_1)) \)

\( P(d) \)

R41. \( \Sigma(se_1) \implies s\Sigma e_1 \)

Since \( S(\Sigma(se_1)) = 6 + 2 S e_1 \) and \( S(s\Sigma e_1) = 4 + 2 S e_1 \) then \( S(\Sigma(se_1)) > S(s\Sigma e_1) \)

\( P(d) \)

\( P(d) \text{ Lemma 4.3.1} \)

\underline{A.3.2 Termination implies Normal Form}

1. **Termination implies normal form** (Lemma 4.3.2).

2. **Proof.** The proof is by examination of the syntax in Figure 2.1. For any syntactic construct,
we show that either the term is in normal form, or there is a rewrite rule that applies (Section A.3.2). We state \( Q(x) \) as a shorthand for the claim that if \( x \) has terminated and is normal form. Additionally we state \( CQ(x) \) if there exists an expression that is not in normal form and has terminated. The following is a proof by contradiction.

\[
\text{Define the following shorthand: } M(e_1) = \sqrt{e_1} | exp(e_1) | e_1^n | \kappa(e_1)
\]

<table>
<thead>
<tr>
<th>Case on Structure</th>
<th>Normal Form</th>
<th>( Q(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = c )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( x = T_\alpha )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( x = F_\alpha )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( x = V_\alpha \otimes H )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( x = \delta_{ij} )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( x = \mathcal{E}_\alpha )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( x = \text{lift}(e_1) )</td>
<td>Incorrect Type</td>
<td>( Q(x) )</td>
</tr>
</tbody>
</table>

Show \( Q(x) \) with proof by contradiction. Assume \( CQ(x) \)

<table>
<thead>
<tr>
<th>Case on Structure</th>
<th>Normal Form</th>
<th>( Q(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( -e )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( \partial \partial_{x_\alpha} \odot e )</td>
<td>Incorrect Type</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( \sum e )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( e \oplus e )</td>
<td>Incorrect Type</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( e + e )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
<tr>
<td>( e - e )</td>
<td>Normal Form</td>
<td>( Q(x) )</td>
</tr>
</tbody>
</table>
Show Q(x) with proof by contradiction. Assume CQ(x)

case on structure $e_1$

Note. $M(e_1) = \exp(e_1) \mid e_1^n \mid \kappa(e_1) \mid \sqrt{e_1}$

c | Normal Form | Q(x)
---|---|---
$c$ | Normal Form | Q(x)
$T_\alpha$ | Normal Form | Q(x)
$F_\alpha$ | Normal Form | Q(x)
$\delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk}$ | Normal Form | Q(x)

$\text{lift}(e)$ | Assume Q(e) then Q(x)

$M(e)$ | Assume Q(e) then Q(x)
$-e$ | Assume Q(e) then Q(x)
$\frac{\partial}{\partial x_\alpha} e$ | Assume Q(e) then Q(x)
$\sum e$ | Assume Q(e) then Q(x)

$e \otimes e$ | Normal Form | Q(x)
$e + e$ | Assume Q(e) then Q(x)
$e - e$ | Assume Q(e) then Q(x)
$e \ast e$ | Assume Q(e) then Q(x)
$\frac{e}{e}$ | Assume Q(e) then Q(x)
$e @ e$ | Assume Q(e) then Q(x)

Q(x)

$x = -e_1$

Show Q(x) with proof by contradiction. Assume CQ(x)

case on structure $e_1$

0 | Apply rule R23 | Q(x)
$c \neq 0$ Normal Form Q(x)

$T_\alpha$ Normal Form Q(x)

$F_\alpha$ Normal Form Q(x)

$\delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk}$ Normal Form Q(x)

\text{lift}(e) \quad \text{Assume Q(e) then Q(x)}

M(e) \quad \text{Assume Q(e) then Q(x)}

-e \quad \text{Apply rule R22 Q(x)}

\frac{\partial}{\partial x_\alpha} \circ e \quad \text{Assume Q(e) then Q(x)}

\sum e \quad \text{Assume Q(e) then Q(x)}

e \odot e \quad \text{Normal Form Q(x)}

e + e \quad \text{Assume Q(e) then Q(x)}

e - e \quad \text{Assume Q(e) then Q(x)}

e * e \quad \text{Assume Q(e) then Q(x)}

\frac{e}{e} \quad \text{Assume Q(e) then Q(x)}

e @ e \quad \text{Assume Q(e) then Q(x)}

Q(x)

x = e_1 + e_2

Prove Q(x)

case on structure $e_1$

0 \quad \text{Apply rule R30 Q(x)}

c \neq 0 \quad \text{Normal Form Q(x)}

T_\alpha \quad \text{Normal Form Q(x)}

F_\alpha \quad \text{Normal Form Q(x)}

\delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk} \quad \text{Normal Form Q(x)}

