THE UNIVERSITY OF CHICAGO

IMPLEMENTING MATHEMATICAL EXPRESSIVENESS IN 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

LIST OF TABLES

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 a being a role model and keeping her office door open. Thank you to Ginny McSwain for being an encouraging REU advisor. Thank you to Clifton Presser for the fun programming projects. 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, who drove halfway across the country to move me into my apartment in Chicago. I would like to thank Robert Singer and Anna Olson 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 the National Science Foundation under awards CCF-1446412 and CCF-1564298. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

ABSTRACT

This dissertation describes the implementation of the mathematical expressiveness in the Diderot programming language. Diderot is a domain-specific language for scientific visualization and image analysis. The datasets are produced by digital imaging technologies that sample physical objects at discrete points. Algorithms in this domain are used to visually explore the data and compute features and properties. Diderot is designed to enable the translation of visualization ideas into code by providing a mathematically familiar syntax and high-level language.

The work in this dissertation enables a high-level of programmability by designing and implementing our intermediate representation and addressing the technical challenges that arise. We evaluate the correctness of our implementation with two complimenting parts; formalizing the properties in our rewrite system and automated testing of our implementation. Lastly, we take the first step to extend the language to another domain. As a result, the Diderot user can write intuitive code, compile programs with complicated tensor math, and believe in the correctness of the compiler.

It is important and challenging work to improve the expressivity of the language. We rely on the expressivity of the Diderot language to support the implementation of visualization ideas. With a richer language we can push the boundaries for the type of programs written with the Diderot language.

CHAPTER 1

INTRODUCTION

The work in this dissertation has been fundamental to the development of Diderot. We provide a way to implement the mathematical expressiveness in the high-level programming language. We present our implementation techniques to solve compilation issues. We provide a way to apply automated testing to a high-level DSL based on a ground truth.

Diderot is a parallel domain-specific language for the analysis and visualization of multidimensional scientific images. The data for these images can be created by CT and MRI scanners. Diderot represents these datasets as tensor fields. 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 that can be directly manipulated. Diderot is designed to support algorithms that are based on differential tensor calculus and supports a higher-order mathematical model.

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 because of memory limits. 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 validate the full compiler pipeline 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 an automated testing model for implementing property-based testing. It successfully generates and evaluates thousands of Diderot programs based on a ground-truth.

The computational core of a visualization program written in Diderot is independent of the source of the data. Fields, however, could only be defined by the convolution between discrete image data and a kernel. If Diderot could define other types of fields then we could apply visualization programs to other types of data. Diderot could then be used to debug and visualize fields created by another domain. We describe 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 [15, 16, 18, 44]. In this section, we provide an overview of the computational core of the Diderot language and its compiler.

1.1.1 The Diderot Language

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 "**tensor**$[d_1, \ldots, d_n]$" is an nth-order tensor in $\mathbb{R}^{d_1} \times \cdots \times \mathbb{R}^{d_n}$; we refer to $d_1, \ldots, d_n$ as the *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 eigenvalues *etc.* Diderot's expression syntax is designed to look similar to 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 [42]. In visualization algorithms, tensor fields serve as a mathematical abstraction of the data sets produced by various digital imaging 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.

Figure 1.1 provides an example of a Diderot program used to measure surface curvature. *Curvature* of a surface is defined by the relationship between small positional changes on the surface and changes in the surface normal [45]. 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. The Hessian is used to compute the principles of curvature $k1$, and $k2$. The Diderot types are described in more detail in the following text.

1. The exclusive internal use of the orthonormal elementary basis for representing tensors means that covariant and contra-variant indices can be treated equally.

```
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] P = identity[3] - norm⊗norm;
            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 [45] from paper[18].

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 with shape σ. For example, a 3D grayscale image is "**image**(3)[]. We use convolution ($\circledast$) 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)[$d_1 \ldots d_n$]" is a C^k continuous function (*i.e.*, we can apply up to k levels of differentiation) in $\mathbb{R}^d \rightarrow \mathbb{R}^{d_1} \times \cdots \times \mathbb{R}^{d_n}$. Note, empty brackets '[ ]' indicate a scalar or scalar field.

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:

```
field#2(3)[]F = bspln3 ⊛ img;
```

The field F is reconstructed using the bspln3 kernel from the file `img1.nrrd`.[2] 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 **tensor**[3]), then F(p) will evaluate to a scalar (since F is a scalar field).

Definition 1 (*lifted* operators)**.** *We call operators either tensor or field operators. Field operators include differentiation and probe while tensor operators compromise the rest* ($\bullet, \times, +, \ldots$)*. A P operator is tensor operator applied to tensors and returns a tensor argument (P: tensor $\longrightarrow$ tensor). A $P^{\uparrow}$, or lifted operator is tensor operator applied to fields and returns a field argument ($P^{\uparrow}(f) = \lambda x.P(f(x))$ where $P^{\uparrow}$: field $\longrightarrow$ 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 the addition tensor operator 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 supports the standard differentiation operators on fields (∇, $\nabla\otimes$, and a restricted form of $\nabla\times$).

While the original implementation of Diderot supported tensors and fields, it limited the operations that can be applied to tensor fields. Our work in the IR (Chapter 2) removes this limitation. Tensor operators ($\times, \otimes, \bullet, :$,trace,transpose, ...) can now be applied between tensor fields and between a tensor field and a tensor. New (also *lifted*) tensor operators (inv,det,normalize,sine, ...) and field operators ($\nabla\bullet$ and a more flexible $\nabla\times$) have been added to the language.

The operations supported in Diderot serve as ingredients to the computations created in a visualization program. Differentiated and manipulated tensor fields are used to measure

2. We use the Teem library's Nrrd file format to represent multidimensional data sets (both input and output) [71].

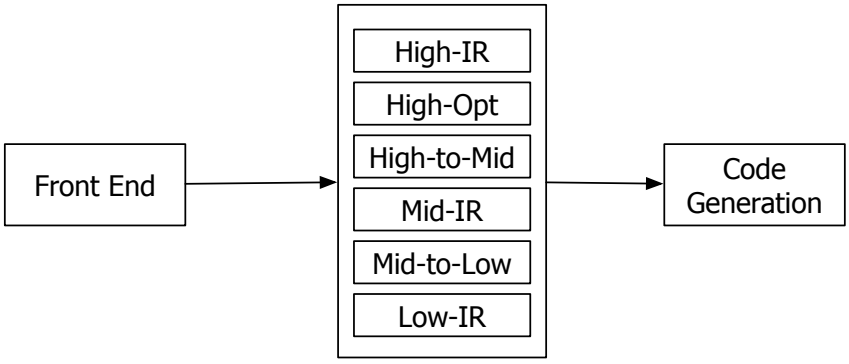

Figure 1.2: Diderot pipeline: Front end, optimization and lowering, and code generation.

geometric features from images. In the earlier example, the Hessian of the field F is used to compute curvature:

```
tensor[3,3] H = ∇⊗∇ F(pos);.
```

The richer Diderot language has enabled the prototyping of new Diderot programs. Later in this dissertation we describe some examples of visualization features enabled by Diderot's new expressivity (Section 7.2).

1.1.2 The Diderot Compiler

The Diderot compiler is organized into three main phases: the front-end, optimization and lowering, and code generation (Figure 1.2). This dissertation is primarily concerned with the optimization 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 [25] 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 directly 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 a standalone 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.[3] The Diderot user declares a field creating a declarative structure for the field expression. The computational expressions sample the field. Translating computational field

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.

operations into executable code is one of the central challenges of the Diderot compiler.

1.2.1 Background

In this section, we describe some additional mathematical concepts used by Diderot. We define some specific operators and their properties. These concepts are used in the following description about tensor fields and in other parts of the dissertation.

The permutation tensor or Levi-Civita tensor is represented in EIN with $\mathcal{E}_{ij}$ and $\mathcal{E}_{ijk}$ for the 2-d and 3-d case, respectively.

$$\mathcal{E}_{ij} = \begin{cases} +1 & ij \text{ is (0,1)} \\ -1 & ij \text{ is (1,0)} \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \mathcal{E}_{ijk} = \begin{cases} +1 & ijk \text{ is cyclic (0,1,2)} \\ -1 & ijk \text{ is anti-cyclic (2,1,0)} \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

The kronecker delta function is δ_{ij}.

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

The Krnocker delta value has the following property when two deltas share an index:

$$\delta_{ik}\delta_{kj} = \delta_{ij} \tag{1.3}$$

and the following when the indices are equal:

$$\delta_{ii} = 3 \tag{1.4}$$

We reflect on the following properties that hold in an orthonormal basis [39]. Let us define an orthonormal basis β with unit basis vectors as $b_i, b_j, \ldots$. Each basis vector is linearly

independent and normalized such that

$$\delta_{ij} \quad = \quad b_i \cdot b_j \quad = \quad \begin{cases} 1 & \text{i=j} \\ 0 & \text{otherwise} \end{cases} \tag{1.5}$$

Any vector **u** can be defined by a linear combination of these basis vectors.

$$\mathbf{u} = \sum_i u_i b_i$$

A component of a tensor can be expressed in the following way

$$u_j = \mathbf{u} \cdot b_j \tag{1.6}$$

1.2.2 Concepts

In this section, we given an informal description of the basic techniques used to implement the translation from computational fields to executable code.

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 over multiple arguments.

$$H(x, y) = h(x)h(y) \text{in 2D}$$

Probing the field F at a point $\mathbf{p}$ involves mapping $\mathbf{p}$ 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) [18].

Let us assume that F is a 2D field; then the probe $F(\mathbf{p})$ can be computed as

$$(V \circledast H)(\mathbf{p}) = \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} (V[\mathrm{n}_0 + i, \mathrm{n}_1 + j]h(\mathrm{f}_0 - i)h(\mathrm{f}_1 - j))$$

where the *support* of the kernel H is $2s$, M is a matrix for array orientation, $\mathbf{x}$ is $\mathbf{p}$ mapped to V's coordinate system (image space) using M, $\mathbf{n} = \lfloor \mathbf{x} \rfloor, \mathbf{n} = (\mathrm{n}_0, \mathrm{n}_1)$, $\mathbf{f} = \mathbf{x} - \mathbf{n}$, and $\mathbf{f} = (\mathrm{f}_0, \mathrm{f}_1)$.

Differentiating Tensor fields We use the notation $\nabla^{(i)}$ to denote i levels of differentiation, where $i > 0$ [18]. The superscript indicates the level of differentiation, *e.g.*, the term $\nabla^{(2)}$ indicates the second derivative (Hessian) and not the Laplacian (or the Trace of the Hessian). A rewrite rule is used to track levels of differentiation.

$$\nabla(V \circledast \nabla^{(i)} H) \longrightarrow_{direct-style} V \circledast \nabla^{(i+1)} H \tag{1.7}$$

In the Diderot language a field expression ∇F is probed at a position $\nabla F(\mathbf{p})$. We can normalize the expression using direct-style operators as follows:

$$\nabla F(\mathbf{p}) \longrightarrow_{direct-style} \nabla((V \circledast H)(\mathbf{p})) \longrightarrow_{direct-style} (V \circledast (\nabla H))(\mathbf{p})$$

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 of probing a differentiated tensor field is in world-space ($[\nabla\varphi]_{\mathcal{W}}$) but needs to be in index-space ($[\nabla\varphi]_{\mathcal{B}}$). This transformation is done by multiplying the result by the inverse transpose of the transformation matrix (M^{-T}). The following provides the mathematical background for this transformation. Consider vector g made up of directional derivatives d:

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

Consider a component of g

$$\begin{aligned} g &= \nabla\varphi \cdot d_i \\ &= \tfrac{\partial}{\partial x_j} \diamond \varphi \cdot (b_j(b_j \cdot d_i)) && \text{use orthonormality of } \mathcal{B} \text{ to rewrite.} \\ &= \sum_j (\tfrac{\partial}{\partial x_j} \diamond \varphi \cdot b_j)(d_i \cdot b_j) && \text{by associativity of multiplication} \\ &= [d_i \cdot b_1 \;\; d_i \cdot b_2 \;\; d_i \cdot b_3] \begin{bmatrix} \nabla\varphi \cdot b_1 \\ \nabla\varphi \cdot b_2 \\ \nabla\varphi \cdot b_3 \end{bmatrix} && \text{by unfolding summation index j} \end{aligned}$$

The vector form

$$\begin{aligned} \vec{g} &= \begin{bmatrix} d_1 \cdot b_1 & d_1 \cdot b_2 & d_1 \cdot b_3 \\ d_2 \cdot b_1 & d_2 \cdot b_2 & d_2 \cdot b_3 \\ d_3 \cdot b_1 & d_3 \cdot b_2 & d_3 \cdot b_3 \end{bmatrix} \begin{bmatrix} \nabla\varphi \cdot b_1 \\ \nabla\varphi \cdot b_2 \\ \nabla\varphi \cdot b_3 \end{bmatrix} \\ &= \mathrm{M}^T [\nabla\varphi]_{\mathcal{B}} && \text{by variable substitution for matrix} \\ (\mathrm{M}^{-T})\vec{g} &= [\nabla\varphi]_{\mathcal{B}} && \text{by dividing both sides by M} \end{aligned}$$

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

1.2.3 Field Normalization

Part of the challenge of supporting a rich language is the complexity of the field definitions. A Diderot user can define a tensor field either with image data and a convolution kernel or as the result of some (simple or complicated) computation. The computations can involve a mix of different operators and arguments making the resulting expression inside the compiler more complicated.

Our solution is to apply a process we call field normalization, to simplify expressions

inside the compiler. 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 the higher-order expression $(F + G)$ to a first-order sum of tensors. The fields terms are then in a simple recognizable form (Definition 2 on page 12).

Definition 2 (Recognizable field term)**.** *A normalized or recognizable field term is a simple form that represents a probed field* ($F(x)$ *or* $(\nabla^{(i)}F)(x)$)*. In this form a probe operator is wrapped around a field variable and there are no tensor operators.*

1.3 Contributions

In this section, we describe our contributions organized by area. We improved the programmability and the expressivity of the language by supporting a higher level of math. We will describe the formal semantics for our core rewriting system and our rigorous automated testing model to test the new features in the language. We also provide the first step to extending Diderot to another domain.

1.3.1 Implementing mathematical expressiveness

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 of Diderot programs. EIN expressions are used to concisely represent field reconstruction and operations on and between

tensors 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 a 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 high-level Diderot code 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 these compilation issues, we developed a number of implementation techniques to 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 cannot only compile more programs than previously possible, but also it compiles faster and produces 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 define the typing judgments on an EIN structure. We show that the rewriting system is type preserving (Theorem 4.1.1). We show that the rewriting system is value preserving (Theorem 4.2.1) for the tensor valued rules. Chapter 4 provides the details for these proofs. We show that the rewriting system is terminating. We define a size metric on the structure on an EIN expression. The rewriting system always decrease the size of an expression (Lemma 4.3.1). We define a subset of the EIN expressions to be *normal form*. We show that termination implies normal form (Lemma 4.3.2) and that normal form implies termination (Lemma 4.3.3). For any expression we can apply rewrites until termination, at which point we will have reached a normal form expression (Theorem 4.3.4).

Automated Testing model

We present *DATm*, Diderot's automated testing model to check the correctness of the core operations in the programming language [17]. *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. The model has found numerous bugs that are deep in the compiler and only could be triggered with a unique combination of operations. Lastly, we demonstrate testing in the context of a visualization program.

1.3.3 Extension of the language

We demonstrate our first and modest approach of visualizing FEM data with Diderot 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 for FEM. In addition, it provides motivation for future work. Chapter 6 describes the extension of the Diderot language to include FEM data.

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 most natural to reason about these concepts using a mathematical notation rather than in lower-level code. For instance, we might write operations between two fields as $c = a + b$ while intending to compute $c[x] = a[x] + b[x]$. The tensor operation $(+)$ is *lifted* to operate between tensor fields. Writing computations using *lifted* operations is intuitive and easy. Writing these computations with lower-level code can be difficult and tedious.

Diderot eases the transformation of mathematical ideas into workable code by allowing the math-like notation to be written directly in the language. Our work generalizes 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, we can enable the implementation of more complicated computations. For example, Crest Lines are places were the surface curvature is maximal along the curvature direction [51]. To get maximum, we need at least one level of differentiation (maximal is where the derivative is 0). We can find the crest lines by taking the Hessian of these principles (fourth derivative overall). Since crest lines and curvature are related computations, we should be able to build on the previous Diderot program (Figure 1.1). For conciseness we will refer to one part of the program (the definition of k1 and omit the rest).

```
real k1 = (trace(G) + disc)/2.0;
```

To compute crest lines, the value k1 is differentiated twice (among applying other tensor operations). If we insert the necessary computations directly into the Diderot program

```
real out = ∇⊗∇... k1;
```

it is not permitted (since k1 is a **real** and not a **field**). The easiest solution is to redefine these values (k1 and some preceding tensors) as field types.

```
field#k(d)[] k1 = (trace(G) + disc)/2.0;
field#0(3)[] out = ∇⊗∇... k1;
```

The program now includes a mix of tensor and field operators applied between tensors and tensor fields. The program can not compiler. In the older version of Diderot, such statements were not supported. Tensor operators trace and division can not be applied to field types. Our work supports this program and others like this.

In the older version of Diderot, to derive this computation the work is imposed on the user. The user would need to rewrite the program so that the derivative operator is only applied to field types and tensor operators are only applied to tensor 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.

Our work enables a new level of flexibility and expressiveness. 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.

1. Our representation was inspired by *Einstein Index Notation*, which is a concise written notation for tensor calculus invented by Albert Einstein [32].

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 of the EIN IR design 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

$$\mathrm{t} = \lambda\, params \langle e \rangle_{\sigma}(\mathrm{args})$$

where

- t is a tensor or field variable being assigned.
- $\lambda\, params \langle e \rangle_{\sigma}$ is an operator defined in the EIN IR, with formal parameters *params*, EIN expression e, and index space σ.
- $\sigma \in (\textsc{IndexVar} \xrightarrow{\text{fin}} (\mathbb{Z} \times \mathbb{Z}))^*$. We key the map with an index i and get a pair (lb, ub), where lb is the lower bound on i, ub is the upper bound, and each variable in σ is unique. Sometimes, we show this relationship as a triple (lb, i, ub).
- args are the argument variables to the EIN operator

For example, the outer product between two n-length vectors (u⊗v) is represented in High-IR as the assignment

$$\underset{init}{\Longrightarrow} \mathrm{t}_1 = \lambda\,(U, V)\big\langle U_i V_j \big\rangle_{\sigma}(\mathrm{u}, \mathrm{v}) \quad \text{where } \sigma = \langle 1 \le i \le n, 1 \le j \le n \rangle$$

We adopt some notational conventions to keep this punctuation concise. We might combine bounds when they are the same (*e.g.*, $\sigma = \langle 1 \le i, j \le n \rangle$). We will omit single or both

bounds for brevity (*e.g.*, $\sigma = \langle i : n, j : n \rangle$ or $\sigma = \langle \hat{i}, \hat{j} \rangle$) in cases where they are unimportant. That is, if i is an index, we will write $i : n$ or $\hat{i}$ 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. For example, the trace of a square matrix is defined as

$$\underset{init}{\Longrightarrow} \mathrm{t}_2 = \lambda\, T \left\langle \sum_{\hat{i}} T_{ii} \right\rangle (\mathrm{t}_1)$$

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 as at the EIN operator level).

As part of the translation process, applications of EIN operators are composed to form larger EIN expressions, which the compiler then simplifies using rewrite rules (see Section 3.2.2). For example, the expression trace(u⊗v) will be automatically reduced to

$$\underset{subst}{\Longrightarrow} \mathrm{t}_2 = \lambda\, (U, V) \left\langle \sum_{\hat{i}} U_i V_i \right\rangle (\mathrm{u}, \mathrm{v})$$

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

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.

Defining EIN operators It is useful to define the EIN body and EIN operator separately.

$$\mathrm{out} = \lambda T \langle T_{ji} \rangle_{ij} \quad \text{or} \quad \begin{aligned} \mathrm{out} &= \lambda T \langle e \rangle_{ij} \\ e &= T_{ji} \end{aligned}$$

The body of the EIN operator e is an EIN expression.

The EIN expression e has two free indices (i, j) . We use the following syntax $e_{[ij/ab]}$ to

show how the free indices are initiated ($i \longrightarrow a$ and $j \longrightarrow b$).

$$e_{[ij/ba]} = T_{ab} \qquad e_{[ij/aa]} = T_{aa} \qquad e_{[ij/ab]} = T_{ba}$$

We use the notation $E_{\mathbf{name}}$ to define an EIN operator and refer to it later by name.

$$E_{\mathbf{transposeT}} = \lambda T \langle T_{ji} \rangle_{ij}$$

The notation $E_{\mathbf{name}}(\text{arg})$ is the application of the EIN operator to the argument (arg).

$$\text{s} = E_{\mathbf{name}}(\text{arg})$$

The variable s is bound to the application of $E_{\mathbf{name}}$ to an argument. As a shorthand, we might refer to s as an EIN operator.

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 α and β 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 tensor variables (T_α) and field variables (F_α), which have multi-index subscripts that specify the individual component. Summations ($\sum_\sigma e$) have the usual semantics. The Levi-Civita tensors ($\mathcal{E}_{ij}$,$\mathcal{E}_{ijk}$) and Kronecker delta (δ_{ij}) are used to permute and cancel components based on their indices (defined in Section 1.2.1).

In the following, we illustrate the constructs of the EIN representation by example. Consider the EIN operator

$$\lambda\, \bar{x} \left\langle \sum_{\sigma'} e \right\rangle_{\sigma = \langle 1 \le i \le n \rangle} \qquad \text{where } \sigma' = \langle c \le j \le d \rangle$$

which has two indices i, j. The bound index i ranges from 1 to n and gives the expression

$$
\begin{array}{rcll}
\mathbf{E} & = & \lambda\, \bar{x} \langle e \rangle_{\sigma} & \text{EIN Operator} \\
\sigma & = & (\mathbb{Z} \times \textsc{IndexVar} \times \mathbb{Z})^{*} & \text{Index map} \\
e & ::= & e_{base} \mid e_{high} & \text{EIN expressions} \\
e_{base} & ::= & T_{\alpha}, A_{\alpha}, B_{\alpha} & \text{Tensor} \\
 & \mid & F_{\alpha}, G_{\alpha} & \text{Field} \\
 & \mid & \delta_{ij} & \text{Kronecker deltas} \\
 & \mid & \mathcal{E}_{ij}, \mathcal{E}_{ijk} & \text{Levi-Civita tensor} \\
 & \mid & \text{sine}(e), \text{arcsine}(e), .., \text{arctangent}(e) & \text{Trig functions} \\
 & \mid & \sqrt{e}, -e, \exp(e), e^{n} & \text{Unary operators} \\
 & \mid & e + e, e - e, \frac{e}{e}, ee & \text{Binary operators} \\
 & \mid & \sum_{\sigma} e & \text{Summation} \\
e_{high} & ::= & e_1 @ e_2 & \text{Probe of a field } e_1 \text{ at position } e_2 \\
 & \mid & \mathbf{lift}_{d}(e) & \text{Lift tensor } e \text{ to a field} \\
 & \mid & V_{\alpha} \circledast H^{\beta} & \text{Convolution} \\
 & \mid & \frac{\partial}{\partial x_{\alpha}} e & \text{Derivative of } e \\
i, j, k & \in & \textsc{IndexVar} & \text{Variable index} \\
n & \in & \textsc{ConstVar} = \mathbb{N} & \text{Constant index} \\
\mu & \in & \textsc{IndexVar} \cup \textsc{ConstVar} & \text{Single index} \\
\alpha, \beta, \gamma & = & \bar{\mu} & \text{Sequence of indices}
\end{array}
$$

Figure 2.1: The syntax of EIN operators' E and EIN expressions e in High-IR

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.

Base EIN terms EIN operators provide a mathematically sound and compact representation for tensor and field operations. EIN represents generic tensor variables T and field variables F. In lieu of individual indices to specify a shape, an α is used to represent generic fields and arbitrary-sized tensors (F_α and T_α). A tensor expression T_α has a list of indices (α) that refer to the shape of the tensor (T_{ij} is a matrix and T_{ijk} is a third-order tensor). Similarly, a field expression F_α has a list of indices that refer to the output shape (a scalar field is expressed as F and a vector field is expressed as F_i).

The indices to an EIN term dictate how the components of a tensor or field are sampled. The following are a few examples changing the type, ordering, and binding of two indices on

a 2x2 matrix M. Using two variable indices

$$\lambda T\langle T_{ij}\rangle_{\hat{i}\hat{j}}(\mathrm{M}) = \begin{bmatrix} \mathrm{M}_{11} & \mathrm{M}_{12} \\ \mathrm{M}_{21} & \mathrm{M}_{22} \end{bmatrix} \qquad \lambda T\langle T_{ji}\rangle_{\hat{i}\hat{j}}(\mathrm{M}) = \begin{bmatrix} \mathrm{M}_{11} & \mathrm{M}_{21} \\ \mathrm{M}_{12} & \mathrm{M}_{22} \end{bmatrix}$$

Binding the indices to a summation expression creates the trace operation

$$\lambda T\left\langle \sum_i T_{ii} \right\rangle(\mathrm{M}) = \ \mathrm{M}_{11} + \mathrm{M}_{22}$$

Changing one index to a constant index will slice the matrix and create a sliced term

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

Definition 3 (Sliced Terms)**.** *We use the phrase "sliced tensor" or "sliced field" to indicate an EIN term that has at least one constant index. A constant index refers to a specific component of a tensor or tensor field. For example, a 3x3 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.*

Standard arithmetic operations $(*, +, -, /)$ are represented in EIN. A $\sum_{\sigma}(e)$ indicates a summation operation of expression e. Additionally, we added some scalar operators such as trigonometric functions (we omit the full set of arithmetic operators for compactness).

Field Operations There are several forms of EIN expressions that are special to fields. The representation depends on the phase of the compiler (Figure 1.2). They translate from an abstract syntax in High-IR to computations directly on images and kernels in Mid-IR. We describe these field terms in more detail in the following.

High-IR Field Terms The probe operator $e_1@e_2$ applies field e_1 to the point e_2. The lift operator $\mathbf{lift}_d(e)$ is wrapped around a tensor expression e inside a field expression. The convolution expression $V_{\alpha_1} \circledast H^{\alpha_2}$ is the convolution operation of an image field V, with the range α_1 and a piecewise polynomial kernel H with a level of differentiation indicated by α_2.

The expression $\frac{\partial}{\partial x_\alpha} \diamond e$ denotes differentiation of the field specified by the expression e. The representation can indicate different types of differentiation. The gradient operator (∇) uses a variable index bound outside the EIN operator:

$$E_{\mathbf{gradient}} = \lambda F \left\langle \frac{\partial}{\partial x_i} \diamond F \right\rangle_{\hat{i}}$$

The variable index is bound to a summation operator to support the divergence ($\nabla\bullet$) of a vector field:

$$E_{\mathbf{divergence}} = \lambda F \left\langle \sum_{\hat{i}} \frac{\partial}{\partial x_i} \diamond F_i \right\rangle$$

We use two variable indices to represent the matrix output of the Jacobian ($\nabla\otimes$):

$$E_{\mathbf{jacobian}} = \lambda F \left\langle \frac{\partial}{\partial x_j} \diamond F_i \right\rangle_{\hat{i}\hat{j}}$$

We use a permutation tensor to support the curl ($\nabla\times$):

$$E_{\mathbf{curl3}} = \lambda F \left\langle \sum_{jk} \mathcal{E}_{ijk} \frac{\partial}{\partial x_j} \diamond F_k \right\rangle_{\hat{i}}$$

Other types of differentiation can be created by applying these differentiation operators to each other. The concise representation can be used to support a wide variety of differentiation operations.

Mid-IR Field Terms During the Mid-IR stage of the compiler probed fields are replaced by terms that directly represent computations on images and kernels (Figure 2.2). The expression $v_\alpha[\bar{e}]$ is an image field indexed at a list of integer positions $\bar{e}$. The $\mathbf{val}(i)$ notation

$$
\begin{array}{rcll}
e & ::= & e_{base} & \text{Base EIN expressions} \\
 & | & v_\alpha(\bar{e}) & \text{Image indexed at position} \\
 & | & \mathbf{val}(i) & \text{Variable represented as integer} \\
 & | & h^{\bar{\psi}}(e) & \text{Kernel with differentiation}
\end{array}
$$

Figure 2.2: The syntax of EIN expressions e in Mid-IR. The syntax builds on definitions in Figure 2.1.

lifts an index variable to an integer expression; *e.g.*, $\sum_{i=0}^{n} \mathbf{val}(i) = 0 + 1 + 2 + \cdots + n$. We use the notation h^n to refer to the nth derivative of univariate function h. In this 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 interpreted like Kronecker-deltas pairs (*i.e.*, $\psi = (c, i)$ is interpreted as $\delta_{c,i}$) and added together.

Rewrites We use arrows ($\rightarrow$ and $\Rightarrow$) to illustrate transitions into and in the EIN syntax. We use the single arrow ($\rightarrow$) to show the rewrite of an EIN expression. We use the double arrow ($\Rightarrow$) to define EIN operators and show the translation of EIN operators to other EIN operators. This includes (1) the definition of a set of EIN operators, (2) decomposition of a single EIN operator to multiple, or (3) the composition of multiple EIN operators to a single one. When relevant we label the arrows to indicate the result of a method or different part of the compiler. In this chapter we use the notation $\underset{init}{\Longrightarrow}$ to translate surface levels operators into the EIN syntax.

2.2 Generating EIN Operators

In this section, we describe the implementation process to define EIN operators (Section 2.2.1) and some of the benefits (Section 2.2.2).

2.2.1 Implementation

The Diderot compiler generates High-IR, including the EIN operators, from an explicitly typed simplified AST representation. We provide both a set of fixed and shape polymorphic EIN operators. We discuss how they are initialized and provide examples. In both cases, the body of the EIN operators are made up of EIN expressions.

Instantiate generic operations 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 α and uses de Bruijn numbering:

$$\Lambda\alpha\lambda\,(A, B)\langle A_\alpha + B_\alpha\rangle_\alpha$$

We specialize the operator to a particular shape by replacing α with a multi-index that ranges over the shape.

Built-in operators We have a fixed set of EIN operators that are used for specific types. For example, the following EIN operator:

$$\lambda T\langle T_{00}T_{11} - T_{01}T_{10}\rangle$$

is only used to compute the inverse of a 2-by-2 matrix. This operator relies on the use of constant indices. As you may recall, constant indices are used to illustrate specific components of tensors and tensor fields.

Building Blocks The body of an EIN operator uses EIN expressions to represent a computation. EIN expressions can be used together as building blocks to represent a range of

computations. For instance, consider the following three ways to multiply three tensors.

$$\begin{aligned} E_{\mathbf{outerT}} = &\quad \lambda\,(A,B)\big\langle A_i B_j \big\rangle_{\hat{i}\hat{j}} \\ E_{\mathbf{modT}} = &\quad \lambda\,(A,B)\langle A_i B_i \rangle_{\hat{i}} \\ E_{\mathbf{innerT}} = &\quad \lambda\,(A,B)\left\langle \sum_{\hat{i}} A_i B_i \right\rangle \end{aligned}$$

The EIN operators superficially look alike, but they are fundamentally different. From up to down they produce a matrix, vector, and scalar. EIN expressions could be used as building blocks to describe more intricate operators such as the 2-d matrix inverse.

$$\lambda\, F\left\langle \frac{(\sum_{\hat{k}} F_{kk}\delta_{ij}) - F_{ij}}{F_{00}F_{11} - F_{01}F_{10}} \right\rangle_{\hat{i}\hat{j}}$$

2.2.2 Advantages

The implementation of EIN operators has certain advantages. It is easier to add generality to existing operators, new operators, and *lifted* operators. The richer language enables a higher-order programming model. We discuss the benefits in more detail in the following section and provides examples.

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:

$$\lambda\,(A,B)\left\langle \sum_{\hat{k}} A_{\alpha k} B_{k\beta} \right\rangle_{\hat{\alpha}\hat{\beta}} \tag{2.1}$$

where α and β are specialized to handle different shapes. The inner product between a vector A and a matrix B is realized by instantiating α to the empty multi-index and β to a single index.

$$\underset{init}{\Longrightarrow} \lambda\,(A,B)\left\langle \sum_{\hat{k}} A_k B_{ki} \right\rangle_{\hat{i}}$$

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. To recall, *lifted* operators ($P^{\uparrow}$) are tensor operators that can be applied to field types (Definition 1 on page 5). The EIN IR makes implementing *lifted* operators easy. Consider the P operator created for the inner product (Equation 2.1), instead of using tensor variables (A, B) we can use field variables (F, G) to create a $P^{\uparrow}$ operator.

$$\lambda\,(F, G)\left\langle \sum_{\hat{k}} F_{\alpha k} G_{k\beta} \right\rangle_{\hat{\alpha}\hat{\beta}}$$

The type checker then instantiates the generic definition by the field types.

$$\underset{init}{\Longrightarrow} \lambda\,(F, G)\left\langle \sum_{\hat{k}} F_{k} G_{ki} \right\rangle_{\hat{i}}$$

Just as easily, an EIN operator can be applied between a tensor variable and field variable.

$$\lambda\,(T, F)\left\langle \sum_{\hat{k}} T_{\alpha k} F_{k\beta} \right\rangle_{\hat{\alpha}\hat{\beta}} \qquad \lambda\,(F, T)\left\langle \sum_{\hat{k}} F_{\alpha k} T_{k\beta} \right\rangle_{\hat{\alpha}\hat{\beta}}$$

The ease of implementing *lifted* operators makes it easy to add a certain generality and flexibility to the language.

Adding to the surface language Adding new (and *lifted*) tensor operators to the Diderot language is simple. Once a computation is in EIN notation the compiler can translate it to executable code. So it is easy to add new operators (using the EIN syntax) without having to add a lot of extra code in the compiler. The new operators include shape-specific operators (2-d inverse and scalar trig operators), support for arbitrary sized tensors and tensor fields

Table 2.1: EIN operators

$$E_{\textbf{trigF}} = \lambda F \langle \kappa(F) \rangle \qquad E_{\textbf{curl3}} = \lambda F \left\langle \sum_{\hat{j}\hat{k}} \mathcal{E}_{ijk} \frac{\partial}{\partial x_j} \diamond F_k \right\rangle_{\hat{i}}$$

$$E_{\textbf{normalizeF}} = \lambda F \left\langle \frac{F_{\hat{\alpha}}}{\sqrt{\sum_{\beta} F_\beta F_\beta}} \right\rangle_{\hat{\alpha}} \qquad E_{\textbf{divergence}} = \lambda F \left\langle \sum_{\hat{j}} \frac{\partial}{\partial x_j} \diamond F_{\alpha j} \right\rangle_{\hat{\alpha}}$$

$$E_{\textbf{crossTF}} = \lambda T, G \left\langle \sum_{\hat{j}\hat{k}} \mathcal{E}_{ijk} T_j G_k \right\rangle_{\hat{i}} \qquad E_{\textbf{det3x3F}} = \lambda F \left\langle \sum_{\hat{i}\hat{j}\hat{k}} (F_{0i} F_{1j} F_{2k} \mathcal{E}_{ijk}) \right\rangle$$

$$E_{\textbf{gradient}} = \lambda F \left\langle \frac{\partial}{\partial x_i} \diamond F \right\rangle_{\hat{i}} \qquad E_{\textbf{normF}} = \lambda F \left\langle \sqrt{\sum_{\hat{\alpha}} F_\alpha F_\alpha} \right\rangle$$

(normalize(F) and $\nabla\bullet$), and index-dependent operations (T $\times$ G and det(F)). Table 2.1 refers to some examples of the following EIN operators.

2.3 Field Reconstruction

During the transition from High-IR to Mid-IR, higher-order constructs get replaced by lower-order constructs. Probed fields $v \circledast h(x)$ are replaced with terms that directly express computations on images and kernels. We continue using the notation for fields, field transformation matrix M, integer position vectors n, and fractional position vector f introduced in Section 1.2.

Design Traditional index notation [32] is not sufficient to define the reconstruction of fields. We created the EIN term $v \circledast h$ to represent the convolution operator in High-IR. We introduce the EIN expressions $v_\alpha(\bar{e})$, $\textbf{val}(i)$, and $h^{\bar{\psi}}(e)$ to represent reconstructed fields in Mid-IR. The design details were described in Section 2.1.

Implementation We build on the exposition from our previous work [18], reproduced here for convenience, to explain how field reconstruction is represented in the EIN syntax. Let F be a 2-d vector field defined as follows:

field#0(2)[2] F = tent $\circledast$ img("i.nrrd");

vec2 out = F(p);

The result of probing vector field is evaluated as

$$\begin{bmatrix} \sum_{i,j:1-s}^{s} v_0[\mathrm{n}_0 + i, \mathrm{n}_1 + j] h(\mathrm{f}_0 - i) h(\mathrm{f}_1 - j) \\ \sum_{i,j:1-s}^{s} v_1[\mathrm{n}_0 + i, \mathrm{n}_1 + j] h(\mathrm{f}_0 - i) h(\mathrm{f}_1 - j) \end{bmatrix}$$

In High-IR the operation is represented as a single EIN operator

$$\underset{init}{\Longrightarrow} \lambda\,(V, H, T) \langle V_i \circledast H(T) \rangle_{\langle i:2 \rangle} (\mathrm{F}, \mathrm{tent}, \mathrm{P}) \tag{2.2}$$

The field F is reconstructed in EIN notation as

$$\begin{array}{l} \lambda\,(v, h, n, f) \langle e \rangle_{\langle i:2 \rangle} (\mathrm{v},\ \mathrm{tent},\ \mathrm{n},\ \mathrm{f}) \\ \underset{recon}{\Longrightarrow} \quad e = \sum_{jk=1-s}^{s} v_i[n_0 + \mathbf{val}(j), n_1 + \mathbf{val}(k)] h(f_0 - \mathbf{val}(j)) h(f_1 - \mathbf{val}(k)) \end{array} \tag{2.3}$$

We use notation $\underset{recon}{\Longrightarrow}$ to represent the application of field reconstruction. The specific axis for the fractional **f** and integer **n** position are represented with a constant index, such as f_0.

The EIN notation tracks the differentiation component applied to the convolution kernel by keeping a list of indices.

$$\frac{\partial}{\partial x_i} \diamond H(x, y) = h^{\delta_{0i}}(x) h^{\delta_{1i}}(y)$$

A second derivative adds another variable index:

$$\frac{\partial}{\partial x_{ij}} \diamond H(x, y) = \sum_{\hat{i}} \sum_{i} h^{\delta_{0i} + \delta_{0j}}(x) h^{\delta_{1i} + \delta_{1j}}(y)$$

Generally, applying differentiation operators creates an expression of the form

$$\frac{\partial}{\partial x_{\alpha_0, \alpha_1, .., \alpha_n}} \diamond H(x, y) = h^{\delta_{0\alpha_0} + \delta_{0\alpha_1} .. + \delta_{0\alpha_n}}(x) h^{\delta_{1\alpha_0} + \delta_{1\alpha_1} .. + + \delta_{1\alpha_n}}(y) \tag{2.4}$$

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}]. More generally the field term: $\frac{\partial}{\partial x_{\alpha_0,\alpha_1,..,\alpha_n}} \diamond F$ is multiplied by P for each index of convolution measured in the derivatives: $\sum_{\beta} P_{\alpha_0\beta_0} P_{\alpha_1\beta_1} \ldots P_{\alpha_n\beta_n} (\frac{\partial}{\partial x_\beta} \diamond F)$.

Example The Hessian of a scalar field out= $\nabla \otimes \nabla \varphi(x)$; is represented with EIN operator

$$\begin{aligned} & \text{out} &&= \mathrm{P} \bullet \mathrm{T}_0 \bullet \mathrm{P} \\ \xRightarrow[recon]{} \quad & \mathrm{T}_0 &&= \lambda(v,h,n,f)\langle e \rangle_{\hat{i}\hat{j}}(\mathrm{img}, \mathrm{bspln3}, \mathrm{n}, \mathrm{F}) \\ & e &&= \sum_{k,l=1-s}^{s} v[\mathrm{n}_0 + \mathbf{val}(k), \ldots] h^{\delta_{0i}+\delta_{0j}}(\mathrm{f}_0 - \mathbf{val}(k)) h^{\delta_{1i}+\delta_{1j}}(\mathrm{f}_1 - \mathbf{val}(l)) \end{aligned} \tag{2.5}$$

In out (Equation 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. The original compiler used a direct-style notation inside of the compiler. In Section 2.4.1 we describe the direct-style notation and compare it to the EIN syntax. We imagine creating an EIN expression that could more directly represent big computations, called"direct-EIN". We describe and discuss that approach in Section 2.4.2. Lastly, we discuss and summarize the results of creating the new IR for the Diderot user.

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 ∇, were primitive operators) in its intermediate representation (IR) [18]. 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 ideas 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 compares the direct-style compiler and the EIN syntax.

Direct-Style: The first design of Diderot used direct-style notation using italics. 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 vectors of length 2 is represented in direct-style as $\mathit{InnerP\_VecVec}_2$ (u,v).

$$u \bullet v = (u[0] * v[0]) + (u[1] * v[1]) \tag{2.6}$$

Direct-style notation gives a compact representation, but requires more tensor-shape specific operators (*e.g.*, *InnerP_Vec2Mat22*, *InnerP_Vec2Mat23*(u,m), and *InnerP_Mat2Mat2*(m,m)). EIN aims to be almost as compact as the direct-style notation while revealing internal details that enable the translation and optimization of 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 as lifted operators (Definition 1 on page 5). 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 the 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.*, $\nabla(e_1 + e_2) \longrightarrow_{direct-style} \nabla e_1 + \nabla e_2$). This approach suffered from a combinatorial blowup in the number of rules, which made it difficult to add new lifted operators. Furthermore, the direct-style IR 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 fields ($\nabla, \nabla\otimes$), 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 1.7);
i.e., $\nabla \bullet F \longrightarrow_{direct-style} \nabla \bullet (V \circledast H) \neq V \circledast H^{(1)}$. 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 completely avoids dependence on any particular choice of basis for representing tensor components (like scaling vector field F with 4*F), while index-dependent means the evaluation 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 depend on the indices of components and thus are not index free.

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 ($\nabla\otimes(\nabla\times$F)) is needed for shading renderings of curl-related quantities. Mathematically, the computation is represented as:

$$\begin{bmatrix} \frac{\partial^2}{\partial x \partial y}F_2 - \frac{\partial^2}{\partial x \partial z}F_1, & \frac{\partial^2}{\partial y \partial y}F_2 - \frac{\partial^2}{\partial y \partial z}F_1, & \frac{\partial^2}{\partial z \partial y}F_2 - \frac{\partial^2}{\partial z \partial z}F_1 \\ \frac{\partial^2}{\partial x \partial z}F_0 - \frac{\partial^2}{\partial x \partial x}F_2, & \frac{\partial^2}{\partial y \partial z}F_0 - \frac{\partial^2}{\partial y \partial x}F_2, & \frac{\partial^2}{\partial z \partial z}F_0 - \frac{\partial^2}{\partial z \partial x}F_2 \\ \frac{\partial^2}{\partial x \partial x}F_1 - \frac{\partial^2}{\partial x \partial y}F_0, & \frac{\partial^2}{\partial y \partial x}F_1 - \frac{\partial^2}{\partial y \partial y}F_0, & \frac{\partial^2}{\partial z \partial x}F_1 - \frac{\partial^2}{\partial z \partial y}F_0 \end{bmatrix} \tag{2.7}$$

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 cannot handle the example. EIN can handle the example by using the following compact representation:

$$\mathrm{t} = \lambda\, F\Big\langle \Sigma_{\{1\leq k,l\leq 3\}} \mathcal{E}_{ikl} \frac{\partial}{\partial x_{jk}} F_l \Big\rangle_{\sigma}(\mathrm{F}) \quad \sigma = \langle 1 \leq i,j \leq 3 \rangle \tag{2.8}$$

2.4.2 EIN as building blocks

In the following section, we analyze how EIN operators are created for more complicated operators. Each time a new operator is implemented in EIN we first consider using the existing EIN syntax to create it rather than creating a new EIN expression. By using the existing EIN IR, there is no other code that needs to be added to the compiler. 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 defining more complicated operations then we might consider creating a new EIN expression ("Direct-Ein") instead of creating the computation with existing ones ("Building blocks"). "Direct-Ein" notation would be used to directly represent the computation, (in-

spired by the Direct-Style syntax of the original compiler) but created as an EIN expression so it can be used as a building block in another operator. Consider possible representations of the 3-d determinant and 2-d inverse.

	Building blocks	Direct-Ein
$E_{\text{det-d3}}$	$\left\langle \Sigma_{\hat{i}}(F_{0i}\Sigma_{\hat{j}}(F_{1j}\Sigma_{\hat{k}}(F_{2k}\mathcal{E}_{ijk}))) \right\rangle_{\hat{i}\hat{j}\hat{k}}$	$\langle \text{DET-3d(F)} \rangle_{\hat{i}\hat{j}}$
$E_{\text{inv-d2}}$	$\left\langle \frac{(\Sigma_{\hat{k}}F_{kk}\delta_{ij}) - F_{ij}}{F_{ii}F_{jj} - F_{ij}F_{ji}} \right\rangle_{\hat{i}\hat{j}}$	$\langle \text{INV-2d(F,ij)} \rangle_{\hat{i}\hat{j}}$

The Direct-Ein structure offers a concise representation in place of a more complicated EIN operation and it could enable new rewrites such as INV-2d (INV-2d(e)) $\longrightarrow_{direct-style}$ 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 Direct-Ein structures could make the rewriting system more complicated.

While the Direct-Ein structure can offer a printer-friendly representation, the concise representation may not be sustainable. In fact, the Direct-Ein structure may inevitably need to be broken into smaller pieces for three reasons. First, realizable field terms (Definition 2 on page 12) need to be identified before field reconstruction (Section 2.3). Second, new rewrite rules might lead to a term that is difficult to express in the Direct-Ein structure. Lastly, decomposing Direct-Ein structures into smaller pieces may be the best way to apply compiler optimizations (such as finding common subexpressions). Considering, the limitations and the implementation costs to adding a new EIN expression 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* [12] 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.