\text{lift}(e) \quad \text{Assume Q(e) then Q(x)}

M(e) \quad \text{Assume Q(e) then Q(x)}

-e \quad \text{Assume Q(e) then Q(x)}
\[
\frac{\partial}{\partial x_\alpha} e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
\sum e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e \otimes e ~ \text{Normal Form } Q(x)
\]
\[
e + e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e - e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e \ast e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e \div e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e @ e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]

Case on structure \( e_2 \)

Proof same as Lines 76-85

\[
Q(x)
\]

\[
x = e_1 - e_2
\]

Show \( Q(x) \) with proof by contradiction. Assume \( CQ(x) \)

Case on structure \( e_1 \)

\[
0 ~ \text{Apply rule R25 } Q(x)
\]
\[
c \ne 0 ~ \text{Normal Form } Q(x)
\]
\[
T_\alpha ~ \text{Normal Form } Q(x)
\]
\[
F_\alpha ~ \text{Normal Form } Q(x)
\]
\[
\delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk} ~ \text{Normal Form } Q(x)
\]
\[
\text{lift}(e) ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
\text{M}(e) ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
-e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
\frac{\partial}{\partial x_\alpha} e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
\sum e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e \otimes e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e + e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]
\[
e - e ~ \text{Assume } Q(e) \text{ then } Q(x)
\]

194
Assume $Q(e)$ then $Q(x)$

Assume $Q(e)$ then $Q(x)$

Assume $Q(e)$ then $Q(x)$

case on structure $e_2$

0 Apply rule R24 $Q(x)$

Proof same as Lines 99-107

$Q(x)$

$x = e_1 * e_2$

Show $Q(x)$ with proof by contradiction. Assume $CQ(x)$

case on structure $e_1$

0 Apply rule R31 $Q(x)$

c $\neq 0$ Normal Form $Q(x)$

$T_\alpha$ Normal Form $Q(x)$

$F_\alpha$ Normal Form $Q(x)$

$\delta_{ij}$

case on structure $e_2$

$T_j$ Apply rule R36 $Q(x)$

$F_j$ Apply rule R37 $Q(x)$

$V \oplus H_j$ Apply rule R38 $Q(x)$

$V \oplus H_j @ e$ Apply rule R39 $Q(x)$

$\frac{\partial}{\partial x_j}(e)$ Apply rule R40 $Q(x)$

otherwise Assume $Q(e)$ then $Q(x)$

$\mathcal{E}_{ij}$ Assume $Q(e)$ then $Q(x)$

$\mathcal{E}_{ijk}$

case on structure $e_2$

$\frac{\partial}{\partial x_{ij}}(e)$ Apply rule R33 $Q(x)$

$V \oplus H_{jk}$ Apply rule R34 $Q(x)$
$E_{ijk}$  
Apply rule R35  
$Q(x)$

otherwise  
Assume $Q(e)$ then $Q(x)$

lift$(e)$  
Assume $Q(e)$ then $Q(x)$

$M(e)$

case on structure $M(e)$

$\sqrt{e_3}$

case on structure $e_2$

$\sqrt{e_3}$  
Apply rule R32  
$Q(x)$

otherwise  
Assume $Q(e)$ then $Q(x)$

otherwise  
Assume $Q(e)$ then $Q(x)$

$-e$  
Assume $Q(e)$ then $Q(x)$

$\frac{\partial}{\partial x_\alpha} \circ e$  
Assume $Q(e)$ then $Q(x)$

$\sum e$  
Assume $Q(e)$ then $Q(x)$

e $\oplus e$  
Assume $Q(e)$ then $Q(x)$

e + e  
Assume $Q(e)$ then $Q(x)$

e - e  
Assume $Q(e)$ then $Q(x)$

e $\ast e$  
Assume $Q(e)$ then $Q(x)$

e/e  
Assume $Q(e)$ then $Q(x)$

e@@e  
Assume $Q(e)$ then $Q(x)$

$Q(x)$

$x=\frac{e_1}{e_2}$

Show $Q(x)$ with proof by contradiction. Assume C$Q(x)$

case on structure $e_1$

$\frac{e_2}{e_3}$

case on structure $e_2$

$\frac{e_4}{e_5}$  
Apply rule R27  
$Q(x)$

otherwise  
Apply rule R29  
$Q(x)$
0 Apply rule R26 Q(x)