```
field#2(3)[]C = -∇(|∇F|) • ∇F/|∇F|;
field#1(3)[σ]D = ∇ C ...;
```

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 that 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 described 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.

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 normalized before field reconstruction (Section 1.2.3). 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 drives the normalization of EIN expressions (Section 3.2).

The initial implementation of EIN had a significant space issue. The source of the problem occurred during the rewriting and composition on EIN operators. During normalization the IR expands quickly, makes large complicated expressions and breaks sharing. 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. To address the compilation issue, we have developed a number of techniques to keep the size of the program under control (Section 3.3).

To evaluate the impact and cost of the compilation techniques we measure the compilation and execution time for various benchmarks (Section 3.4). For the benchmarks we use a mix of Diderot programs with varying degrees of tensor math. We offer three experiments. The first evaluates the impact of implementing the EIN IR by comparing the results to the original compiler. The second measures the impact of different compiler setting (by turning compilation techniques on and off). The third experiment compares first and higher-order versions of programs. Overall, the implementation of EIN can allow programs with higher-level of math than was previously possible.

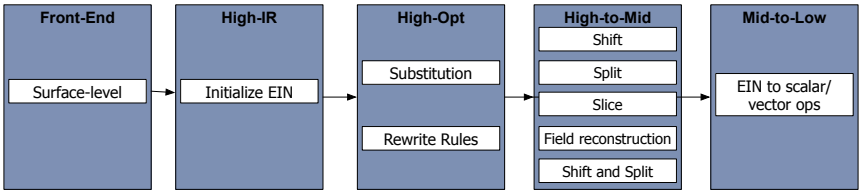

Figure 3.1: Transformations that are applied in the EIN syntax in the Diderot pipeline

We describe the normalization process in Section 3.2 and introduce the compilation techniques we developed in Section 3.3. Lastly we evaluate the techniques in Section 3.4. We use the notation introduced in Section 2.1. To recall, at times we might define an EIN expression separately from the EIN operator. As a shorthand, we refer to an EIN operator bound to a variable s as EIN operator s.

3.1 Overview of implementing the EIN syntax

We provide a diagram of different stages of the implementation process in Figure 3.1. The larger box represents a step in the Diderot compiler and corresponds to compiler phases as introduced in Figure 1.2. The white boxes in the diagram represent different implementation methods and steps that are introduced in this dissertation.

The Diderot language supports a surface level syntax of different operators (Section 1.1.1). The computations are mapped to EIN operators that are initiated by the type-checker (Section 2.2.1). High-IR represents each tensor or field computation as an EIN operator. In High-Opt the EIN operators are normalized (Section 3.2) by applying the substitution method (Section 3.2.1) and rewriting rules (Section 3.2.2).

In High-to-Mid we apply additional optimizations. We apply methods by the name of *Shift* (Section 3.3.1), *Split* (Section 3.3.3), and *Slice* (Section 3.3.4). Afterwards, field terms are reconstructed (Section 2.3). *Shift* and *Split* are applied again after field reconstruction (described in an example in Section 3.3.5). Lastly, the EIN operators are converted to scalar and vector operators in the Mid-to-Low phase. The translation is omitted from this

dissertation but was described in a previous paper [15].

As defined in Section 2.1 we make a distinction between defining a set of EIN operators ($\Rightarrow$) and rewriting an EIN expression ($\rightarrow$). When applicable we label the arrow with a method name or other phrase to indicate the part of the implementation process (as shown in Figure 3.1). We use notation $\underset{init}{\Longrightarrow}$ to translate surface levels operators into the EIN syntax. We use the notation $\underset{subst}{\Longrightarrow}$,$\underset{split}{\Longrightarrow}$, and $\underset{slice}{\Longrightarrow}$ to indicate the application of the method substitution, split, and slice, respectively. We use notation $\underset{recon}{\Longrightarrow}$ to indicate field reconstruction. We use the notation $\underset{shift}{\longrightarrow}$ and $\underset{rule}{\longrightarrow}$ to indicate a rewrite after applying shift and a rewrite rule, respectively. We use the arrow $\underset{rule}{\longrightarrow}^*$ to indicate a rewritten term after applying multiple rewrite rules.

3.2 Substitution and Rewriting

This section describes how EIN operators are normalized. First we use a systematic approach to compose EIN operators with 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 3.2.1 we describe substitution process. In Section 3.2.2 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 3.3.

3.2.1 Substitution

The method of substitution involves composing EIN operators into one operator that represents the computation. An EIN operator is applied to some arguments and we expand uses of the arguments with its binding. The method effectively inlines definitions with this

method. In this section, we describe the implementation of the substitution process followed by an example. We use the notation $\underset{subst}{\Longrightarrow}$ to indicate the application of the substitution method on EIN operators.

Implementation Consider the following scenario. The EIN operator t is an argument to EIN operator r where the binding for variable t is $\lambda(\bar{s})\langle e'' \rangle_{\alpha}(\bar{s})$. Substitution creates the following top-level rewrite:

$$\begin{array}{l} \mathrm{r} = \lambda(T)\langle e \rangle_{\sigma}(\mathrm{t}) \\ \mathrm{t} = \lambda(\bar{s})\langle e'' \rangle_{\alpha}(\bar{s}) \end{array} \underset{subst}{\Longrightarrow} \mathrm{r} = \lambda(\bar{s})\langle e' \rangle_{\sigma}(\bar{s})$$

As a result a single EIN operator that represents the computation. The implementation requires some additional bookkeeping that we omit in the description.

Replace Parameter The parameters of an EIN operator indicate how an argument is represented inside the EIN expression. At the top level there is a mapping from parameter to EIN expression.

$$\frac{\Gamma(T) = \langle e'' \rangle_{\hat{\alpha}}}{\Gamma \vdash T_{\beta} \rightharpoonup e''_{[\alpha/\beta]}} \qquad \frac{\Gamma(F) = \langle e'' \rangle_{\hat{\alpha}}}{\Gamma \vdash F_{\beta} \rightharpoonup e''_{[\alpha/\beta]}}$$

$$\text{where } \Gamma : \text{ Parameter } \longrightarrow \text{ EinExp}$$

The method scans the body of EIN expression (e) and looks for terms with a matching parameter. It replaces the matching term with the body of the substitution (e'') after some rewriting.

Rewriting substitution term The next step takes the substitution term and rewrites it. The indices in the substitution term are instantiated by the (indices of the) term it is replacing (β). In the following we use the arrow $\rightharpoonup$ to indicate a substitution rewrite on EIN expressions.

In a base term (tensor and field) we remap the indices.

$$\frac{\text{if } \alpha_k = i \text{ then } x = \beta_k \text{ else } x = i}{T_{i[\alpha/\beta]} \rightharpoonup T_x} \tag{3.1}$$

If variable index i is in α then we look at β otherwise the expression stays the same. Similarly, the same is done for field terms.

Substitution is distributed over operators and examines the embedded subexpression(s). The following is an example of the differentiation operator.

$$\frac{\text{if } \alpha_k = i \text{ then } j = \beta_k \text{ else } j = i \quad \Gamma \vdash e_1 \rightharpoonup e_1'}{(\frac{\partial}{\partial x_i} \diamond e_1)_{[\alpha/\beta]} \rightharpoonup (\frac{\partial}{\partial x_j} \diamond e_1')} \tag{3.2}$$

The substitution method examines subexpression e_1 and it rewrites the indices on the partial derivative term. Figure 3.2 summarizes the rest of the substation rewrites. We assume the variable convention [6].

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

```
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.

$$\begin{array}{rllll} & \mathrm{t}_1 = \lambda(A,B) & \langle A_\gamma + B_\gamma \rangle_{\hat{\gamma}} & (\mathrm{a},\mathrm{b}) & \text{where } \forall \gamma_i \in \gamma.\ \ 1 \le \gamma_i \le \varsigma_i \\ \overset{\Longrightarrow}{\underset{init}{}} & \mathrm{t}_2 = \lambda(C,T) & \langle C_\alpha T_\beta \rangle_{\hat{\alpha}\hat{\beta}} & (\mathrm{c},\mathrm{t}_1) & \text{where } \forall \alpha_i \in \alpha.\ \ 1 \le \alpha_i \le \sigma_i \\ & & & & \text{and } \forall \beta_i \in \beta.\ \ 1 \le \beta_i \le \varsigma_i \end{array}$$

h

$$1)\frac{\text{if } \alpha_k = i \text{ then } x = \beta_k \text{ else } x = i}{T_{i_{[\alpha/\beta]}} \rightharpoonup T_x}$$

$$2)\frac{\text{if } \alpha_k = i \text{ then } x = \beta_k \text{ else } x = i \quad \text{and} \quad \text{if } \alpha_k = j \text{ then } y = \beta_k \text{ else } y = j}{V_i \circledast h^j{}_{[\alpha/\beta]} \rightharpoonup V_x \circledast h^y}$$

$$3)\ \frac{\text{if } \alpha_k = i \text{ then } x = \beta_k \text{ else } x = i \quad \text{and} \quad \text{if } \alpha_k = j \text{ then } y = \beta_k \text{ else } y = j}{\delta_{ij_{[\alpha/\beta]}} \rightharpoonup \delta_{xy}}$$

Note the same is done for $\mathcal{E}_{ij}$ and $\mathcal{E}_{ijk}$

$$4)\ \frac{\sum_i (e)^{[\alpha/\beta]}}{\sum_i (e_{[\alpha/\beta]})} \text{ note } i \notin \beta$$

$$5)\ \frac{\text{if } \alpha_k = i \text{ then } j = \beta_k \text{ else } j = i \quad \Gamma \vdash e_1 \rightharpoonup e_1'}{\frac{\partial}{\partial x_i} \diamond e_1{}_{[\alpha/\beta]} \rightharpoonup (\frac{\partial}{\partial x_j} \diamond e_1')}$$

$$6)\ \frac{\Gamma \vdash e_1 \rightharpoonup e_1'}{\Gamma \vdash \odot_1 e_1 \rightharpoonup\ \odot_1 e_1'} \quad \frac{\Gamma \vdash e_1 \rightharpoonup e_1'\ \ \Gamma \vdash e_2 \rightharpoonup e_2'}{\Gamma \vdash e_1 \odot_2 e_2 \rightharpoonup e_1' \odot_2 e_2'}$$

where $\odot_1 = \{\mathbf{lift}_d(\cdot)\sqrt{\cdot}, sine(\cdot), \dots\} \quad \odot_2 = \{+, -, *, \dot{\div}, @\}$

Figure 3.2: Substitution rules

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.

$$\underset{subst}{\Longrightarrow} \mathrm{t}_2 = \lambda(C, A, B)\big\langle C_\alpha(A_\beta + B_\beta)\big\rangle_{\sigma_\varsigma}(\mathrm{c}, \mathrm{a}, \mathrm{b}) \text{ where } \gamma \text{ is initialized by } \beta$$

The result is a single EIN operator to represent the composition of the two EIN operators.

3.2.2 Rewriting

Our transformation rules do domain-specific rewriting and apply optimizations. Domain-specific and tensor calculus based rewriting are applied to normalize field terms (Definition 2 on page 12). Index-based and algebraic rewrites are used to simplify EIN terms.

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 Chapter 2 to represent EIN expressions. We use the notation $\underset{rule}{\longrightarrow}$ to indicate a rewrite rule.

Domain-specific rewrites involve pushing a probe operation towards the field terms. The probe operation is pushed past unary operators by the following rewrite rules:

$$(-F_\alpha)@x \underset{rule}{\longrightarrow} -(F_\alpha@x) \qquad \sqrt{F_\alpha}@x \underset{rule}{\longrightarrow} \sqrt{F_\alpha@x}$$

$$\mathbf{sine}(F_\alpha)@x \underset{rule}{\longrightarrow} \mathbf{sine}(F_\alpha@x) \qquad \mathbf{arcsine}(F_\alpha)@x \underset{rule}{\longrightarrow} \mathbf{arcsine}(F_\alpha@x)$$

The probe operator is distributed over binary operators by the following rewrite rules:

$$(e_1 + e_2)@x \underset{rule}{\longrightarrow} (e_1@x) + (e_2@x) \qquad (e_1 * e_2)@x \underset{rule}{\longrightarrow} e_1@x * e_2@x$$

$$(e_1 - e_2)@x \underset{rule}{\longrightarrow} (e_1@x) - (e_2@x) \qquad (\tfrac{e_1}{e_2})@x \underset{rule}{\longrightarrow} \tfrac{e_1@x}{e_2@x}$$

The probe of a non-field term is reduced.

$$\begin{aligned} \mathbf{lift}_d(e)@x &\xrightarrow[rule]{} e & \qquad \delta_{ij}@x &\xrightarrow[rule]{} \delta_{ij} \\ \mathcal{E}_{ij}@x &\xrightarrow[rule]{} \mathcal{E}_{ij} & \qquad \mathcal{E}_{ijk}@x &\xrightarrow[rule]{} \mathcal{E}_{ijk} \end{aligned}$$

We push the field expression past tensor operators, which changes a higher-order term to a first-order term. For example, consider scaling field F_β by a scalar s:

$$(sF_\beta)(x) \xrightarrow[rule]{} s(F_\beta(x))$$

The rewriting changes the probe of the scaling of a field F_β by a scalar s to the scaling of a tensor resulting from probing $F_\beta(x)$.

Differentiation The rewrites push the differentiation operator to the field leaves in an EIN expressions. We demonstrate rewriting rules based on tensor calculus that are used to apply the differentiation operator on EIN expressions. In the following we label each identity on the left and rewrite the EIN syntax on the right.

Product rule:	$\frac{\partial}{\partial x_i} \diamond (e_1 e)$	$\xrightarrow[rule]{} e_1(\frac{\partial}{\partial x_i} \diamond e) + e(\frac{\partial}{\partial x_i} \diamond e_1)$
Quotient rule:	$\frac{\partial}{\partial x_i} \diamond \frac{e_1}{e_2}$	$\xrightarrow[rule]{} \frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2}$
Power Rule:	$\frac{\partial}{\partial x_i} \diamond (e^n)$	$\xrightarrow[rule]{} \mathbf{lift}_d(n) e^{n-1}(\frac{\partial}{\partial x_i} \diamond e)$
Chain Rule:	$\frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e))$	$\xrightarrow[rule]{} \mathbf{exp}(e)(\frac{\partial}{\partial x_i} \diamond e)$
Power and Chain Rule:	$\frac{\partial}{\partial x_i} \diamond (\sqrt{e})$	$\xrightarrow[rule]{} \frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{lift}_d(2)\sqrt{e}}$

The rewrites include the differentiation of constants

$$\begin{aligned} \frac{\partial}{\partial x_i} \diamond (\mathbf{lift}_d(0)) &\xrightarrow[rule]{} \mathbf{lift}_d(0) \\ \frac{\partial}{\partial x_i} \diamond (\delta_{ij}) &\xrightarrow[rule]{} \mathbf{lift}_d(0) \end{aligned}$$

and trigonometric identities.

$$\begin{aligned}
\frac{\partial}{\partial x_i} \diamond \mathbf{cosine}(e) &\xrightarrow[rule]{} -\mathbf{sine}(e)(\frac{\partial}{\partial x_i} \diamond e) \\
\frac{\partial}{\partial x_i} \diamond \mathbf{arccosine}(e) &\xrightarrow[rule]{} \frac{-\mathbf{lift}_d(1)(\frac{\partial}{\partial x_i} \diamond e)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}} \\
\frac{\partial}{\partial x_i} \diamond \mathbf{sine}(e) &\xrightarrow[rule]{} \mathbf{cosine}(e)(\frac{\partial}{\partial x_i} \diamond e) \\
\frac{\partial}{\partial x_i} \diamond \mathbf{arcsine}(e) &\xrightarrow[rule]{} \frac{\mathbf{lift}_d(1)(\frac{\partial}{\partial x_i} \diamond e)}{\sqrt{\mathbf{lift}_d(1) - (e * e)}} \\
\frac{\partial}{\partial x_i} \diamond \mathbf{tangent}(e) &\xrightarrow[rule]{} \frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{cosine}(e)\mathbf{cosine}(e)} \\
\frac{\partial}{\partial x_i} \diamond \mathbf{arctangent}(e) &\xrightarrow[rule]{} \frac{\mathbf{lift}_d(1)(\frac{\partial}{\partial x_i} \diamond e)}{\mathbf{lift}_d(1) + (e * e)}
\end{aligned}$$

The differentiation operator is distributed over an arithmetic operator in the following rewrite rules:

$$\begin{aligned}
\frac{\partial}{\partial x_i} \diamond (-e) &\xrightarrow[rule]{} -(\frac{\partial}{\partial x_i} \diamond e) \\
\frac{\partial}{\partial x_i} \diamond (e_1 - e_2) &\xrightarrow[rule]{} (\frac{\partial}{\partial x_i} \diamond e_1) - (\frac{\partial}{\partial x_i} \diamond e_2) \\
\frac{\partial}{\partial x_i} \diamond (e_1 + e_2) &\xrightarrow[rule]{} (\frac{\partial}{\partial x_i} \diamond e_1) + (\frac{\partial}{\partial x_i} \diamond e_2)
\end{aligned}$$

The EIN syntax can simplify the total number of rewrites that need to be created. Rewrite rules can stand for different computations. For example, the rewrite written in EIN notation

$$\frac{\partial}{\partial x_\alpha} \diamond (e_1 + e_2) \xrightarrow[rule]{} \frac{\partial}{\partial x_\alpha} \diamond e_1 + \frac{\partial}{\partial x_\alpha} \diamond e_2$$

represents multiple direct-style rewrites

$$\begin{aligned}
\nabla(\varphi_1 + \varphi_2) &\longrightarrow_{direct-style} (\nabla\varphi_1) + (\nabla\varphi_2), \\
\nabla \times (F + G) &\longrightarrow_{direct-style} (\nabla \times F) + (\nabla \times G), \text{ and} \\
\nabla \bullet (F + G) &\longrightarrow_{direct-style} (\nabla \bullet F) + (\nabla \bullet G)
\end{aligned}$$

We simplify the differentiation of a field term by moving the indices on the differentiation

operator onto the kernel inside a convolution expression.

$$\frac{\partial}{\partial x_\mu} \diamond (V_\alpha \circledast H^\beta) \xrightarrow[rule]{} V_\alpha \circledast H^{\beta\mu} \tag{3.3}$$

By using a list of variable indices to represent differentiation (rather than just an integer) we can represent different types of differentiation. The direct style version of this rule (Equation 1.7) limited the type of differentiation operators that could be supported and differentiated to only gradient ∇ and Hessian $\nabla \otimes \nabla$.

Index based optimizations Unlike in the direct-style IR, the EIN IR allows us to express index-based reductions. Applying one of these optimizations removes at least one summation from the operation. For example, the variable indices on a differentiation operator could match two indices as an epsilon term.

$$\mathcal{E}_{ijk} \frac{\partial}{\partial x_{ij}} \diamond e \xrightarrow[rule]{} \mathbf{lift}_d(0)$$

This rewrite enables the compiler to find identities:

$$\nabla \times \nabla\varphi \longrightarrow_{direct-style} 0 \qquad \text{and} \qquad \nabla \bullet \nabla \times F \longrightarrow_{direct-style} 0 \tag{3.4}$$

Two epsilons in an expression with a shared index can be rewritten to deltas [21].

$$\mathcal{E}_{ijk}\mathcal{E}_{ilm} \xrightarrow[rule]{} \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} \tag{3.5}$$

A δ_{ij} expression can be applied to tensors, fields, and the del operator.

$$\delta_{ij}T_j \xrightarrow[rule]{} T_i \qquad \delta_{ij}F_j \xrightarrow[rule]{} F_i \qquad \frac{\partial}{\partial x_j} \diamond \delta_{ij} e \xrightarrow[rule]{} \frac{\partial}{\partial x_i} \diamond e \tag{3.6}$$

Algebraic rewrites The remaining rewrites are used to make simplifications and reductions.

$$
\begin{array}{lll}
-0 \underset{rule}{\longrightarrow} 0 & e_1 - 0 \underset{rule}{\longrightarrow} e_1 & 0 * e_1 \underset{rule}{\longrightarrow} 0 \\
0 + e_2 \underset{rule}{\longrightarrow} e_2 & 0 + e_2 \underset{rule}{\longrightarrow} e_2 & \sqrt{(e)}\sqrt{(e)} \underset{rule}{\longrightarrow} e \\
\frac{\frac{e_1}{e_2}}{e_3} \underset{rule}{\longrightarrow} \frac{e_1}{e_2 e_3} & \frac{\frac{e_1}{e_2}}{e_3} \underset{rule}{\longrightarrow} \frac{e_1 e_3}{e_2} & \frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}} \underset{rule}{\longrightarrow} \frac{e_1 e_4}{e_2 e_3}
\end{array}
\tag{3.7}
$$

3.3 Optimization and Transformations

As discussed earlier in this chapter, field normalization drives the normalization of EIN expressions. We chose to simplify rewriting by first composing EIN operators and then applying rewrites to a single body. Unfortunately, during substitution and normalization we replicate code and break sharing, which causes a blowup in the size of the IR.

The code replication issue can be caused by term rewriting. Rewriting can create multiple instances of the same term. Consider the expansion of the gradient of the norm of the gradient:

```
field#k(d)[] f;
field#k-1(d)[d] g = ∇f;
field#k-1(d)[d] n = |g|;
field#k-2(d)[d,d] h = ∇n;
```

The surface-level operators are mapped (Section 2.2) to the following EIN operators (defined in Figure 2.1).

$$
\begin{array}{rl}
 & \mathrm{g} = E_{\textbf{gradient}}(\mathrm{f}) \\
\underset{init}{\Longrightarrow} & \mathrm{n} = E_{\textbf{normF}}(\mathrm{g}) \\
 & \mathrm{h} = E_{\textbf{gradient}}(\mathrm{n})
\end{array}
$$

Substitution (Section 3.2.1) composes the EIN operators into the following EIN operator.

$$
\begin{array}{rl}
 & \mathrm{h} = \lambda f \langle e \rangle_i(\mathrm{f}) \\
\underset{subst}{\Longrightarrow} & e = \frac{\partial}{\partial x_i} \diamond \sqrt{\sum_j (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}
\end{array}
$$

Normalization rewrites (Section 3.2.2) on the EIN body e. In the following we write the EIN expression on the left-hand-side and add a description on the right.

$$e \xrightarrow[rule]{}{}^{*} \frac{\frac{\partial}{\partial x_i} \diamond \sum_j (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}{\mathbf{lift}_d(2) \sqrt{\sum_j (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}} \quad \text{Apply chain rule}$$

$$\xrightarrow[rule]{}{}^{*} \frac{\sum_j \frac{\partial}{\partial x_i} \diamond (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}{\mathbf{lift}_d(2) \sqrt{\sum_j (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}} \quad \text{Push derivative past summation}$$

$$\xrightarrow[rule]{}{}^{*} \frac{\sum_j (\frac{\partial}{\partial x_j} \diamond F) \frac{\partial}{\partial x_i} \diamond (\frac{\partial}{\partial x_j} \diamond F) + (\frac{\partial}{\partial x_j} \diamond F) \frac{\partial}{\partial x_i} \diamond (\frac{\partial}{\partial x_j} \diamond F)}{\mathbf{lift}_d(2) \sqrt{\sum_j (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}} \quad \text{Apply product rule}$$

$$\xrightarrow[rule]{}{}^{*} \frac{\sum_j (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_{ij}} \diamond F) + (\frac{\partial}{\partial x_{ij}} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}{\mathbf{lift}_d(2) \sqrt{\sum_j (\frac{\partial}{\partial x_j} \diamond F)(\frac{\partial}{\partial x_j} \diamond F)}} \quad \text{Compact differentiation term}$$

To more clearly see the redundancies we replicate the expanded computation using a familiar syntax. We use notation $\xrightarrow[\text{Surface}]{EIN}$ to translate an operator in the EIN syntax into a mathematically friendly syntax.

$$\text{h} \xrightarrow[\text{Surface}]{EIN} \frac{(t_1 \bullet t_2) + (t_2 \bullet t_1)}{2\sqrt{t_1 \bullet t_1}}$$

where t_1 is the gradient of f and t_2 is the Hessian of f.

The expressions t_1 and t_2 are mentioned multiple times, but ideally each would only be translated to lower-level constructs once.

Reducing the IR by common computations is commonly known as common subexpression elimination (CSE). In order to apply CSE inside an EIN operator we must be able to compare computations in the EIN syntax efficiently and correctly. EIN notation adds a level of complexity to comparing EIN expressions since so much of the meaning of an expression is captured in the indices.

We could factor common subterms but it is more complicated than just looking at syntactic equality. In a single EIN operator, the same expression can be used multiple times but not sampled in the same way. For example, consider the expression:

$$e_{[\cdot/ij]}e_{[\cdot/k1]} - e_{[\cdot/0i]}e_{[\cdot/jk]} + (e_{[\cdot/ij]}e_{[\cdot/k1]})(\sum_l e_{[\cdot/ll]}).$$

If you recall, $e_{[ab/ij]}$ indicates that the expression e is parameterized with different indices. In this expression e is instantiated in different ways (we omit a, b for simplify). Variation in indices occur because different operators create different variable bindings. Comparing the EIN expressions structurally we can find terms that are exactly the same (such as $e_{[\cdot/ij]}$ and $e_{[\cdot/ij]}$). This approach is limited 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 the context of the whole EIN operator. It is with this context that we can distinguish between terms that just have shifted indices (such as $e_{[\cdot/ij]}$ and $e_{[\cdot/jk]}$), those that create a subset of the same computations (such as $\sum_l e_{[\cdot/ll]}$ and $e_{[\cdot/ij]}$), and those that are not equivalent (such as $e_{[\cdot/k1]}$, and $e_{[\cdot/0i]}$).

Our goal is to reduce the size of an EIN operator *and* reduce the number of computations created by an EIN operator. This section describes the two techniques we developed while preserving the mathematical meaning of the computation. The first technique, called *Split*, is used to split large EIN operators into a series of simple and small operators (Section 3.3.3). As we split we use hash consing to find equivalent computations.The second technique, called *Slice*, finds nonequivalent fields terms that create many of the same lower-level base computations (Section 3.3.4).

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 unclear. 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. Our implementation

decomposes large and complicated EIN operators into many small and simple ones that are easier to analyze.

The remainder of this section describes our compilation techniques. First, to enable these techniques we simplify EIN terms with some rewriting (Section 3.3.1) and impose a formal sense of shape to handle indices correctly (Section 3.3.2). We then describe the implementation details for the *Split* method (Section 3.3.3) and the *Slice* method (Section 3.3.4). Lastly, we present some examples of the methods (Section 3.3.5).

3.3.1 Shift

The goal of the *Shift* method is to move invariant terms outside a summation. Consider the term $\sum_{\sigma}(T_\alpha * e)$ if the property $\forall i \in \alpha.i \notin \sigma$ holds then we do the following rewrite.

$$\sum_{\sigma}(T_\alpha * e) \xrightarrow[shift]{} T_\alpha * \sum_{\sigma}(e)$$

This simple rewrite can help simplify expressions before the next optimizations passes. We use the notation $\xrightarrow[shift]{}$ to indicate the application of the shift method on an EIN expression.

We do an analysis on the structure of the EIN body. Most of the method is made up of a series of rewrites to move the summation operator. When there is an embedded summation (at least two variable indices bound at the summation operator) then the method applies more analysis. The analysis can also turn an embedded summation into two independent loop nests and remove summation loops.

Example The best way to rewrite or implement a computation can be unclear. Consider the operation (s*(v⊗x)) • u represented in the EIN syntax as

$$\xRightarrow[norm]{surface} \begin{aligned} \mathrm{t} = \;& \lambda(S,V,X,U)\langle e\rangle_i(\mathrm{s},\mathrm{v},\mathrm{x},\mathrm{u}) \\ e = \;& \sum_j (SV_iX_jU_j) \end{aligned}$$

We move the invariant terms outside the summation operator.

$$e \xrightarrow[shift]{} SV_i \sum_{\hat{j}} (X_j U_j)$$

By moving the invariant term outside the summation operator the dot product

$$\sum_{j} (X_j U_j) \xrightarrow[\text{Surface}]{EIN} x \bullet u$$

is recognizable and compiler has the option to generate vector code.

3.3.2 Shape

To enable some of the compilation techniques, we need to impose a concept of shape on an EIN expression, called *Eshape* and *Tshape*.

$$\begin{aligned} \mathit{Tshape}: &\quad EIN \text{ expression } \times \sigma \times \sigma \rightarrow \textsc{IndexVar list} \\ \mathit{Eshape}: &\quad EIN \text{ expression } \rightarrow \textsc{IndexVar list} \end{aligned}$$

The *Eshape* is an ordered list of free index variables used by the EIN expression. The *Tshape* puts that list in context of a full EIN operator. As input the *Tshape* function takes an EIN expression and two index maps and returns a list of indices. The first map is bound outside the EIN operator and the second map is bound by a summation.

Consider the following application of *Tshape*:

$$\left\langle \sum_{\sigma_1} e \ldots \right\rangle_{\sigma_0} \text{ and } \mathit{Tshape}(e, \sigma_0, \sigma_1) = \{\beta\}$$

$$\mathit{Tshape}(e, \sigma_0, \sigma_1) = \quad \forall i \in \mathit{Eshape}(e).\ \text{if } i \notin \sigma_0 \text{ and } i \notin \sigma_1 \text{ then } i \in \beta$$

The method *Eshape* does a case analysis of the body and returns the ordered free indices. The result is the list of variable indices used by e but bound outside of it.

The following are examples of extracting the *Eshape* from EIN expressions. The shape extracted needs to reflect simple operations A+B.

$$Eshape(A_{ijk} + B_{ijk}) = \{i, j, k\}$$

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

$$Eshape(F_i G_j) = \{i, j\}$$

the order that the indices appear Transpose(a⊗b),

$$Eshape(A_j B_i) = \{j, i\}$$

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

$$Eshape(\sum_i V_{ii} \circledast H) = \{\}$$

if the indices repeat in multiple terms modulate(a,b),

$$Eshape(A_{ij} B_{ij}) = \{i, j\}$$

remove indices bound by a summation as in a : b

$$Eshape(\sum_{ij} A_{ij} B_{ij}) = \{\}$$

or embedded summation $(V \circledast H) \cdot M$,

$$Eshape(\sum_j (V \circledast \frac{\partial}{\partial x_j} H) M_{ji}) = \{i\}$$

and with summation operators a ⊗(F· C),

$$Eshape(A_i(\sum_k F_k C_{kj})) = \{i, j\}.$$

For example, consider the following computation

```
tensor[d]a;
tensor[d,d]b;
field#k(d)[]F = v ⊛ h;
tensor[d,d]t2 = a ⊗ (∇ F· b);
```

After normalization the compiler represents the computation with a single operator:

$$\xRightarrow[norm]{surface} t_2 = \lambda(V, H, A, B)\left\langle A_i \sum_k ((V \circledast (\frac{\partial}{\partial x_k} \diamond H)) B_{kj}) \right\rangle_{\hat{i}\hat{j}} (\mathrm{v}, \mathrm{h}, \mathrm{a}, \mathrm{b}) \tag{3.8}$$

We define a subterm e_2 as

$$e_2 = \sum_{\hat{k}}((V \circledast (\frac{\partial}{\partial x_k} \diamond H))B_{kj})$$

The *Tshape* of the subterm e_2 is j ($Tshape(e_2, \{i, j\}, \{\}) = \{j\}$). Variable index k is not included because it is bound inside the sub-term.

3.3.3 Split

A single EIN operator can be used to concisely represent many different computations. These figures can easily become large and do not maintain sharing. The translation of these EIN operators to lower-level code can create a lot of common constructs.

To eliminant the number of redundant constructs created we want to enable comparisons in the EIN syntax. Comparing EIN expressions is insufficient since the same computation can have different variable indices (structurally unequal). Therefore, the *Split* method compares EIN operators to identify common computations.

The *Split* method decomposes a large EIN operator into smaller and simple ones. *Split* finds an EIN expression and creates a new EIN operator around it. It is then possible to compare the new EIN operator to existing ones and enable CSE. In this section, we describe the implementation details followed by an example. We use the notation $\underset{split}{\Longrightarrow}$ to indicate the application of the method split to an EIN operator.

The *Split* method does the following decomposition.

$$\mathrm{t}_2 = \lambda A, B\left\langle \sum_{\sigma_1} e_1 \odot e \right\rangle_{\sigma_0} (\mathrm{a}, \mathrm{b}) \quad \underset{split}{\Longrightarrow} \quad \begin{aligned} \mathrm{t}_1 &= \lambda B\langle e_1' \rangle_{\hat{\alpha}}(\mathrm{b}) \\ \mathrm{t}_2 &= \lambda A, T\left\langle \sum_{\sigma_1} T_\alpha \odot e \right\rangle_{\sigma_0} (\mathrm{a}, \mathrm{t}_1) \\ &\quad \text{where } Tshape(e_1) = \alpha \end{aligned}$$

An EIN operator t_2 is decomposes into two EIN operators. The inner EIN expression e_1

is pulled out to create a new EIN operator t_1. In the original EIN operator t_2 the EIN expression e_1 is replaced with a tensor variable T_α. The implementation is described in more detail next using the variable names in this example.

split	: *binding* $\rightarrow$ *bindings*.
is_Split	: EIN expression $\times \sigma \rightarrow$ *bool*
split_core	: EIN expression $\times \sigma \rightarrow$ *bindings*
	where *binding* = variable $\times$ EIN operator $\times$ arguments

As input the split () method receives a variable, an EIN operator, and arguments. The input to split () has the form:

$$t_2 = \lambda A, B \left\langle \sum_{\sigma_1} e_1 \odot e \right\rangle_{\sigma_0} (\text{args}).$$

We use method is_Split () to scan the EIN body to find an embedded EIN subexpression. We call function split_core () when we find a term e_1 to lift out. In the following we provide a sketch of split_core ().

1. The implementation uses *Tshape* (introduced in Section 3.3.2)

   $$Tshape(e_1, \sigma_0, \sigma_1) = \{\alpha\}$$

   where e_1 is the subexpression that is pulled out and σ_1 is a list of indices bound by a summation. The outshape $\hat{\alpha}$ is found by looking up the indices α in σ_0.

2. A new operator is created for the EIN expression e_1. We remap the body's parameter ids and indices to only those included in the new EIN operator.

   $$\lambda B. \left\langle e' \right\rangle_{\hat{\alpha}} (\text{b})$$

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

ing details. The method can then check the hash table to see if the computation has been derived previously before assigning a variable (t_1).

$$\mathrm{t}_1 = \lambda B.\left\langle e' \right\rangle_{\hat{\alpha}}(\mathrm{b})$$

3. The original EIN operator t_2 is rewritten. The expression e_1 is replaced with term T_α in the EIN body to create $T_\alpha \odot e$. A reference to the t_1 is added.

$$\mathrm{t}_2 = \lambda A, T\left\langle \sum_{\sigma_1} T_\alpha \odot e \right\rangle_{\sigma} (\mathrm{a}, \mathrm{t}_1)$$

As a result the subexpression is lifted out of the original EIN operator.

Splitting example Continuing the example from the previous section (3.8).

$$\underset{split}{\Longrightarrow} \begin{array}{l} \mathrm{t}_1 = \lambda(V, H, B)\left\langle \sum_j ((V \circledast (\frac{\partial}{\partial x_j} \diamond H))B_{ji}) \right\rangle_{\hat{i}} (\mathrm{v}, \mathrm{h}, \mathrm{b}) \\ \mathrm{t}_2 = \lambda(A, T)\left\langle A_i T_j \right\rangle_{\hat{i}\hat{j}} (\mathrm{a}, \mathrm{t}_1) \end{array}$$

The *Split* method pulls out the sub-term e_2 (3.8). A new EIN operator is created using the sub-term e_2 as its body. The variable indices and parameters are remapped accordingly. In the original operator, the sub-term e_2 is replaced with a tensor variable $T_{Tshape(e2)}$.

Substitution and Split In a way, the *Split* method is the inverse of the Substitution method (Section 3.2.1). 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 representation.

3.3.4 Slice

Field terms can be transformed into a large number of lower-level constructs (during field reconstruction Section 2.3). Different sliced fields can generate many of the same lower-level constructs (base computations between image data and kernels). To avoid this redundancy, we use *Slice* to transform a sliced field term, called OptSlice), into a pair of an unsliced field probe and a tensor-valued EIN operator that extracts the components of the probe that are of interest. We use the notation $\underset{slice}{\Longrightarrow}$ to indicate the application of the method slice on the EIN operator.

Definition 4 (OptSlice)**.** *As a reminder, a sliced field (Definition 3 on page 22) is a field term that has at least one constant index (c). The components are distinct (*i.e., $F_{01} \neq F_{11}$*) and independent of each other but they depend on the same source data. A OptSlice term wraps a probe operator around a sliced field term and has the following structure:* $\frac{\partial}{\partial x_\beta} \diamond F_{c\alpha}(x)$.

We scan the the body of an EIN operator to identify OptSlice terms (Definition 4 on page 55). The slice method creates the following decomposition of an EIN operator.

$$\mathrm{g} = \lambda F \left\langle \frac{\partial}{\partial x_\beta} \diamond F_{c\alpha}(x) \right\rangle_{\hat{\alpha}\hat{\beta}} (\mathrm{f}) \qquad \underset{slice}{\Longrightarrow} \qquad \begin{array}{l} \mathrm{g} = \lambda T \left\langle T_{c\alpha\beta} \right\rangle_{\hat{\alpha}\hat{\beta}} (\mathrm{t}) \\ \mathrm{t} = \lambda F \left\langle \frac{\partial}{\partial x_\beta} \diamond F_\mu(x) \right\rangle_{\hat{\mu}\hat{\beta}} (\mathrm{f}) \\ \text{where } |\alpha| + 1 = |\mu| \\ \text{and } \forall i \in |c\alpha| \text{ if } (i == 0) \\ \quad \text{then } u[i] = n_0 (\text{described by field type}) \\ \quad \text{else } u[i] = \alpha[i-1] \end{array}$$

We create a new EIN operator (t) to represent unsliced versions of the field term. The field term $F_{c\alpha}$ is changed by converting the constant index to a variable index $F_\mu(x)$. The slice operation is pushed to a tensor term ($T_{c\alpha\beta}$). The new operator g samples a tensor, created as a result of the probe operator.

Example Consider the determinant of a second-order tensor field m

```
field#k(3)[3,3]m;
tensor[3]p;
tensor[3,3]t = m(p);
tensor[]g = det(m)(p);
```

transformed to EIN as

$$\xrightarrow[norm]{surface} \begin{array}{lll} \mathrm{t} = \lambda(F,T) & \left\langle F_{ij}(T)\right\rangle_{\hat{i}\hat{j}} & (\mathrm{m},\mathrm{p}) \\ \mathrm{g} = \lambda(F,T) & \left\langle \sum_{ijk} \mathcal{E}_{ijk} F_{0i}(T) F_{1j}(T) F_{2k}(T) \right\rangle_{\hat{i}\hat{j}} & (\mathrm{m},\mathrm{p}) \end{array} \tag{3.9}$$

The compiler creates two EIN operators to represent the computation. EIN operator t can not be further reduced. EIN operator g is more complicated and can be split into multiple pieces. The *Split* method creates EIN operators for each field term (F_{0i}, F_{1j}, F_{2k}).

$$\xrightarrow[split]{} \begin{array}{lll} \mathrm{t} = \lambda(F,T) & \left\langle F_{ij}(T)\right\rangle_{\hat{i}\hat{j}} & (\mathrm{m},\mathrm{p}) \\ \mathrm{g} = \lambda(A,B,C) & \left\langle \sum_{ijk} \mathcal{E}_{ijk} A_i B_j C_k \right\rangle & (\mathrm{t}_0,\mathrm{t}_1,\mathrm{t}_2) \\ \mathrm{t}_0 = \lambda(F,T) & \langle F_{0i}(T)\rangle_{\hat{i}} & (\mathrm{m},\mathrm{p}) \\ \mathrm{t}_1 = \lambda(F,T) & \langle F_{1i}(T)\rangle_{\hat{i}} & (\mathrm{m},\mathrm{p}) \\ \mathrm{t}_2 = \lambda(F,T) & \langle F_{2i}(T)\rangle_{\hat{i}} & (\mathrm{m},\mathrm{p}) \end{array} \tag{3.10}$$

A naive implementation of EIN would stop here. The individual terms $(F_{ij}, F_{0i}, F_{1i}, F_{2i})$ are unique and could not be further reduced with *Slice*. In the next phase of the compiler, each field term $(F_{ij}, F_{0i}, F_{1i}, F_{2i})$ is transformed to lower-level constructs that will have many common computations.