\( c \neq 0 \) Normal Form Q(x)

\( T_\alpha \) Normal Form Q(x)

\( F_\alpha \) Normal Form Q(x)

\( \delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk} \) Normal Form Q(x)

\( \frac{\partial}{\partial x_\alpha} \diamond e \) Assume Q(e) then Q(x)

\( \sum e \) Assume Q(e) then Q(x)

\( \text{lift}(e) \) Assume Q(e) then Q(x)

\( M(e) \) Assume Q(e) then Q(x)

\( -e \) Assume Q(e) then Q(x)

\( e \odot e \) Assume Q(e) then Q(x)

\( e + e \) Assume Q(e) then Q(x)

\( e - e \) Assume Q(e) then Q(x)

\( e * e \) Assume Q(e) then Q(x)

\( e \odot e \) Assume Q(e) then Q(x)

case on structure \( e_2 \)

\( \frac{e_4}{e_5} \) Apply rule R28 Q(x)

otherwise proof same as Lines 165-171

Q(x)

\( x = e_1 \odot e_2 \)

Show Q(x) with proof by contradiction. Assume CQ(x)

case on structure \( e_1 \)

\( c \) Incorrect Type Q(x)

\( T_\alpha \) Incorrect Type Q(x)

\( F_\alpha \) Assume Q(e) then Q(x)

\( \delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk} \) Apply rule R5 Q(x)

Lift(e) Apply rule R5 Q(x)
Apply rule R3 Q(x)

Apply rule R3 Q(x)

Assume Q(e) then Q(x)

Apply rule R4 Q(x)

Assume Q(e) then Q(x)

Apply rule R2 Q(x)

Apply rule R2 Q(x)

Apply rule R1 Q(x)

Apply rule R1 Q(x)

Incorrect Type Q(x)

Q(x)

Show Q(x) with proof by contradiction. Assume CQ(x)

case on structure e_1

c Incorrect Type Q(x)

T_α Incorrect Type Q(x)

F_α Normal Form Q(x)

δ_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk} Apply rule R20 Q(x)

M(e)

case on structure M(e)

Cosine(e) Apply rule R9 Q(x)

Sine(e) Apply rule R10 Q(x)

Tangent(e) Apply rule R11 Q(x)

ArcCosine(e) Apply rule R12 Q(x)

ArcSine(e) Apply rule R13 Q(x)

ArcTangent(e) Apply rule R14 Q(x)

exp(e) Apply rule R15 Q(x)
e^n  Apply rule R16  Q(x)

\sqrt{e}  Apply rule R8  Q(x)

\text{lift}(e)  Apply rule R20  Q(x)

-e  Apply rule R18  Q(x)

\frac{\partial}{\partial x_\alpha} \circ e  Apply rule R42  Q(x)

\sum e  Apply rule R19  Q(x)

e \otimes e  Apply rule R21  Q(x)

e + e  Apply rule R17  Q(x)

e - e  Apply rule R17  Q(x)

e \ast e  Apply rule R6  Q(x)

\frac{e}{e}  Apply rule R7  Q(x)

e @ e  Incorrect Type  Q(x)

Q(x)

x=\sum(e_1)

Show Q(x) with proof by contradiction. Assume CQ(x)

case on structure e_1

c  Apply rule R41  Q(x)

T  Apply rule R41  Q(x)

T_\alpha  Normal Form  Q(x)

F  Apply rule R41  Q(x)

F_\alpha  Normal Form  Q(x)

V \otimes H  Apply rule R41  Q(x)

\delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk}  Normal Form  Q(x)

\text{lift}(e)  Assume Q(e) then Q(x)

M(e)  Assume Q(e) then Q(x)

-e  Assume Q(e) then Q(x)