Our goal is to reduce the number of fields terms that need to be transformed. To do

that, we apply the split method. Slice defines the following operators (with some rewriting).

$$\underset{slice}{\Longrightarrow} \begin{array}{ll} \mathrm{g} = \lambda(T) & \left\langle \sum_{ijk} \mathcal{E}_{ijk} T_{0i} T_{1j} T_{2k} \right\rangle_{\hat{i}\hat{j}} \quad (\mathrm{t}) \\ \mathrm{t} = \lambda(F,T) & \left\langle F_{ij}(T) \right\rangle_{\hat{i}\hat{j}} \quad (\mathrm{m},\mathrm{p}) \end{array}$$

There is only one field term F_{ij} instead of four. This one field term does the bulk of field computations just once. Consider another example, if the program computed the Hessian of the determinant

tensor[3,3]u = $\nabla \otimes \nabla$ det(m);

then the following ten EIN operators are created:

$$\underset{split}{\overset{surface}{\Longrightarrow}} \begin{array}{l} \text{//Defined in } \textit{Equation } 3.10 \\ \mathrm{t}, \mathrm{t}_0, \mathrm{t}_1, \text{ and } \mathrm{t}_2 = \ldots \\ \text{//First derivative} \\ \mathrm{g}_0 = \lambda P \left\langle \frac{\partial}{\partial x_j} \diamond F_{0i}(T) \right\rangle_{\hat{i}\hat{j}} (\mathrm{m},\mathrm{p}) \\ \mathrm{g}_1 = \lambda P \left\langle \frac{\partial}{\partial x_j} \diamond F_{1i}(T) \right\rangle_{\hat{i}\hat{j}} (\mathrm{m},\mathrm{p}) \\ \mathrm{g}_2 = \lambda P \left\langle \frac{\partial}{\partial x_j} \diamond F_{2i}(T) \right\rangle_{\hat{i}\hat{j}} (\mathrm{m},\mathrm{p}) \\ \text{//Second derivative} \\ \mathrm{h}_0 = \lambda P \left\langle \frac{\partial}{\partial x_{jk}} \diamond F_{0i}(T) \right\rangle_{\hat{i}\hat{j}\hat{k}} (\mathrm{m},\mathrm{p}) \\ \mathrm{h}_1 = \lambda P \left\langle \frac{\partial}{\partial x_{jk}} \diamond F_{1i}(T) \right\rangle_{\hat{i}\hat{j}\hat{k}} (\mathrm{m},\mathrm{p}) \\ \mathrm{h}_2 = \lambda P \left\langle \frac{\partial}{\partial x_{jk}} \diamond F_{2i}(T) \right\rangle_{\hat{i}\hat{j}\hat{k}} (\mathrm{m},\mathrm{p}) \end{array} \tag{3.11}$$

Our method reduces the ten field terms down to three unique field terms (one for each level of differentiation).

$$\begin{array}{rll} & \mathrm{t} = \lambda(F,T) & \left\langle F_{ij}(T) \right\rangle_{\hat{i}\hat{j}} \quad (\mathrm{m},\mathrm{p}) \\ \xRightarrow[slice]{} & \mathrm{g} = \lambda(F,T) & \left\langle \frac{\partial}{\partial x_k} \diamond F_{ij}(T) \right\rangle_{\hat{i}\hat{j}\hat{k}} \quad (\mathrm{m},\mathrm{p}) \\ & \mathrm{h} = \lambda(F,T) & \left\langle \frac{\partial}{\partial x_{kl}} \diamond F_{ij}(T) \right\rangle_{\hat{i}\hat{j}\hat{k}\hat{l}} \quad (\mathrm{m},\mathrm{p}) \end{array}$$

The difference can be more significant when considering more complicated tensor computations. The slice method creates a smaller representation of the computation while maintaining mathematical meaning.

Slice and Split comparison The goal of *Split* is to find common computations embedded in an EIN expression. The goal of slice is to find a specific type of computation and create a general EIN operator (that can be sampled). In other words, *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).

3.3.5 Examples

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

Index-based optimization Inlining exposes the opportunity for other optimizations. The compiler takes 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

$$\begin{array}{rlll} & \mathrm{t}_1 & = \lambda(F) & \left\langle \frac{\partial}{\partial x_{jk}} F \right\rangle_{\hat{j}\hat{k}} \quad (\varphi) \\ \underset{init}{\Longrightarrow} & \mathrm{t}_2 & = \lambda(T) & \left\langle \sum_i T_{ii} \right\rangle \quad (\mathrm{t}_1) \end{array}$$

During substitution the outer indices for t_1 (j, k) are mapped to indices in term T_{ii}.

$$\underset{subst}{\Longrightarrow} \quad \mathrm{t}_2 = \lambda(F) \left\langle \sum_i \frac{\partial}{\partial x_{ii}} F \right\rangle (\varphi)$$

where j is mapped to i and k is mapped to i

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

Tensor Identity The following is an example of a tensor term being rewritten using our methods and revealing a tensor identity. Consider the expression $(\mathrm{a}\times\mathrm{b}) \cdot (\mathrm{c}\times\mathrm{d})$, which is represented with a single EIN operator as

$$\begin{array}{rl} & t = \lambda(A, B, C, D)\langle e \rangle(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}) \\ \underset{subst}{\overset{surface}{\Longrightarrow}} & e = \sum_i ((\sum_{jk} \mathcal{E}_{ijk} a_j b_k)(\sum_{lm} \mathcal{E}_{ilm} c_l d_m)) \end{array}$$

The body is normalized using the rewrites in Equation 3.5 and Equation 3.6 and then unused indices are removed to produce:

$$e \xrightarrow[rule]{}{}^{*} \sum_{jk} ((A_j B_k C_j D_k) - (A_j B_k C_k D_j))$$

The *Shift* method moves the invariant terms outside of the outer loop.

$$e \xrightarrow[shift]{} (\sum_j (A_j C_j) \sum_k (B_k D_k)) - (\sum_j (A_j D_j) \sum_k (B_k C_k))$$

The *Split* method then creates four simple EIN operators.

$$\xRightarrow[split]{} \begin{aligned} \mathrm{w} &= \lambda(A, C)\left\langle \sum_i (A_i C_i) \right\rangle (\mathrm{a}, \mathrm{c}) \\ \mathrm{x} &= \lambda(B, D)\left\langle \sum_i (B_i D_i) \right\rangle (\mathrm{b}, \mathrm{d}) \\ \mathrm{y} &= \lambda(A, D)\left\langle \sum_i (A_i D_i) \right\rangle (\mathrm{a}, \mathrm{d}) \\ \mathrm{z} &= \lambda(B, C)\left\langle \sum_i (B_i C_i) \right\rangle (\mathrm{b}, \mathrm{c}) \\ \mathrm{t} &= \lambda(W, X, Y, Z)\langle (W * X) - (Y * Z) \rangle (\mathrm{w}, \mathrm{x}, \mathrm{y}, \mathrm{z}) \end{aligned}$$

As a result the compiler has discovered the tensor identity:

$$(\mathrm{a} \times \mathrm{b}) \cdot (\mathrm{c} \times \mathrm{d}) \longrightarrow_{direct-style} (\mathrm{a} \cdot \mathrm{c}) * (\mathrm{b} \cdot \mathrm{d}) - (\mathrm{a} \cdot \mathrm{d}) * (\mathrm{b} \cdot \mathrm{c}).$$

Post field reconstruction The next phase of the compiler translates EIN operators into scalar and vector operators. The way a computation is written using the EIN syntax can determine how it is translated. A bulky EIN operator might translate entirely into scalar operators, while a simple one is easier to translate to vector operators. In the following we show three different ways to represent a computation in EIN and measure the compile and run times of each representation.

Consider the transformation out = P $\cdot t_0 \cdot$ P and probed field expression e from Equation 2.5. The following three equations defines three ways to represent this computation. The first approach is by representing the entire computation in one EIN operator

$$\Rightarrow \quad \mathrm{out} = \quad \lambda(v, h, n, f, P)\left\langle \sum_{kl=0}^{1} e_{[ijkl/klmn]} P_{ik} P_{jl} \right\rangle_{\hat{i}\hat{j}} (\mathrm{img}, \mathrm{bspln3}, \mathrm{n}, \mathrm{f}, \mathrm{P}) \tag{3.12}$$

The second approach applies *Split* to Equation 3.12 and as a result creates two EIN operators.

$$
\begin{aligned}
\mathrm{t}_0 = &\quad \lambda(v,h,n,f)\Big\langle e_{[ijkl/ijkl]}\Big\rangle_{\hat{i}\hat{j}}(\mathrm{img},\mathrm{bspln3},\mathrm{n},\mathrm{f}) \\
\underset{split}{\Longrightarrow}\ \mathrm{out} = &\quad \lambda(T,P)\langle e_2\rangle_{\hat{i}\hat{j}}(\mathrm{t}_0,P) \\
e_2 = &\quad \sum_{kl=0}^{2} T_{kl}P_{ik}P_{jl}
\end{aligned}
\tag{3.13}
$$

The third approach applies *Shift* to EIN expression e_2 before splitting

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

and as a result Equation 3.13 becomes three EIN operators.

$$
\begin{aligned}
\mathrm{t}_0 = &\quad \lambda(v,h,n,f)\Big\langle e_{[ijkl/ijkl]}\Big\rangle_{\hat{i}\hat{j}}(\mathrm{img},\mathrm{bspln3},\mathrm{n},\mathrm{f}) \\
\underset{split}{\Longrightarrow}\ \mathrm{t}_1 = &\quad \lambda(T,P)\left\langle \sum_{k=0}^{1} T_{ik}P_{jk} \right\rangle_{\hat{i}\hat{j}} (\mathrm{t}_0,P) \\
\mathrm{out} = &\quad \lambda(T,P)\left\langle \sum_{k=0}^{1} P_{ik}T_{kj} \right\rangle_{\hat{i}\hat{j}} (\mathrm{t}_1,P)
\end{aligned}
\tag{3.14}
$$

Equation 3.14 uses simple terms to represent the computation. This is easier for the compiler to analyze. As a result, the representation also compiles and runs faster (Section 3.4.3).

3.4 Benchmarks

Implementing the EIN IR has caused many technical and compilation challenges. Addressing those challenges has enabled a richer language that can be used to support richer visualization programs (Section 7.2). It is useful to evaluate the impact of our work. In this section, we present three sets of benchmark results. The first is an evaluation of implementing EIN over the original direct-style compiler. The second measures the effect of the techniques described in Section 3.3. The third is a comparison of using the higher-order features of the language versus equivalent first-order implementations.

3.4.1 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 increasing mathematical complexity. Some of the visualization concepts that inspired the benchmarks are described in Section 7.2. Benchmarks "illust-vr," "lic2d," "Mandelbrot," "ridge3d," and "vr-lite-cam" are small examples that could run on the original compiler [18]. The benchmarks "mode," "canny," and "moe" are used to create figures in previous work [44]. "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 [12].

The benchmarks "dec-crest," "dec-grad," "rsvr," and"mode-rig' were not featured in previous work, because they previously could not compile. 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 of a field. Unlike the previous benchmarks, these program are not visualization programs. Their run times are negligible and are omitted.

3.4.2 Impact on implementing Diderot

Our experiments consist of eighteen benchmarks run on three versions of the compiler. The first is the original direct-style compiler that does not use EIN. 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

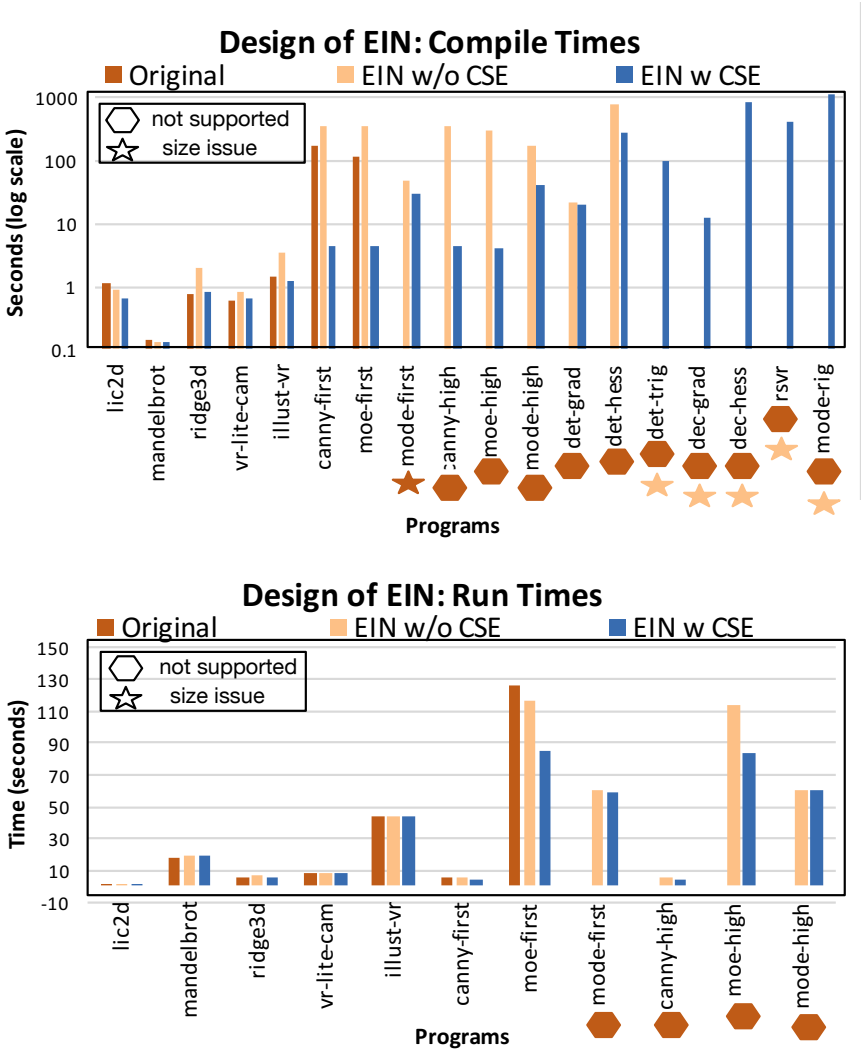

Figure 3.3: The "Original" version of the compiler does not use the EIN IR. "EIN with restrictions" is the more naive implementation of EIN. "EIN " is the baseline with the EIN IR with full optimizations applied. Fully implementing EIN allows more programs to compile than previously possible.

least two versions, we also measure the run times. We use a hexagon symbol to indicate that the compiler does not have the feature support to run this program. We use a star symbol to indicate a compiler crash.

Figure 3.3 compares the application of EIN with the "original compiler". The "original compiler" did not have the feature support required to run many of the benchmarks. The development of EIN has affected the type and complexity of programs that we can write in Diderot.

Figure 3.3 compares the application of EIN with full or restricted level of optimizations. We can summarize that fully implementing the optimizations enables Diderot to compile programs that otherwise cannot compile. At worst EIN with CSE is comparable to original compiler, but mostly faster, and is more expressive

3.4.3 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 compiled, we want to evaluate the cost and benefit they might impose on the programs that could already compile. In the following, we evaluate the effectiveness of these techniques together and isolated at different steps in the compiler.

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 3.4 measures the effectiveness of applying *Split* and *Slice* on a high-IR EIN operator. The *Slice* technique is necessary to compile three of the thirteen benchmarks. *Split* is the most consequential technique. Limiting the application of the method stops five of the programs from being able to compile. Techniques Split and Slice do not add take a considerably longer time to run.

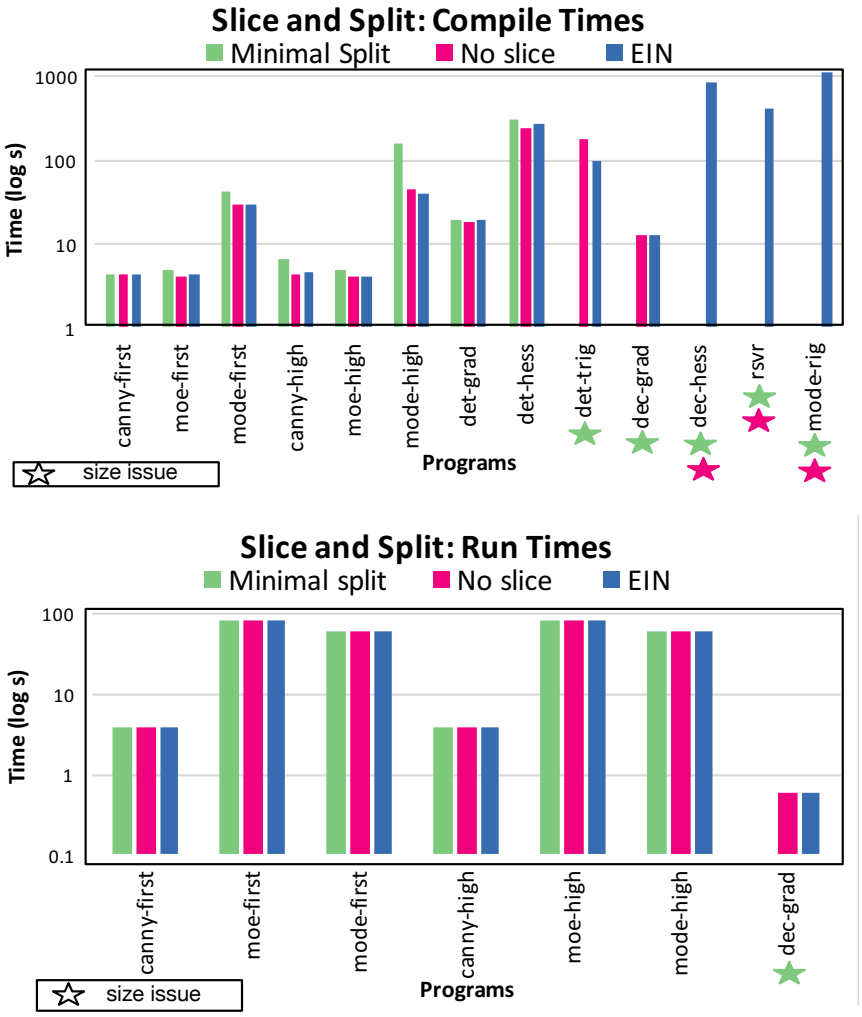

Figure 3.4: Compile and run time measurements when implementing *Slice* and *Split* on High-IR. Doing no amount of splitting prevents most of these programs from compiling so instead we measure its impact by limiting it, "Minimal *Split*". EIN is the baseline with both *Split* and *Slice* enabled.

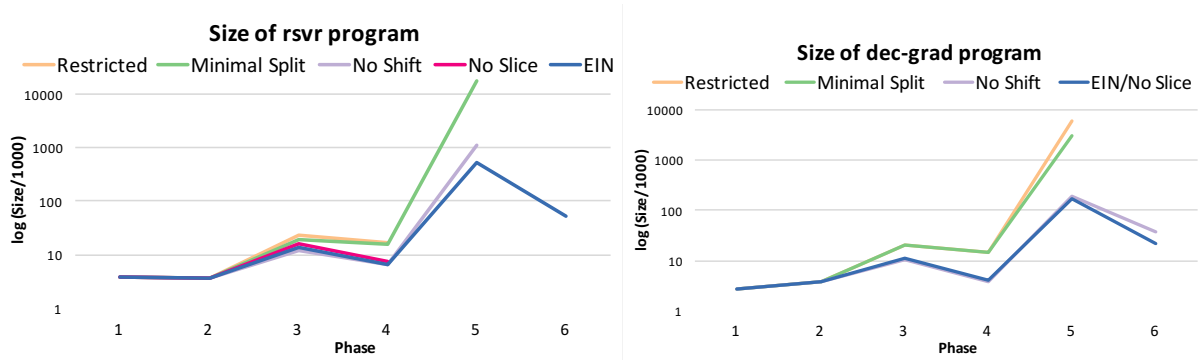

Figure 3.5: The graphs shows the size of the dec-grad (left) and rsvr (right) programs at different phases in the compiler. "EIN" is the baseline with full optimizations.

Size Reduction To take a closer look at how the optimizations are affecting the size of programs, we measure the size of a Diderot program at six different phases of the compiler under five different settings. Figure 3.5 illustrates the size of two programs at different phases of the compiler. 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.

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

Application to lower-order constructs We measured the effect of optimizations at a later phase of the compiler. Section 3.3.5 demonstrates the transformation of differentiation indices in a reconstructed field term. We can write this computation in three different ways: in place Equation 3.12, by applying *Split* Equation 3.13, or by applying *Shift* and *Split* together Equation 3.14. Figure 3.6 measures the impact of these optimizations on this computation.

The implementation of the techniques directly affect compile time. Applying optimizations *Shift* and *Split* together offers a consistent speed-up on the execution time and compile

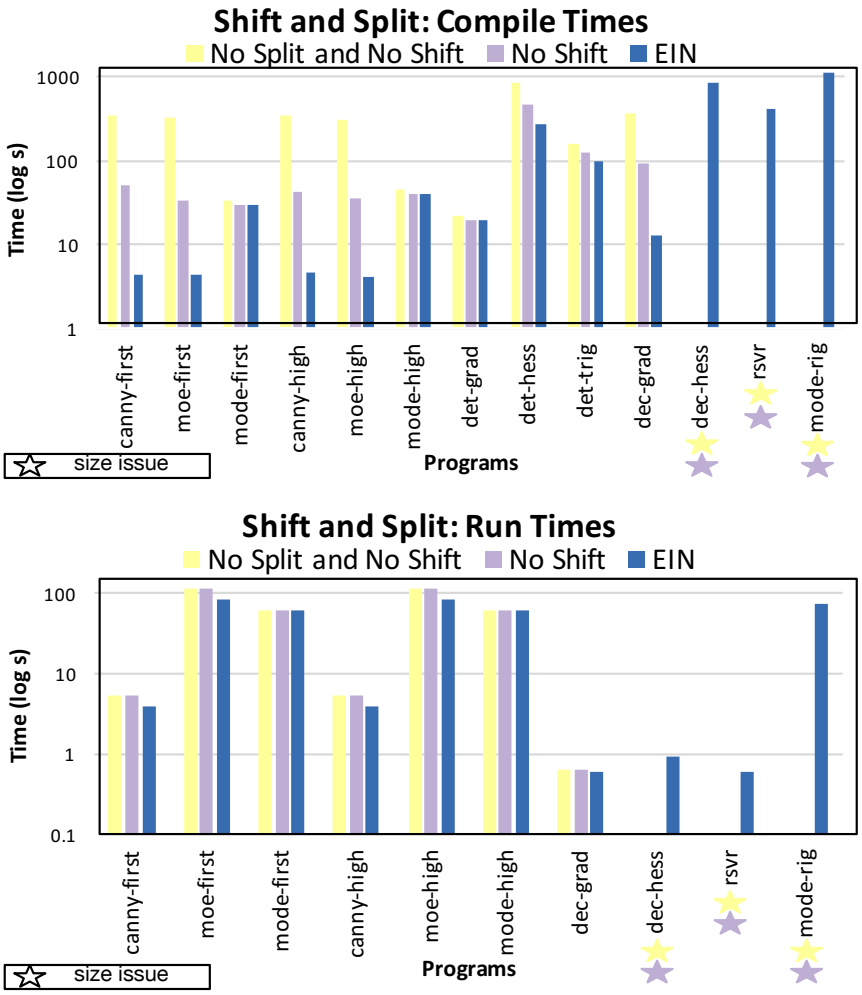

Figure 3.6: Compile and run time measurements for implementing *Shift* and *Split* techniques on reconstructed field terms. EIN baseline includes the application of both *Shift* and *Split*.

time for all thirteen benchmarks. Five programs experienced at least a 20-times improvement in compile time speed-up and four of the benchmarks offered at least a 1.3 speed up on execution time while the rest saw no change in the execution time.

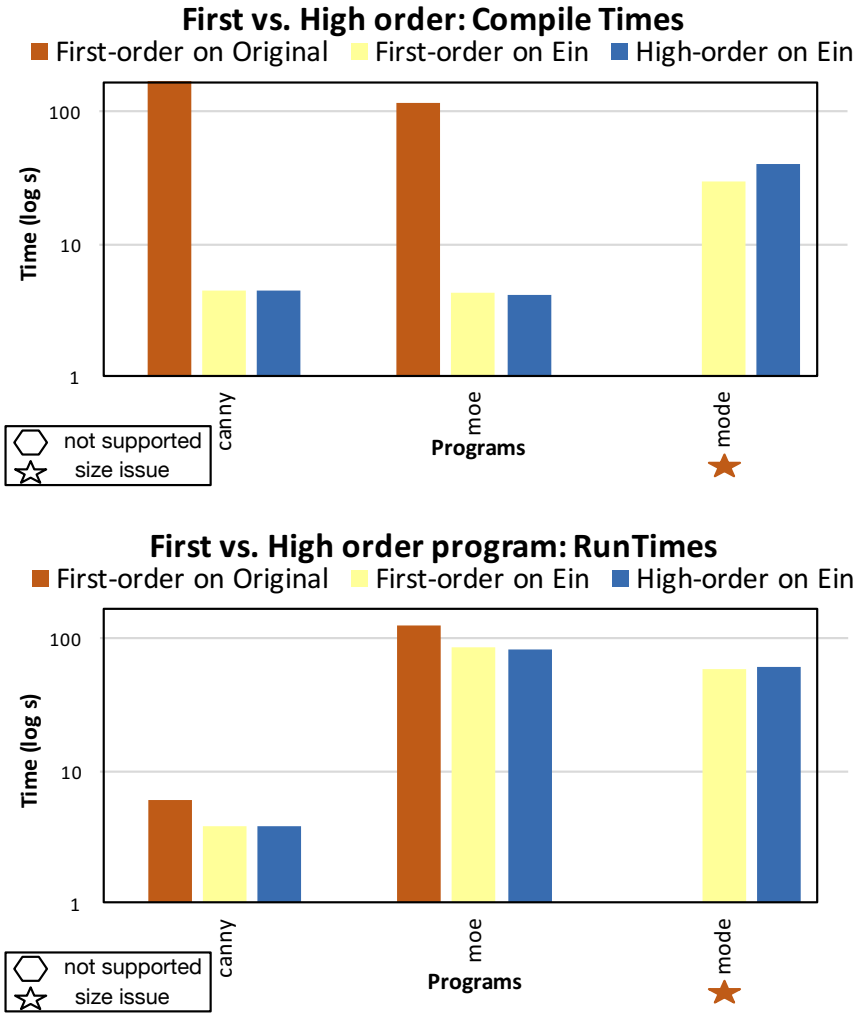

Figure 3.7: A comparison of hand-derived first-order programs with their high-order equivalent. The original compiler can only compile the first-order versions, while the compiler with EIN can compile both versions.

3.4.4 First-Order versus Higher-Order

In this experiment we wrote two version of three programs. For each program the higher-order version of the program *lifted* tensor operators to the field level, while first-order versions

only uses tensor operators on tensors. The higher-order version of the program are easier and faster to write. Their first-order counterparts require more lines of code and require that the user do tricky derivations by hand, which is time-consuming, tedious, and error-prone. Figure 3.7 reports the compile and run time for two versions of the programs.

This experiment actually tests two things: (1) the original compiler versus the EIN compiler and (2) comparing the first-order and higher-order versions of a program using the same compiler. The first order programs compiled and ran faster on EIN than on the original compiler. The first-order program "mode-vr" slightly compiled faster than the high-order counterpart but the rest of the programs are comparable. Ultimately, it is more feasible 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 3.2.2) plays a key rôle in the compilation of Diderot programs. The transformation is 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 formal analysis. We define a type system for EIN operators in Section 4.1 and provide evaluation rules in Section 4.2. We prove that normalization preserves types. We also prove that the rewrite system is terminating in Section 4.3. We include full proofs in the paper [19] and in the appendix.

4.1 Type Preservation

4.1.1 Typing EIN Operators

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

$$
\begin{array}{rcll}
\theta & ::= & \mathbf{Ten}[d_1, \ldots, d_n] & \text{tensors} \\
& | & \mathbf{Fld}(d)[d_1, \ldots, d_n] & \text{fields} \\
& | & \mathbf{Img}(d)[d_1, \ldots, d_n] & \text{images} \\
& | & \mathbf{Krn} & \text{kernels}
\end{array}
$$

An EIN operator $\lambda\, \bar{x}\langle e \rangle_\sigma$ can then be given a function type $(\theta_1 \times \cdots \times \theta_n) \to \theta$, where θ is either $\mathbf{Ten}[d_1, \ldots, d_n]$ or $\mathbf{Fld}(d)[d_1, \ldots, d_n]$ and σ is $1 < i_1 < d_1, \ldots, 1 < i_n < d_n$. The EIN expression (e) is the body of the operator, cannot be given a type θ, however since it represents a computation indexed by σ. Thus the type system for EIN expressions must track the index space as part of the context.

$$[\text{TYJUD}_1]\frac{\Gamma(T) = \mathbf{Ten}[d_1, \ldots, d_n] \qquad |\alpha| = n \qquad \sigma \vdash \alpha < [d_1, \ldots d_n]}{\Gamma, \sigma \vdash T_\alpha : (\sigma)\mathcal{T}}$$

$$\frac{\Gamma(F) = \mathbf{Fld}(d)[d_1, \ldots, d_n] \qquad |\alpha| = n \qquad \sigma \vdash \alpha < [d_1, \ldots d_n]}{\Gamma, \sigma \vdash F_\alpha : (\sigma)\mathcal{F}^d}$$

$$[\text{TYJUD}_2]\frac{\begin{array}{c}\Gamma(V) = \mathbf{Img}(d)[d_1, \ldots, d_n] \qquad \Gamma(H) = \mathbf{Krn} \\ |\alpha\beta| = n \qquad \sigma \vdash \alpha\beta < [d_1, \ldots d_n]\end{array}}{\Gamma, \sigma \vdash V_\alpha \circledast H^\beta : (\sigma)\mathcal{F}^d}$$

$$[\text{TYJUD}_3]\frac{i \notin \text{dom}(\sigma) \qquad \sigma' = \sigma[i \mapsto (1, n)] \quad \Gamma, \sigma' \vdash e : (\sigma')\tau_0}{\Gamma, \sigma \vdash \sum_{i=1}^{n} e : (\sigma)\tau_0}$$

$$[\text{TYJUD}_4]\frac{\sigma(i) = d \qquad \sigma' = \sigma \setminus i \qquad \Gamma, \sigma' \vdash e : (\sigma')\mathcal{F}^d}{\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} e : (\sigma)\mathcal{F}^d}$$

$$[\text{TYJUD}_5] \ \frac{i, j \in dom(\sigma)}{\Gamma, \sigma \vdash \delta_{ij} : (\sigma)\mathcal{T}} \qquad \frac{\Gamma, \sigma \vdash ok}{\Gamma, \sigma \vdash \delta.\delta. : (\sigma)\mathcal{T}}$$

$$[\text{TYJUD}_5] \ \frac{\sigma' = \sigma[j \mapsto (1, d)]/\ i \qquad \Gamma, \sigma' \vdash e : (\sigma')\tau_0}{\Gamma, \sigma \vdash (\delta_{ij} * e) : (\sigma)\tau_0}$$

$$[\text{TYJUD}_6] \ \frac{\forall i \in \alpha.i \in dom(\sigma)}{\Gamma, \sigma \vdash \mathcal{E}_\alpha : (\sigma)\mathcal{T}} \qquad \Gamma, \sigma \vdash \mathcal{E}_{ijk}\mathcal{E}_{ilm} : (\sigma)\mathcal{T}$$

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

Figure 4.1: Typing Rules for each EIN expression.

$$[\text{TYJUD}_7]\frac{\Gamma, \sigma \vdash \delta_{ij} : \tau \quad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash \delta_{ij} @ x : \tau} \quad \frac{\Gamma, \sigma \vdash \mathcal{E}_\alpha : \tau \quad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash \mathcal{E}_\alpha @ x : \tau} \quad \frac{\Gamma, \sigma \vdash e : (\sigma)\mathcal{F}^d \quad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash e @ x : (\sigma)\mathcal{T}}$$

$$[\text{TYJUD}_8] \ \frac{\Gamma, \sigma \vdash e : (\sigma)\mathcal{T}}{\Gamma, \sigma \vdash \mathbf{lift}_d(e) : (\sigma)\mathcal{F}^d}$$

$$[\text{TYJUD}_9]\frac{\Gamma, \sigma \vdash e : ()\tau_0 \ \ \odot_1 \in \{\surd, -, \kappa, \exp, (\cdot)^n\}}{\Gamma, \sigma \vdash \odot_1(e) : ()\tau_0}$$

$$[\text{TYJUD}_{10}]\frac{\Gamma, \sigma \vdash e_1 : \tau \ \ \Gamma, \sigma \vdash e_2 : \tau \ \ \odot_2 \in \{+, -\}}{\Gamma, \sigma \vdash (e_1 \odot_2 e_2) : \tau} \quad \frac{\Gamma, \sigma \vdash e : \tau}{\Gamma, \sigma \vdash -e : \tau}$$

$$[\text{TYJUD}_{11}]\frac{\Gamma, \sigma \vdash e_1 : \tau \ \ \Gamma, \sigma \vdash e_2 : \tau}{\Gamma, \sigma \vdash (e_1 * e_2) : \tau}$$

$$[\text{TYJUD}_{12}]\frac{\Gamma, \sigma \vdash e_1 : (\sigma)\tau_0 \ \ \Gamma, \sigma \vdash e_2 : ()\tau_0}{\Gamma, \sigma \vdash \frac{e_1}{e_2} : (\sigma)\tau_0}$$

Figure 4.2: Typing Rules for each EIN expression.

We define the syntax of indexed EIN-expression types as

$$\tau_0 ::= \mathcal{T} \mid \mathcal{F}^d$$
$$\tau ::= (\sigma)\tau_0$$

where $(\sigma)\mathcal{T}$ is the type of indexed tensors and $(\sigma)\mathcal{F}^d$ is the type of indexed d-dimensional fields. We define our typing contexts as $\Gamma, \sigma \in (\textsc{Var} \xrightarrow{\text{fin}} \tau)^* \times (\textsc{IndexVar} \xrightarrow{\text{fin}} (\mathbb{Z} \times \mathbb{Z}))^*$. The typing context Γ, σ includes both the index map and an assignment of types to non-index variables.

With Γ we key the map with a variable. The notation

$$\Gamma(V) = \mathbf{Img}(d)[d_1, \ldots, d_n]$$

indicates that we can look up parameter id V in Γ and find the resulting type.

We first introduced σ in Section 2.1. We key the map with an index $\sigma \in (\textsc{IndexVar} \xrightarrow{\text{fin}}$

$(\mathbb{Z} \times \mathbb{Z}))^*$. To recall, the notation $i : n$ represents the upper boundary $1 < i < n$. We use notation

$$\sigma(i) = n$$

to indicate that we can look up variable (i) in σ and the upper bound of the variable is n. It is helpful to view σ as defining a finite map from index variables to the size of their range. To indicate the addition of a binding we use "$\sigma = \sigma'[i \mapsto (1, n)]$". The domain of σ is a sequence, which has to be disjoint ($\text{dom}(\sigma) = \{i_1, \ldots, i_n\}$). We use $i \notin \text{dom}(\sigma)$ to show that i is not in σ. We use "$\sigma = \sigma' \setminus i$" to indicate that i is not in σ' but it is in σ.

We state $\vdash \Gamma, \sigma$ **ok** to show that the environment is okay and the following apply

- with σ we key the map with an index and index variables do not repeat $\in \text{dom}(\sigma)$.
- in Γ we key the map with a unique variable parameter.

We define judgement form $\Gamma, \sigma \vdash e : \tau$ to mean that if the environment is okay then EIN expression e has type τ.

We define the judgement $\sigma \vdash \alpha < [d_1, \ldots d_n]$ as a shorthand for the following judgement.

$$\frac{\forall \mu_i \in \alpha, \text{ either } \mu_i \in \mathbb{N} \text{ and } 1 \leq \mu_i \leq d_i \text{ or } \sigma(\mu_i) = d_i}{\sigma \vdash \alpha < [d_1, \ldots d_n]}$$

Recall that an EIN index μ is either a constant ($\mu \in \mathbb{N}$) or a variable index $\mu \in \text{dom}(\sigma)$

We present a few typing rules next and refer the reader to Figure 4.1 and Figure 4.2 for a complete list of the rules. First consider the base case of a tensor variable T_α; the typing rule is

$$\frac{\Gamma, \sigma(T_\alpha) = \mathbf{Ten}[d_1, \ldots, d_n] \qquad |\alpha| = n \qquad \sigma \vdash \alpha < [d_1, \ldots d_n]}{\Gamma, \sigma \vdash T_\alpha : (\sigma)\mathcal{T}}$$

The antecedents of this rule state that T_α has a type that is compatible with both the multi-index α and the index map σ. A similar rule applies for field variables. The rule for

convolution yields an indexed field type.

$$\frac{\Gamma(V) = \mathbf{Img}(d)[d_1, \ldots, d_n] \qquad \Gamma(H) = \mathbf{Krn} \qquad |\alpha\beta| = n \qquad \sigma \vdash \alpha\beta < [d_1, \ldots d_n]}{\Gamma, \sigma \vdash V_\alpha \circledast H^\beta : (\sigma)\mathcal{F}^d}$$

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:

$$\frac{i \notin \mathrm{dom}(\sigma) \qquad \sigma' = \sigma[i \mapsto (1, n)] \qquad \Gamma, \sigma' \vdash e : (\sigma')\mathcal{T}}{\Gamma, \sigma \vdash \sum_{i=1}^{n} e : (\sigma)\mathcal{T}}$$

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. We use a similar rule for differentiation.

$$\frac{\sigma(i) = d \qquad \sigma' = \sigma \setminus i \qquad \Gamma, \sigma' \vdash e : (\sigma')\mathcal{F}^d}{\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} e : (\sigma)\mathcal{F}^d}$$

We can look up index i in σ with $\sigma(i) = d$ which indicates $1 \leq i \leq d$. The term $\sigma' = \sigma \setminus i$ indicates that the index map σ' has all the same index bindings as σ except i.

The term δ_{ij} does not change the context.

$$\frac{i, j \in \mathrm{dom}(\sigma)}{\Gamma, \sigma \vdash \delta_{ij} : (\sigma)\mathcal{T}} \qquad \frac{\Gamma, \sigma \vdash ok}{\Gamma, \sigma \vdash \delta.\delta. : (\sigma)\mathcal{T}}$$

The application of a Kronecker delta function δ_{ij} adds index j to the context and removes index i.

$$\frac{\sigma' = \sigma[j \mapsto (1, d)]/\ i \qquad \Gamma, \sigma' \vdash e : (\sigma')\tau_0}{\Gamma, \sigma \vdash (\delta_{ij} * e) : (\sigma)\tau_0}$$

Similarly, the $\mathcal{E}$ term by itself does not change the context.

$$\frac{\forall i \in \alpha. \quad i \in dom(\sigma)}{\Gamma, \sigma \vdash \mathcal{E}_\alpha : (\sigma)\mathcal{T}} \qquad \Gamma, \sigma \vdash \mathcal{E}_{ijk}\mathcal{E}_{ilm} : (\sigma)\mathcal{T}$$

When applying $\mathcal{E}$ to another term we preserve that term's type.

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

The Probe operation probes an expression and a tensor $\mathbf{Ten}[d]$.

$$\frac{\Gamma, \sigma \vdash \delta_{ij} : \tau \quad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash \delta_{ij}@x : \tau} \qquad \frac{\Gamma, \sigma \vdash \mathcal{E}_\alpha : \tau \quad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash \mathcal{E}_\alpha@x : \tau} \qquad \frac{\Gamma, \sigma \vdash e : (\sigma)\mathcal{F}^d \quad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash e@x : (\sigma)\mathcal{T}}$$

Consider lifting a tensor term to the field level:

$$\frac{\Gamma, \sigma \vdash e : (\sigma)\mathcal{T}}{\Gamma, \sigma \vdash \mathbf{lift}_d(e) : (\sigma)\mathcal{F}^d}$$

The sub-term e has a tensor type $(\sigma)\mathcal{T}$ but the lifted term $\mathbf{lift}_d(e)$ has 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 valued terms such as reals and scalar fields.

$$\frac{\Gamma, \sigma \vdash e : ()\tau_0 \;\; \odot_1 \in \{\sqrt{}, -, \kappa, \exp, (\cdot)^n\}}{\Gamma, \sigma \vdash \odot_1(e) : ()\tau_0}$$

$$\begin{array}{ll} \Gamma, \sigma \vdash T_\alpha : \tau & \mapsto \tau = (\sigma)\mathcal{T} \\ \Gamma, \sigma \vdash F_\alpha : \tau & \mapsto \tau = (\sigma)\mathcal{F}^d \\ \ldots & \end{array}$$

Figure 4.3: The inversion lemma makes inferences based on a structural type judgements. Given a conclusion (left), we can infer something about the type τ (right).

The subexpressions in an addition or subtraction expression have the same type as the result.

$$[\text{TYJUD}_{10}]\frac{\Gamma, \sigma \vdash e_1 : \tau \qquad \Gamma, \sigma \vdash e_2 : \tau \qquad \odot_2 \in \{+, -\}}{\Gamma, \sigma \vdash (e_1 \odot_2 e_2) : \tau}$$

The full set of typing judgements and corresponding inversion lemmas are contained in Figure 4.1, Figure 4.2, and Figure 4.3, respectively.

$$\frac{\sigma = i_1 : d_1, \ldots, i_m : d_m(\sigma, \{x_i \mapsto \theta_i \mid 1 \leq i \leq n\}) \vdash e : (\sigma)\mathcal{T}}{\vdash \lambda\,(x_1 : \theta_1, \ldots, x_n : \theta_1)\langle e \rangle_\sigma : (\theta_1 \times \cdots \times \theta_n) \to \mathbf{Ten}[d_1, \ldots, d_m]}$$

4.1.2 Type preservation Theorem

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

Theorem 4.1.1 (Type preservation)**.** *If* $\vdash \Gamma, \sigma$ **ok**, $\Gamma, \sigma \vdash e : \tau$, *and* $e \xrightarrow[rule]{} e'$, *then* $\Gamma, \sigma \vdash e' : \tau$

Given a derivation d of the form $e \xrightarrow[rule]{} e'$ we state T(d) as a shorthand for the claim that the derivation preserves the type of the expression e. For each rewrite rule $(e \xrightarrow[rule]{} e')$, the structure of the left-hand-side (LHS) 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 (RHS) of the rewrite. Provided below are key cases of the proof (Section A.1).