\frac{\partial}{\partial x_\alpha} \circ e  Assume Q(e) then Q(x)
\[ \sum e \] Assume \( Q(e) \) then \( Q(x) \)

\[ e \odot e \] Assume \( Q(e) \) then \( Q(x) \)

\[ e + e \] Assume \( Q(e) \) then \( Q(x) \)

\[ e - e \] Assume \( Q(e) \) then \( Q(x) \)

\[ e \ast e \] Assume \( Q(e) \) then \( Q(x) \)

\[ \frac{e}{e} \] Assume \( Q(e) \) then \( Q(x) \)

\[ F \hat{\ast} e \] Apply rule R41 \( Q(x) \)

\[ V \hat{\ast} h \hat{\ast} e \] Apply rule R41 \( Q(x) \)

\[ e \hat{\ast} e \] Assume \( Q(e) \) then \( Q(x) \)

\[ Q(x) \]

\[ Q(x) \) (Lemma 4.3.2) \]

\[ \text{A.3.3 Normal Form implies Termination} \]

The section offers a proof for Lemma 4.3.3.

**Non-terminated** A term has not terminated if it is the source term of a rewrite rule.

**Normal form implies Termination.** (Lemma 4.3.3).

**Proof.** We state \( M(e) \) as a shorthand for the claim that if \( e \) is in normal form then it has terminated. The following is a proof by contradiction. \( CM(e) \): There exists an expression \( e \) that has not terminated and is in normal form. More precisely, given a derivation \( d \) of the form \( e \longrightarrow e' \), there exists an expression that is the source term \( e \) of derivation \( d \) therefore not-terminated, and is in normal form. \( \square \)

Case analysis on the source of each rule

\[ R1. (e_1 \odot_n e_2) \hat{\ast} x \implies (e_1 \hat{\ast} x) \odot_n (e_2 \hat{\ast} x) \]

Let \( y = (e_1 \odot_n e_2) \hat{\ast} x \) and since \( y \) is not in normal form then \( M( R1) \) OK
R2. \((e_0 \otimes_2 e_1)@x \implies (e_0@x) \otimes_2 (e_1@x)\)

Let \(y = (e_0 \otimes_2 e_1)@x\) and since \(y\) is not in normal form then \(M(\text{ R2})\) OK

R3. \((\circ_1 e_1)@x \implies \circ_1(e_1@x)\)

Let \(y = (\circ_1 e_1)@x\) and since \(y\) is not in normal form then \(M(\text{ R3})\) OK

R4. \((\sum_{i=1}^{n} e_1)@x \implies \sum_{i=1}^{n}(e_1@x)\)

Let \(y = (\sum_{i=1}^{n} e_1)@x\) and since \(y\) is not in normal form then \(M(\text{ R4})\) OK

R5. \((\chi)@x \implies \chi\)

Let \(y = (\chi)@x\) and since \(y\) is not in normal form then \(M(\text{ R5})\) OK

R6. \(\nabla_i \diamond (e_1 \ast e_2) \implies e_1(\nabla_i \diamond e_2) + e_2(\nabla_i \diamond e_1)\)

Let \(y = \nabla_i \diamond (e_1 \ast e_2)\) and since \(y\) is not in normal form then \(M(\text{ R6})\) OK

R7. \(\nabla_i \diamond \left(\frac{e_1}{e_2}\right) \implies \frac{(\nabla_i \circ e_1) e_2 - e_1(\nabla_i \circ e_2)}{e_2^2}\)

Let \(y = \nabla_i \diamond \left(\frac{e_1}{e_2}\right)\) and since \(y\) is not in normal form then \(M(\text{ R7})\) OK

R8. \(\nabla_i \diamond (\sqrt{e_1}) \implies \text{lift}(1/2) \ast \frac{\nabla_i \circ e_1}{\sqrt{e_1}}\)

Let \(y = \nabla_i \diamond (\sqrt{e_1})\) and since \(y\) is not in normal form then \(M(\text{ R8})\) OK

R9. \(\nabla_i \diamond (\cosine(e_1)) \implies (-\sin(e_1)) \ast (\nabla_i \circ e_1)\)

Let \(y = \nabla_i \diamond (\cosine(e_1))\) and since \(y\) is not in normal form then \(M(\text{ R9})\) OK

R10. \(\nabla_i \diamond (\sin(e_1)) \implies (\cosine(e_1)) \ast (\nabla_i \circ e_1)\)