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

The left hand side of the rewrite rule is a tensor type because it is the result of a probe operation. The LHS has the following type.

$$\Gamma, \sigma \vdash (\sum_{i=1}^{n} e_1)@x : (\sigma)\mathcal{T}$$

We want to show that the RHS has the same type.

$$\Gamma, \sigma \vdash \sum_{i=1}^{n}(e_1@x) : (\sigma)\mathcal{T}.$$

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma[i \mapsto (1,n)] \vdash e_1 : (\sigma[i \mapsto (1,n)])\mathcal{F}^d[\text{TYINV}_3]}{\Gamma, \sigma \vdash (\sum_{i=1}^{n}(e_1)) : (\sigma)\mathcal{F}^d[\text{TYINV}_7]} \qquad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash (\sum_{i=1}^{n}(e_1))@x : (\sigma)\mathcal{T}}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : (\sigma[i \mapsto (1,n)])\mathcal{F}^d$

then $\Gamma, \sigma \vdash e_1@x : (\sigma[i \mapsto (1,n)])\mathcal{T}$ by $[\text{TYJUD}_7]$

and $\Gamma, \sigma \vdash \sum_{i=1}^{n}(e_1@x) : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_3]$

`T(R4) OK`

R6 The rewrite rule (R6) has the form $\frac{\partial}{\partial x_i} \diamond (e_1 * e_2) \xrightarrow[rule]{} e_1(\frac{\partial}{\partial x_i} \diamond e_2) + e_2(\frac{\partial}{\partial x_i} \diamond e_1)$.

The left hand side of the rewrite rule is a field type because it is the result of a field operation. The LHS has the following type.

$$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (e_1 * e_2) : (\sigma)\mathcal{F}^d$$

We want to show that the RHS has the same type.

$$\Gamma, \sigma \vdash e_1 \frac{\partial}{\partial x_i} \diamond e_2 + e_2 \frac{\partial}{\partial x_i} \diamond e_1 : (\sigma)\mathcal{F}^d.$$

The type derivation for the LHS is the following structure.

We use inversion to find the type for subexpressions e_1 and e_2.

$$\frac{\dfrac{\Gamma, \sigma \setminus i \vdash e_1 \qquad e_2 : (\sigma \setminus i)\mathcal{F}^d, [\text{TYINV}_{11}]}{(\Gamma, \sigma \setminus i \vdash e1 * e2 : (\sigma \setminus i)\mathcal{F}^d [\text{TYINV}_4]}}{\Gamma, \sigma \vdash \dfrac{\partial}{\partial x_{i:d}} \diamond (e1 * e2) : (\sigma)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1, e_2 : (\sigma \setminus i)\mathcal{F}^d$

then $\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (e_1), \frac{\partial}{\partial x_i} \diamond (e_2) : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_4]$,

$\Gamma, \sigma \vdash e_1 * \frac{\partial}{\partial x_i} \diamond (e_2), e_2 * \frac{\partial}{\partial x_i} \diamond (e_1) : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_{11}]$,

and $\Gamma, \sigma \vdash e_1 * \frac{\partial}{\partial x_i} \diamond (e_2) + e_2 * \frac{\partial}{\partial x_i} \diamond (e_1) : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_{10}]$.

`T( R6) OK`

R7 The rewrite rule (R7) has the form $\frac{\partial}{\partial x_i} \diamond (\frac{e_1}{e_2}) \xrightarrow[rule]{} \frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2}$. The left hand side of the rewrite rule is a field type because it is the result of a field operation. The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (\frac{e_1}{e_2}) : (\sigma)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2} : (\sigma)\mathcal{F}^d$.

The type derivation for the LHS is the following structure.

We use inversion to find the type for subexpressions e_1 and e_2.

$$\frac{\dfrac{\Gamma, \sigma \setminus i \vdash e_1 : (\sigma \setminus i)\mathcal{F}^d \qquad \Gamma, \sigma \vdash e_2 : ()\mathcal{F}^d, [\text{TYINV}_{12}]}{(\Gamma, \sigma \setminus i \vdash \dfrac{e1}{e2} : (\sigma \setminus i)\mathcal{F}^d [\text{TYINV}_4]}}{\Gamma, \sigma \vdash \dfrac{\partial}{\partial x_{i:d}} \diamond (\dfrac{e1}{e2}) : (\sigma)\mathcal{F}^d}$$

From that we can make the RHS derivations.

We use a type judgement to get the type of the subexpressions $(e_2 * e_2)$ in the right hand side of the rewrite rule.

Given that $\Gamma, \sigma \vdash e_2 : ()\mathcal{F}^d$ then $\Gamma, \sigma \vdash e_2 * e_2 : ()\mathcal{F}^d$ by $[\text{TYJUD}_{11}]$

We use a type judgement to get the type of the subexpressions $(\frac{\partial}{\partial x_{i:d}} \diamond e_2)$ in the right hand side of the rewrite rule.

Given that $\Gamma, \sigma \vdash e_2 : ()\mathcal{F}^d$ then $\Gamma, \sigma[i \mapsto (1,d)] \vdash \frac{\partial}{\partial x_{i:d}} \diamond e_2 : (i)\mathcal{F}^d$ by [TYJUD$_4$]

Next, we use a type judgement to get the type of the subexpressions $(e_1 * \frac{\partial}{\partial x_{i:d}} \diamond e_2)$ in the right hand side of the rewrite rule.

Given that $\Gamma, \sigma[i \mapsto (1,d)] \vdash \frac{\partial}{\partial x_{i:d}} \diamond e_2 : (i)\mathcal{F}^d$

and $\Gamma, \sigma \vdash e_1 : (\sigma \setminus i)\mathcal{F}^d$

then $\Gamma, \sigma \vdash e_1 \frac{\partial}{\partial x_{i:d}} \diamond e_2 : (\sigma)\mathcal{F}^d$ by [TYJUD$_{11}$]

The same is done to find $\Gamma, \sigma \vdash e_2 \frac{\partial}{\partial x_{i:d}} \diamond e_1 : (\sigma)\mathcal{F}^d$

Given that $\Gamma, \sigma \vdash ((\frac{\partial}{\partial x_i} \diamond e_1) * e_2), (e_1 * \frac{\partial}{\partial x_i} \diamond e_2) : (\sigma)\mathcal{F}^d$

and $\Gamma, \sigma \vdash e_2 * e_2 : ()\mathcal{F}^d$

then $\Gamma, \sigma \vdash ((\frac{\partial}{\partial x_i} \diamond e_1) * e_2) - (e_1 * \frac{\partial}{\partial x_i} \diamond e_2) : (\sigma)\mathcal{F}^d$ by [TYJUD$_{10}$]

and $\Gamma, \sigma \vdash \frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2} : (\sigma)\mathcal{F}^d$ by [TYJUD$_{12}$]

`T( R7) OK`

R10 The rewrite rule (R10) has the form $\frac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1)) \underset{rule}{\longrightarrow} (\mathbf{cosine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$. The left hand side of the rewrite rule is a field type because it is the result of a field operation. The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1)) : (i)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash (\mathbf{cosine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d$.

The type derivation for the LHS is the following structure.

We use inversion to find the type for subexpression e_1.

$$\dfrac{\dfrac{\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d [\text{TYINV}_9]}{\Gamma, \sigma \vdash \mathbf{sine}(e_1) : ()\mathcal{F}^d}}{\Gamma, \sigma[i \mapsto (1,d)] \vdash \dfrac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1)) : (i)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$

then $\Gamma, \sigma[i \mapsto (1,d)] \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (i)\mathcal{F}^d$ by [TYJUD$_4$],

$\Gamma, \sigma \vdash \mathbf{cosine}(e_1) : ()\mathcal{F}^d$ by [TYJUD$_9$],

and $\Gamma, \sigma[i \mapsto (1, d)] \vdash (\mathbf{cosine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d$ by [TYJUD$_{11}$].

`T( R10) OK`

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

We use inversion to find the type for subexpression e_1, e_2, e_3.

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\frac{e_1}{e_2}}{e_3} :(\sigma)\tau_0$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \frac{e_1}{e_2 e_3}:(\sigma)\tau_0$.

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma)\tau_0, \Gamma, \sigma \vdash e_2 : ()\tau_0[\text{TYINV}_{12}]}{\Gamma, \sigma \vdash \dfrac{e_1}{e_2} : (\sigma)\tau_0} \qquad e_3 : ()\tau_0[\text{TYINV}_{12}]}{\Gamma, \sigma \vdash \dfrac{\frac{e_1}{e_2}}{e_3} : (\sigma)\tau_0}$$

From that we can make the RHS derivations.

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

then $\Gamma, \sigma \vdash e_2 * e_3 : ()\mathcal{T}$ by [TYJUD$_{11}$],

and $\Gamma, \sigma \vdash \frac{e_1}{e_2 e_3} : (\sigma)\mathcal{T}$ by [TYJUD$_{12}$].

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

`T( R27) OK`

R40 The rewrite rule (R40) has the form $\delta_{ij}\frac{\partial}{\partial x_j} \diamond e_1 \xrightarrow[rule]{} \frac{\partial}{\partial x_i} \diamond (e_1)$.

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

We claim the type for the subexpression $(e_1).\Gamma, \sigma_2 \vdash e_1 : (\sigma_2)\mathcal{F}^d$

We use a type judgement to get the type of the subexpression $(\frac{\partial}{\partial x_j} \diamond e_1)$.

Given that $\Gamma, \sigma_2 \vdash e_1 : (\sigma_2)\mathcal{F}^d$ then $\Gamma, \sigma_j \vdash \frac{\partial}{\partial x_j} \diamond e_1 : (\sigma_j)\mathcal{F}^d$ by [TYJUD$_4$]

We switch the indices when applying the δ.

so that $\Gamma, \sigma_i \vdash \delta_{ij}(\frac{\partial}{\partial x_j} \diamond e_1) : (\sigma_i)\mathcal{F}^d$ by [TYJUD$_5$]

From that we can make the RHS derivations.

Given that $\Gamma, \sigma_2 \vdash e_1 : (\sigma_1)\mathcal{F}^d$ then $\Gamma, \sigma_i \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (\sigma_i)\mathcal{F}^d$ by [TYJUD$_4$]

`T( R40) OK`

R41 The rewrite rule (R41) has the form $\sum(se_1) \xrightarrow[rule]{} s \sum e_1$.

We use inversion to find the type for subexpression s and e.

The LHS has the following type.

$\Gamma, \sigma \vdash \sum(se_1) :(\sigma)\tau_0$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash s \sum e_1 :(\sigma)\tau_0$.

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma' \vdash s : ()\tau_0, [\text{TYINV}_{11}] \qquad \Gamma, \sigma' \vdash e_1 : (\sigma')\tau_0}{\sigma' = \sigma[i \mapsto (1, n)] \qquad \Gamma, \sigma' \vdash s * e_1 : (\sigma')\tau_0 [\text{TYINV}_3]}}{\Gamma, \sigma \vdash (\sum_{i=1}^{n}(s * e)) : (\sigma)\tau_0}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : (\sigma[i \mapsto (1, n)])\tau_0$ and $\Gamma, \sigma \vdash s : ()\tau_0$

then $\Gamma, \sigma \vdash \sum_{i=1}^{n}(e_1) : (\sigma)\tau_0$ by [TYJUD$_3$]

and $\Gamma, \sigma \vdash s * \sum_{i=1}^{n}(e_1) : (\sigma)\tau_0$ by [TYJUD$_{11}$].

`T( R41) OK`

4.2 Value Preservation

4.2.1 Value Definition

To show that the rewriting system preserves the semantics of the program, we must give a dynamic semantics to EIN expressions. We assume a set of values ($v \in \textsc{Value}$) that include reals, permutation tensor, Kronecker delta functions, and tensors. Rather than define the meaning of an expression to be a function from indices to values, we include a mapping ρ from index variables to indices as part of the dynamic environment. We define a dynamic environment to be $\Psi, \rho \in (\textsc{IndexVar} \xrightarrow{\text{fin}} \mathbb{Z}) \times (\textsc{Var} \xrightarrow{\text{fin}} \textsc{Value})$, where Value is the domain of computational values (*e.g.*, tensors, *etc.*). We define the meaning of an EIN expression (for a subset of EIN expressions) using a big-step semantics $\Psi, \rho \vdash e \Downarrow v$, where v is a value. We describe values next and present evaluation rules Figure 4.5.

$\mathbf{v}$	::=	$Real(n)$	$n \in \mathcal{R}$
	|	$Tensor[p \cdot b_1 \dots b_n]$	index tensor argument p using basis values b
	|	E_α	Reduces Levi-Civita tensor
	|	K_{ij}	Reduces Kronecker delta function

Figure 4.4: Value definitions ($\mathbf{v}$) for a subset of EIN expression

We assume an orthonormal basis function. Inspired by Equation 1.6, we use b_i to represent a basis vector inside a value expression. The value of a vector is defined as

$$\Psi, \rho \vdash T_i \Downarrow Tensor[T \cdot b_i]$$

A term b_i is created for each variable index i in the EIN expressions. The full tensor judgement

$$\Psi, \rho \vdash T_\alpha \Downarrow Tensor[T \cdot b_{\alpha 1} \dots b_{\alpha n}]$$

is used to represent an arbitrary sized tensor. The lift operation is used to lift a tensor to a

field. The value of a lifted term is the value of that term.

$$\frac{\Psi, \rho \vdash e \Downarrow v}{\Psi, \rho \vdash \mathbf{lift}_d(e) \Downarrow v}$$

We support arithmetic operations on and between u. The summation expression can be evaluated with the following judgement:

$$\frac{\Psi, \rho \vdash e \Downarrow v}{\Psi, \rho \vdash \sum_{i=1}^{n} e \Downarrow \Sigma_{i=1}^{n} v}$$

The summation operator is applied to the u. Generally, the judgement for unary operators ($\odot_1 \in \{\Sigma \mid \surd \mid - \mid \kappa \mid \exp \mid (\cdot)^n\}$) is as follows:

$$\frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Real}(r1)}{\Psi, \rho \vdash \odot_1 e_1 \Downarrow \mathit{Real}(\odot_1 r1)}$$

$$\frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Tensor}[e_1 \cdot b1]}{\Psi, \rho \vdash \odot_1 e_1 \Downarrow \odot_1(\mathit{Tensor}[e_1 \cdot b1])}$$

The binary operators ($\odot_2 = + \mid - \mid * \mid /$) can be applied between u. .

$$\frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Real}(r_1) \;\; \Psi, \rho \vdash e_2 \Downarrow \mathit{Real}(r_2)}{\Psi, \rho \vdash (e_1 \odot_2 e_2) \Downarrow \mathit{Real}(r_1 \odot_2 r_2)}$$

$$\frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Tensor}[e_1 \cdot b1] \qquad \Psi, \rho \vdash e_2 \Downarrow \mathit{Tensor}[e_2 \cdot b2]}{\Psi, \rho \vdash (e_1 \odot_2 e_2) \Downarrow \mathit{Tensor}[e_1 \cdot b1] \odot_2 \mathit{Tensor}[e_2 \cdot b2]}$$

The epsilon and Kronecker delta functions are each reduced to a distinct permutation value (E_α or K_{ij}).

$$\Psi, \rho \vdash \mathcal{E}_{ijk} \Downarrow E_{ijk} \quad \Psi, \rho \vdash \delta_{ij} \Downarrow K_{ij}$$

The value for $\mathcal{E}_{ijk}$ is subject to Equation 1.1. The value for δ_{ij} is subject to Equation 1.2, Equation 1.3, and Equation 1.4.

We use notation $v_1 \mapsto v_2$ to indicate a value that is reduced or rewritten. We combine permutation values with tensor values as

$$K_{ij} * \mathit{Tensor}[T \cdot \beta] \mapsto \mathit{Tensor}[T \cdot b_i \cdot b_j \cdot \beta]. \tag{4.1}$$

The full set of evaluation rules are given in Figure 4.5.

4.2.2 Value Preservation Theorem

Our correctness theorem states the 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 and their value is defined.

Theorem 4.2.1 (Value Preservation)**.** *If* $\vdash \Gamma, \sigma$ ***ok****,* $\Gamma, \sigma \vdash e : \tau$*,* $\Gamma, \sigma \vdash \Psi, \rho$ ***ok****,* $e \xrightarrow[rule]{} e'$*, and* $\Psi, \rho \vdash e \Downarrow v$*, then* $\Psi, \rho \vdash e' \Downarrow v$

Assume $\Psi, \rho \vdash e \Downarrow v$ and $e \xrightarrow[rule]{} e'$, then the proof proceeds by case analysis of the rewrite rules. Does not include rules that involve fields terms (values for fields are not defined). We show the full proof in Section A.2 and select a few key examples below.

R24 The rewrite rule (R24) has the form $e_1 - 0 \xrightarrow[rule]{} e_1$.

Claim $e_1 - 0$ evaluates to v.

We need to define v.

Assume that $e_1 \Downarrow v'$

then $\Psi, \rho \vdash e_1 - 0 \Downarrow v' - \mathit{Real}(0)$ by $[\text{VALJUD}_1], [\text{VALJUD}_5]$.

The value of v is $v' - \mathit{Real}(0)$.

$$[\text{VALJUD}_1] \quad \Psi, \rho \vdash c \Downarrow \mathit{Real}(c)$$

$$[\text{VALJUD}_2] \quad \Psi, \rho \vdash T_\alpha \Downarrow \mathit{Tensor}[T \cdot b_{\alpha 1} \ldots b_{\alpha n}]$$

$$[\text{VALJUD}_3] \quad \frac{\Psi, \rho \vdash e \Downarrow v}{\Psi, \rho \vdash \mathbf{lift}_d(e) \Downarrow v}$$

$$[\text{VALJUD}_4] \quad \odot_1 \in \{\textstyle\sum \mid \sqrt{} \mid - \mid \kappa \mid \exp \mid (\cdot)^n\}$$
$$\frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Real}(r1)}{\Psi, \rho \vdash \odot_1 e_1 \Downarrow \mathit{Real}(\odot_1 r1)} \qquad \frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Tensor}[e_1 \cdot b1]}{\Psi, \rho \vdash \odot_1 e_1 \Downarrow \odot_1 \mathit{Tensor}[e_1 \cdot b1]}$$

$$[\text{VALJUD}_5] \quad \odot_2 = + \mid - \mid * \mid /$$
$$\frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Real}(r1) \qquad \Psi, \rho \vdash e_2 \Downarrow \mathit{Real}(r2)}{\Psi, \rho \vdash (e_1 \odot_2 e_2) \Downarrow \mathit{Real}(r1 \odot_2 r2)}$$
$$\frac{\Psi, \rho \vdash e_1 \Downarrow \mathit{Tensor}[e_1 \cdot b1] \qquad \Psi, \rho \vdash e_2 \Downarrow \mathit{Tensor}[e_2 \cdot b2]}{\Psi, \rho \vdash (e_1 \odot_2 e_2) \Downarrow \mathit{Tensor}[e_1 \cdot b1] \odot_2 \mathit{Tensor}[e_2 \cdot b2]}$$

$$\frac{\Psi, \rho \vdash e_1 \Downarrow v_1 \;\; \Psi, \rho \vdash e_2 \Downarrow v_2 \;\; \odot_2 = + \mid - \mid * \mid /}{\Psi, \rho \vdash (e_1 \odot_2 e_2)@x \Downarrow \mathit{Probe}(v_1)[x] \odot_2 \mathit{Probe}(v_2)[x]}$$

$$[\text{VALJUD}_6] \quad \frac{\Psi, \rho \vdash e \Downarrow v}{\Psi, \rho \vdash \mathbf{lift}_d(e)@e \Downarrow v} \qquad \frac{\Psi, \rho \vdash \delta_{ij} \Downarrow v}{\Psi, \rho \vdash \delta_{ij}@e \Downarrow v} \qquad \frac{\Psi, \rho \vdash \mathcal{E}_\alpha \Downarrow v}{\Psi, \rho \vdash \mathcal{E}_\alpha@e \Downarrow v}$$

$$[\text{VALJUD}_7] \quad \Psi, \rho \vdash \delta_{ij} \Downarrow K_{ij} \qquad \Psi, \rho \vdash \mathcal{E}_\alpha \Downarrow E_\alpha$$

Figure 4.5: Value Judgements for each EIN expression.

By using algebraic reasoning: $v' - \mathit{Real}(0) = v'$.

Since $e_1 - 0 \Downarrow v$ and $e_1 - 0 \Downarrow v'$ then $v = v'$

The last step leads to $e_1 \Downarrow v$

`V( R24) OK`

R32 The rewrite rule (R32) has the form $\sqrt{(e_1)} * \sqrt{(e_1)} \underset{rule}{\longrightarrow} e_1$.

Claim $\sqrt{(e_1)} * \sqrt{(e_1)}$ evaluates to v.

We need to define v.

Assume that $e_1 \Downarrow v'$

then $\Psi, \rho \vdash \sqrt{e_1} \Downarrow \sqrt{(v')}$ by [VALJUD$_4$],

and $\Psi, \rho \vdash \sqrt{e_1}\sqrt{e_1} \Downarrow \sqrt{v'}\sqrt{v'}$ by [VALJUD$_5$]

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

By using algebraic reasoning to analyze v

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

The last step leads to $e_1 \Downarrow v$

`V( R32) OK`

R35 The rewrite rule (R35) has the form $\mathcal{E}_{ijk}\mathcal{E}_{ilm} \underset{rule}{\longrightarrow} \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$.

Claim $\mathcal{E}_{ijk}\mathcal{E}_{ilm}$ evaluates to v.

We need to define v.

Given that $\mathcal{E}_{ijk} \Downarrow E_{ijk}$ and $\mathcal{E}_{pqr} \Downarrow E_{pqr}$ then $\mathcal{E}_{ijk}\mathcal{E}_{pqr} \Downarrow E_{ijk}E_{pqr}$.

The value of v is $E_{ijk}E_{pqr}$.