Let \(y = \nabla_i \diamond (\sin(e_1))\) and since \(y\) is not in normal form then \(M(\text{ R10})\) OK
R11. \( \nabla_i \diamond (\text{tangent}(e_1)) \) \( \mapsto \frac{\nabla_i \diamond e_1}{\cos(e_1) \ast \cos(e_1)} \)

Let \( y = \nabla_i \diamond (\text{tangent}(e_1)) \) and since \( y \) is not in normal form then \( M( \text{R11}) \) OK

R12. \( \nabla_i \diamond (\text{arccosine}(e_1)) \) \( \mapsto \frac{-\text{lift}(1)}{\sqrt{\text{lift}(1) - (e_1 \ast e_1)}} \ast (\nabla_i \diamond e_1) \)

Let \( y = \nabla_i \diamond (\text{arccosine}(e_1)) \) and since \( y \) is not in normal form then \( M( \text{R12}) \) OK

R13. \( \nabla_i \diamond (\text{arcsine}(e_1)) \) \( \mapsto \frac{\text{lift}(1)}{\sqrt{\text{lift}(1) - (e_1 \ast e_1)}} \ast (\nabla_i \diamond e_1) \)

Let \( y = \nabla_i \diamond (\text{arcsine}(e_1)) \) and since \( y \) is not in normal form then \( M( \text{R13}) \) OK

R14. \( \nabla_i \diamond (\text{arctangent}(e_1)) \) \( \mapsto \frac{\text{lift}(1)}{\text{lift}(1) + (e_1 \ast e_1)} \ast (\nabla_i \diamond e_1) \)

Let \( y = \nabla_i \diamond (\text{arctangent}(e_1)) \) and since \( y \) is not in normal form then \( M( \text{R14}) \) OK

R15. \( \nabla_i \diamond (\exp(e_1)) \) \( \mapsto \exp(e_1) \ast (\nabla_i \diamond e_1) \)

Let \( y = \nabla_i \diamond (\exp(e_1)) \) and since \( y \) is not in normal form then \( M( \text{R15}) \) OK

R16. \( \nabla_i \diamond (e_1^n) \) \( \mapsto \text{lift}(n) \ast e_1^{n-1} \ast (\nabla_i \diamond e_1) \)

Let \( y = \nabla_i \diamond (e_1^n) \) and since \( y \) is not in normal form then \( M( \text{R16}) \) OK

R17. \( \nabla_i (e_1 \odot e_2) \) \( \mapsto (\nabla_i e_1) \odot (\nabla_i e_2) \)

Let \( y = \nabla_i (e_1 \odot e_2) \) and since \( y \) is not in normal form then \( M( \text{R17}) \) OK

R18. \( \nabla_i (-e_1) \) \( \mapsto -(\nabla_i \diamond e_1) \)

Let \( y = \nabla_i (-e_1) \) and since \( y \) is not in normal form then \( M( \text{R18}) \) OK

R19. \( \nabla \diamond \sum_{v:1}^{v=n} e_1 \) \( \mapsto \sum_{v:1}^{v=n} (\nabla \diamond e_1) \)

Let \( y = \nabla \diamond \sum_{v:1}^{v=n} e_1 \) and since \( y \) is not in normal form then \( M( \text{R19}) \) OK
\[ \nabla \text{lift}(e_1) \Rightarrow 0 \]

Let \( y = \nabla \text{Lift}(e_1) \) and since \( y \) is not in normal form then  \( M( R20) \) OK

\[ R20. \nabla \diamond \chi \Rightarrow 0 \]

Let \( y = \nabla \diamond \) and since \( y \) is not in normal form then  \( M( R20) \) OK

\[ R21. \nabla_i \diamond (V_\alpha \otimes H^\nu) \Rightarrow (V_\alpha \otimes h^{i\nu}) \]

Let \( y = \nabla_i \diamond (V_\alpha \otimes H^\nu) \) and since \( y \) is not in normal form then  \( M( R21) \) OK

\[ R22. - - e_1 \Rightarrow e_1 \]

Let \( y = - - e_1 \) and since \( y \) is not in normal form then  \( M( R22) \) OK

\[ R23. - 0 \Rightarrow 0 \]

Let \( y = - 0 \) and since \( y \) is not in normal form then  \( M( R23) \) OK

\[ R24. e_1 - 0 \Rightarrow e_1 \]

Let \( y = e_1 - 0 \) and since \( y \) is not in normal form then  \( M( R24) \) OK

\[ R25. 0 - e_1 \Rightarrow -e_1 \]

Let \( y = 0 - e_1 \) and since \( y \) is not in normal form then  \( M( R25) \) OK

\[ R26. \frac{0}{e_1} \Rightarrow 0 \]

Let \( y = \frac{0}{e_1} \) and since \( y \) is not in normal form then  \( M( R26) \) OK

\[ R27. \frac{e_1}{e_3} \Rightarrow \frac{e_1 e_3}{e_2 e_4} \]

Let \( y = \frac{e_1}{e_3} \) and since \( y \) is not in normal form then  \( M( R27) \) OK
R28. $\frac{e_1}{e_3} \Rightarrow e_1 e_3 \frac{e_2}{e_3}$

Let $y = \frac{e_1}{e_3}$ and since $y$ is not in normal form then $M(\text{R28})$ OK

R29. $\frac{e_1}{e_3} \Rightarrow e_1 e_3 \frac{e_2}{e_4}$

Let $y = \frac{e_1}{e_3}$ and since $y$ is not in normal form then $M(\text{R29})$ OK

R30. $0 + e_1, e_1 + 0 \Rightarrow e_1$

Let $y = 0 + e_1, e_1 + 0$ and since $y$ is not in normal form then $M(\text{R30})$ OK

R31. $0, e_0 \Rightarrow 0$

Let $y = 0, e_0$ and since $y$ is not in normal form then $M(\text{R31})$ OK

R32. $\sqrt{(e_1)} \times \sqrt{(e_1)} \Rightarrow e_1$

Let $y = \sqrt{(e_1)} \times \sqrt{(e_1)}$ and since $y$ is not in normal form then $M(\text{R32})$ OK

R33. $E_{ijk} \nabla_{ij} \diamond e_1 \Rightarrow \text{lift}(0)$

Let $y = E_{ijk} \nabla_{ij} \diamond e_1$ and since $y$ is not in normal form then $M(\text{R33})$ OK

R34. $E_{ijk}(V_{\alpha} \odot h^{jk}) \Rightarrow \text{lift}(0)$

Let $y = E_{ijk}(V_{\alpha} \odot h^{jk})$ and since $y$ is not in normal form then $M(\text{R34})$ OK

R35. $E_{ijk} E_{ilm} \Rightarrow \delta_{ji} \delta_{km} - \delta_{jm} \delta_{kl}$

Let $y = E_{ijk} E_{ilm}$ and since $y$ is not in normal form then $M(\text{R35})$ OK

R36. $\delta_{ij} T_j \Rightarrow T_i$

Let $y = \delta_{ij} T_j$ and since $y$ is not in normal form then $M(\text{R36})$ OK
R37. $\delta_{ij} F_j \Rightarrow F_i$

Let $y = \delta_{ij} F_j$ and since $y$ is not in normal form then $M(\text{ R37})$ OK

R38. $\delta_{ij} V \otimes H^{\delta_{cj}} \Rightarrow V \otimes H^{\delta_{ci}}$

Let $y = \delta_{ij} V \otimes H^{\delta_{cj}}$ and since $y$ is not in normal form then $M(\text{ R38})$ OK

R39. $\delta_{ij} V \otimes H^{\delta_{cj}}(x) \Rightarrow V \otimes H^{\delta_{ci}}(x)$

Let $y = \delta_{ij} V \otimes H^{\delta_{cj}}(x)$ and since $y$ is not in normal form then $M(\text{ R39})$ OK

R40. $\delta_{ij} \nabla_j \diamond e_1 \Rightarrow \nabla_i \diamond (e_1)$

Let $y = \delta_{ij} \nabla_j \diamond (e_1)$ and since $y$ is not in normal form then $M(\text{ R40})$ OK

R41. $\Sigma(se_1) \Rightarrow s\Sigma e_1$

Let $y = \Sigma(se_1)$ and since $y$ is not in normal form then $M(\text{ R41})$ OK

R42. $\nabla_\alpha \diamond \nabla_\beta \diamond e_1 \Rightarrow \nabla_{\beta \alpha} \diamond e_1$

Let $y = \nabla_\alpha \diamond \nabla_\beta \diamond e_1$ and since $y$ is not in normal form then $M(\text{ R42})$ OK

M(x) Lemma 4.3.3