Consider the product of two E expressions as

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

$$\longrightarrow K_{ip}(K_{jq}K_{kr} - K_{jr}K_{kq}) + K_{iq}(K_{jr}K_{kp} - K_{jp}K_{kr}) + K_{ir}(K_{jp}K_{kq} - K_{jq}K_{kp})$$

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

$$\longrightarrow K_{ii}K_{jq}K_{kr} - K_{ii}K_{jr}K_{kq} + K_{iq}K_{jr}K_{ki} - K_{iq}K_{ji}K_{kr} + K_{ir}K_{ji}K_{kq} - K_{ir}K_{jq}K_{ki}$$

Applying Equation 1.4:

$\longrightarrow 3K_{jq}K_{kr} - 3K_{jr}K_{kq} + K_{iq}K_{jr}K_{ki} - K_{iq}K_{ji}K_{kr} + K_{ir}K_{ji}K_{kq} - K_{ir}K_{jq}K_{ki}$

Applying Equation 1.3:

$\longrightarrow 3K_{jq}K_{kr} - 3K_{jr}K_{kq} + K_{kq}K_{jr} - K_{jq}K_{kr} + K_{jr}K_{kq} - K_{kr}K_{jq}$

Reduces to:

$\longrightarrow K_{jq}K_{kr} - K_{jr}K_{kq}$

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

$\longrightarrow K_{jl}K_{km} - K_{jm}K_{kl}$

We need to show that $\delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$ evaluates to v.

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 [VALJUD$_7$]

then $\Psi, \rho \vdash \delta_{jl}\delta_{km} \Downarrow K_{jl}K_{km}$ $\delta_{jm}\delta_{kl} \Downarrow K_{jm}K_{kl}$ by [VALJUD$_5$]

and $\Psi, \rho \vdash \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} \Downarrow K_{jl}K_{km} - K_{jm}K_{kl}$ by [VALJUD$_5$]

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

`V( R35) OK`

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

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

We need to define v.

Given that $\Psi, \rho \vdash T_j \Downarrow \mathit{Tensor}[T \cdot b_j]$ by [VALJUD$_2$]

and $\Psi, \rho \vdash \delta_{ij} \Downarrow K_{ij}$ by [VALJUD$_7$]

then $\Psi, \rho \vdash \delta_{ij}T_j \Downarrow \mathit{Tensor}[T \cdot b_j \cdot b_i \cdot b_j]$ by *Equation* 4.1

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

By using algebraic reasoning to analyze v

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

We need to show that T_i evaluates to v.

Lastly, $\Psi, \rho \vdash T_i \Downarrow \mathit{Tensor}[T \cdot b_i]$ by [VALJUD$_2$]

The last step leads to $T_i \Downarrow v$

`V( R36) OK`

4.3 Termination

In this section we make the following claims:

1. Rewriting terminates
2. if e $\xrightarrow[rule]{}{}^*$ e' and $\nexists$ e" such that e' $\xrightarrow[rule]{}$ e", then e' $\in \mathcal{N}$

We prove that the normalization rewriting will terminate and that the resulting term will be in normal form.

Our approach uses the standard technique of defining a well-founded size metric $[\![e]\!]$ to show that the rewrite rules always decrease the size of an expression. The size metric guarantees that the normalization process terminates (Section 4.3.1). 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 4.3.2. We then define the *terminal* expressions as $\mathcal{T} = \{e \mid \nexists e' \text{ such that } e \xrightarrow[rule]{} e'\}$. The last section (Section 4.3.3) relates normal form expressions and terminal expressions.

Table 4.3: We define a size metric $[\![\bullet]\!] : e \to \mathbb{N}$ inductively on the structure of the grammar in Figure 2.1.

EIN expression (e)	Size metric $[\![[\![e]\!]$
c, T_α, F_α, $(v_\beta \circledast h^\mu)$, δ_{ij}	1
$\mathcal{E}_\alpha$	4
$\mathbf{lift}_d(e)$, $\sqrt{e}$, $-e$, $\exp(e)$, e^n, $\kappa(e)$	$1 + [\![e]\!]$
$e_1 + e_2$,$e_1 - e_2$, $e_1 * e_2$	$1 + [\![e_1]\!] + [\![e_2]\!]$
$\frac{a}{b}$	$2 + [\![e_1]\!] + [\![e_2]\!]$
$\sum e$	$2 + 2[\![e]\!]$
$\frac{\partial}{\partial x_\nu} \diamond \diamond e$	$5^{[\![e]\!]}[\![e]\!]$
$e(x)$	$2[\![e]\!]$

4.3.1 Size Metric

We define a size metric $[\![e]\!]$ for EIN expressions in Table 4.3 and use it to show that rewrites always decrease the size of the EIN expression.

Lemma 4.3.1. *If* $e \xrightarrow[rule]{} e'$ *then* $[\![e]\!] > [\![e']\!]$

Our proof does a case analysis on the rewrite rules ($e \xrightarrow[rule]{} e'$) and compares the size (Table 4.3) of each side of the rule. Provided below are key cases of the proof (Section A.3.1).

R1 The rewrite rule (R1) has the form $(e_1 \odot_n e_2)@x \xrightarrow[rule]{} (e_1@x) \odot_n (e_2@x)$.

case analysis on the operator $\odot_n$

if $\odot_n = *$

$$\begin{aligned} [\![(e_1 * e_2)@x]\!] &= 2 + 2[\![e_1]\!] + 2[\![e_2]\!] \\ &> 1 + 2[\![e_1]\!] + 2\mathcal{S} \\ &= [\![[\![(e_1@x) * (e_2@x)]\!]]\!] \end{aligned}$$

if $\odot_n = \frac{\bullet}{\bullet}$

$$\begin{aligned} [\![[\![(\tfrac{e_1}{e_2})@x]\!]]\!] &= 4 + 2[\![e_1]\!] + 2[\![e_2]\!] \\ &> 2 + 2[\![e_1]\!] + 2[\![e_2]\!] \\ &= [\![[\![\tfrac{e_1@x}{e_2@x}]\!]]\!] \end{aligned}$$

P(d)

R9 The rewrite rule (R9) has the form $\frac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1)) \xrightarrow[rule]{} (-\mathbf{sine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$.

$$\begin{aligned} [\![\tfrac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1))]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\ &> [\![e_1]\!] * (1 + 5^{[\![e_1]\!]}) + 3 \\ &= [\![(-\mathbf{sine}(e_1)) * (\tfrac{\partial}{\partial x_i} \diamond e_1)]\!] \end{aligned}$$

P(d)

R17 The rewrite rule (R17) has the form $\frac{\partial}{\partial x_i} \diamond (e_1 \odot e_2) \xrightarrow[rule]{} (\frac{\partial}{\partial x_i} \diamond e_1) \odot (\frac{\partial}{\partial x_i} \diamond e_2)$.

$$\begin{aligned} [\![\tfrac{\partial}{\partial x_i} \diamond (e_1 \odot e_2)]\!] &= (1 + [\![e_1]\!] + [\![e_2]\!])5^{(1+[\![e_1]\!]+[\![e_2]\!])} \\ &> [\![e_1]\!]5^{([\![e_1]\!])} + [\![e_2]\!]5^{([\![e_2]\!])} + 1 \\ &= [\![(\tfrac{\partial}{\partial x_i} \diamond e_1) \odot (\tfrac{\partial}{\partial x_i} \diamond e_2)]\!] \end{aligned}$$

P(d)

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

$$
\begin{array}{rcl}
[\![\frac{e_1}{\frac{e_2}{e_3}}]\!] & = & 4 + [\![e_1]\!] + [\![e_2]\!] + [\![e_3]\!] \\
& > & 3 + [\![e_1]\!] + [\![e_2]\!] + [\![e_3]\!] \\
& = & [\![\frac{e_1}{e_2 e_3}]\!]
\end{array}
$$

P(d)

4.3.2 Normal Form

An EIN expression is in normal form if it can not be reduced. The normal form is defined as the subset $\mathcal{N}$ of EIN expressions. In the following, we describe the normal form with the following examples. Some tensors, constants, and permutation terms that are in normal form include:

$$T_\alpha, c \neq 0, \delta_{ij}, \mathcal{E}_{ij}, \text{ and } \mathcal{E}_{ijk}$$

The field forms $\mathcal{F}$ include:

$$F_\alpha, V \circledast H, \frac{\partial}{\partial x_i} \diamond F_\alpha$$

All differentiation is applied (via product rule or otherwise) so in normal form the differentiation is only applied to a field term:

$$\frac{\partial}{\partial x_i} \diamond F_\alpha$$

until it is pushed down to the convolution kernel:

$$V \circledast \frac{\partial}{\partial x_i} \diamond H$$

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

$$F_\alpha @ T, (V \circledast H) @ x, \text{ and } (\frac{\partial}{\partial x_i} \diamond F) @ x$$

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

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

Other arithmetic operations cannot have a zero constant sub-term (3.7):

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

The division structure is subject to algebraic rewrites (3.7). 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*:

$$\begin{aligned}
\mathcal{N} &::= \mathcal{A} \mid c \\
\mathcal{A} &::= \mathcal{D} \mid \mathcal{G} \\
\mathcal{D} &::= \mathcal{B} \mid -\mathcal{G} \\
\mathcal{G} &::= \mathcal{B} \mid \frac{\mathcal{D}}{\mathcal{D}} \\
\mathcal{B} &::= T_\alpha \mid \mathcal{F} \mid \mathcal{F}@T_\alpha \mid c \neq 0 \mid \delta_{ij} \mid \mathcal{E}_{ij} \mid \mathcal{E}_{ijk} \\
&\quad \mid \mathcal{A} + \mathcal{A} \mid \mathcal{A} - \mathcal{A} \mid \sqrt{\mathcal{N}} \\
&\quad \mid \mathbf{lift}_d(\mathcal{N}) \mid \exp(\mathcal{N}) \mid \mathcal{N}^c \mid \kappa(\mathcal{N}) \\
&\quad \mid (\mathcal{A} * \mathcal{A})^{1,2,3,4} \\
&\quad \mid (\textstyle\sum \mathcal{N})^5 \\
\mathcal{F} &::= F_\alpha \mid v \circledast h \mid \frac{\partial}{\partial x_i} \diamond F_\alpha
\end{aligned}$$

subject to the following additional restrictions (noted in the syntax with an upper index):

1. If a term has the form $\mathcal{E}_{ijk} * \mathcal{E}_{i'j'k'}$ then the indices ijk must be disjoint from $i'j'k'$.

2. If a term contains the form $\mathcal{E}_{ijk} * \mathcal{A}$ and $\mathcal{A}$ has a differentiation component then no two

of the indices i, j, and k may occur in the differentiation component of $\mathcal{A}$. For example, $\mathcal{E}_{ijk} * \frac{\partial}{\partial x_{jk}} \diamond e$ is not in normal form and can be rewritten as $\mathcal{E}_{ijk} * \frac{\partial}{\partial x_{jk}} \diamond e \xrightarrow[rule]{}{}^{*} 0$.

3. If a term has the form $\delta_{ij} * \mathcal{A}$ then j may not occur in $\mathcal{A}$. For example, the expression $\delta_{ij} * T_j$ is not in normal form, and thus $\delta_{ij} * T_j$ can be rewritten to T_i.

4. If a term has the form $\sqrt{e_1} * \sqrt{e_2}$ then $e_1 \neq e_2$.

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

4.3.3 Termination and Normal form

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 $e \in \mathcal{N}$*

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 define $Q(e_x) \equiv \nexists e'_x$ such that $e_x \xrightarrow[rule]{} e'_x$ and $e_x \in \mathcal{N}$. The following is a sample of a proof by contradiction (full proof is available Section A.3.2).

case on structure e_x

If $e_x = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = V_\alpha \circledast H$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \delta_{ij}$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \mathcal{E}_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \mathbf{lift}_d(e_1)$

Prove Q(e) by contradiction.

If $e_1 = c$	then $Q(e_x)$ because e_x is in normal form.
If $e_1 = T_\alpha$	then $Q(e_x)$ because e_x is in normal form.
If $e_1 = F_\alpha$	then $Q(e_x)$ because e_x is not a supported type.
If $e_1 = e \circledast e$	then $Q(e_x)$ because e_x is not a supported type.
If $e_1 = \delta_{ij}$	then $Q(e_x)$ because e_x is in normal form.
If $e_1 = \mathcal{E}_\alpha$	then $Q(e_x)$ because e_x is in normal form.
If $e_1 = \mathbf{lift}_d(e)$	then $Q(e_x)$ because e_x is not a supported type.
If $e_1 = M(e_1)$	and assuming $Q(e)$ then $Q(e_x)$
	Given $M(e) = \sqrt{e} \mid exp(e) \mid e_1^n \mid \kappa(e)$
If $e_1 = -e$	and assuming $Q(e)$ then $Q(e_x)$
If $e_1 = \frac{\partial}{\partial x_\alpha} \diamond e$	then $Q(e_x)$ because e_x is not a supported type.
If $e_1 = \sum e$	and assuming $Q(e)$ then $Q(e_x)$
If $e_1 = e_3 + e_4$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e_1 = e_3 - e_4$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e_1 = e_3 * e_4$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e = \frac{e_3}{e_4}$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e_1 = e_3 @ e_4$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
$Q(e_x)$	

The next lemma demonstrates that normal form implies termination.

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

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 $e \xrightarrow[rule]{} e'$, 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).

Case R1.$(e_1 \odot_n e_2)@x \xrightarrow[rule]{} (e_1@x) \odot_n (e_2@x)$

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

Case R2.$(e_0 \odot_2 e_1)@x \xrightarrow[rule]{} (e_0@x) \odot_2 (e_1@x)$

Let y= $(e_0 \odot_2 e_1)@x$ 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 $e \xrightarrow[rule]{}{}^* e'$, 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'.

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 $\mathcal{E}_{ijk}$ 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

In Chapter 4 we provided proofs to state the formal properties of the rewriting system. We showed that the rewriting system is type preserving and terminating. The compiler is much more than just the rewriting system and includes many places where bugs can go undetected. To test the full implementation we develop a more robust approach to testing.

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 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 is 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 [52, 23], 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.

There is extensive previous research in compiler testing. Differential testing relies on comparing different implementations of the same language [56, 74]. There are other applications of testing. Equivalence Modulo Inputs [50] creates a family of programs that are expected to have the same output.

Alone, these approaches do not seem sufficient in the case of Diderot. Most of the transformations that occur during compilation of a Diderot program are necessary and can not be disabled. Additionally, earlier versions of the compiler implemented a smaller language. The current version of the compiler has introduced new operators and more expressivity to the language. We want to create test cases that use a mix of the old and new operators. Lastly, we want to evaluate each tests based on a ground truth.

In this chapter, we present Diderot's automated testing model, *DATm* [17]. It is designed to rigorously test the core mathematical parts of the Diderot implementation. *DATm* com-

bines the generation of test programs with the generation of synthetic data for which the correct values and properties of the generated Diderot program output are known. For each test program, synthetic data is used to synthesize tensors and tensor fields that are then used by the test program. The correct solution can then be derived analytically as an operation on polynomials. The generated test program is compiled and run on the test data, and its output is compared with the analytically derived solution. The test passes if the answers are within an error tolerance.

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 is 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 typechecker.

The remainder of the chapter is organized as follows. We first describe the basic structures and details core to the testing model in Section 5.1. We introduce *DATm* and describe its implementation in Section 5.2. In Section 5.3, we describe how the model can be extended to automatically test a class of visualization algorithms. In Section 5.4 we report the bugs that we found. Section 5.5 presents the results about *DATm*'s efficiency in generating and running tests. Lastly, we discuss the contributions of the work in Section 5.6.

5.1 Core of testing model

In this chapter we introduce the techniques we use to test the Diderot compiler by introducing two models: *DATm* (Section 5.2) and *DAVm* (Section 5.3). The models differ in how a single test is executed but share an implementation core. In this section we introduce that

τ	::=	**tensor**[ς]	tensor with shape ς
		field#$k(d)[\kappa]$	tensor field with continuity k, shape κ, and dimension d ($1 \leq d \leq 3$)
		image$(d)[\kappa]$	image data with shape κ, and dimension d ($1 \leq d \leq 3$)
γ	::=	τ | **field**#$k(d)[\varsigma]$	broader range of shape
d_n	::=	2 | 3	dimension
v_n	::=	2 | 3 | 4	extended dimension
ς	::=	nil' | $v_1, \cdots, v_n$	$n \leq 3$ Tensor shape
κ	::=	nil' | v_1 | d_1, d_1	Field shape

operators	::=	$-$, $\|\cdot\|$, @, $\sqrt{}$, ∇, $\nabla\otimes$, $\nabla\times$, $\nabla\bullet$, inverse normalize, trace, transpose
	|	$+$, $-$, $*$, $/$, $\bullet$, $:$, $\times$, $\otimes$, [:,0], [1,:], [0], [1], [:,1,:], [1,0,:],
	|	det, sin, cos, tan, arccos, arcsin, arctan, $\circledast$, modulate, ...
kernel	::=	c4hexic, tent, ctmr, bspln3,

Figure 5.1: Subset of Diderot types and operators that can be tested with *DATm*

core, which includes the internal representation of our system, out testing input, and a test generator. In Section 5.1.1 we present the basic structures that internally represent the test cases. In Section 5.1.2 we present the testing frame and define some key concepts. Lastly, we describe the test case generator in Section 5.1.2.

5.1.1 Basic structures

The basic structures are illustrated in Figure 5.2 with their attributes. The objects (operators, types, and kernels) are used to represent a part of the Diderot language (Figure 5.1). The arguments represent a tensor or tensor field initiated with synthetic data and a type. An application object applies an operator object to arguments. Lastly, the testing frame is more detailed and described in Section 5.1.2.

Operators Operations include a mix of unary and binary operators applied to tensors and tensor fields (Figure 5.2). The operator object has a few attributes: *out*, *placement*, *arity*, and *limit*. The *arity* is the arity to the operator. The *limit* refers to the restriction

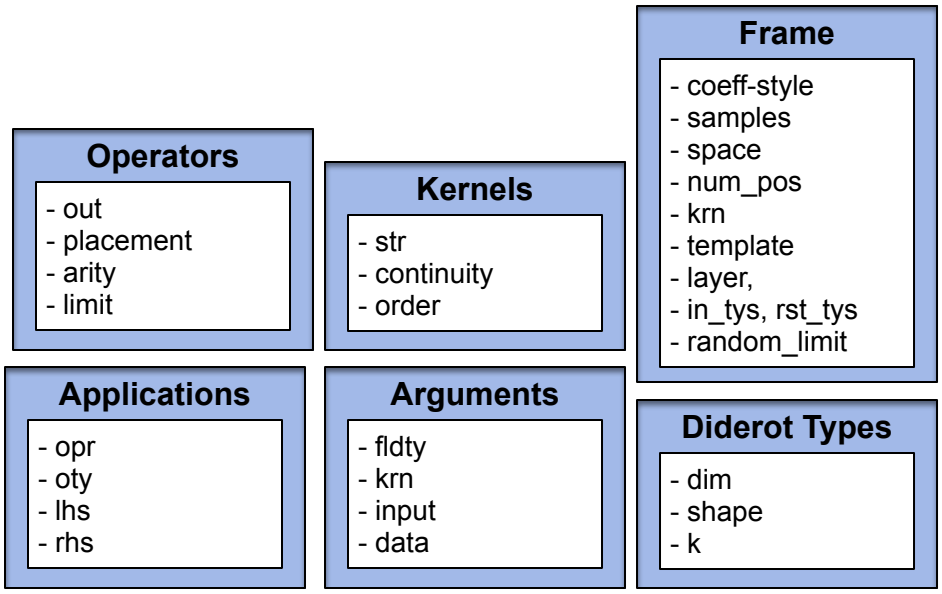

Figure 5.2: Core data structures used in testing

to the value of arguments. The attributes (*out*, *placement*) describe how the operators are translated into Diderot code.

For example, the unary negation operator is defined as

$$\text{op}_{negation} = \{out : -, placement : \text{left}, arity : 1, limit : \text{None}\}$$

When generating Diderot code the unicode symbol $-$ is placed in the left of one argument. The unary square root operator is defined as

$$\text{op}_{sqrt} = \{out : \text{sqrt}, placement : \text{split}, arity : 1, limit : \text{positive}\} \tag{5.1}$$

When generating Diderot code the function $\text{sqrt}(\cdot)$ is placed around a single argument. The limitation attribute refers to the value restriction, which needs to be positive.

A binary operator, such as the outer product, is defined as

$$\text{op}_{outer} = \{out : \otimes, placement : \text{middle}, arity : 2, limit : \text{None}\} \tag{5.2}$$

The unicode symbol $\otimes$ is placed in the middle of two arguments and there are no limitations. A field operator, such as the gradient operator, is defined as

$$\text{op}_{grad} = \{out : \nabla, placement : \text{left}, arity : 1, limit : \text{None}\} \tag{5.3}$$

Note that the gradient needs to be applied to a field where the continuity is greater than zero but the limit attribute is "None". Type restrictions are upheld in the internal type checker, which is introduced in Section 5.1.3.

Types The type object has a few attributes: *shape*, *dim*, and *k*. A tensor type only uses attribute shape. A vector of length 3 is defined as

$$\text{ty}_{tv3} = \{shape : [3], dim : \text{Null}, k : \text{Null}\}$$

The attributes correspond to Diderot field types. This includes the tensor shape (*shape*), dimension (*dim*), and continuity (k). The 3-d scalar field is defined as

$$\text{ty}_{fsd3} = \{shape : [], dim : 3, k : \text{Null}\}$$

the 2-d vector of length 3 is defined as

$$\text{ty}_{fv3d2} = \{shape : [3], dim : 2, k : \text{Null}\}$$

and the 3-d 3-by-3 field is defined as

$$\text{ty}_{fm33d3} = \{shape : [3, 3], dim : 3, k : \text{Null}\}$$

We have a representation for each Diderot types (γ shown in Figure 5.1). Types τ includes reals, vectors, matrices, third-order tensors, scalar fields, vector fields, and n-by-n second

order tensors fields. Type γ extends it to include a broader range of shapes.

Kernel The kernel object has a few attributes: *str*, *continuity*, and *order*. The attributes include the name of the kernel as it is represented in Diderot (*str*), the continuity (*continuity*), and the kernel can accurately represent data with order (*order*). For example, we can define the tent kernel as

$$\text{krn}_{tent} = \{str : \text{“tent”}, continuity : 0, order : \text{linear}\}$$

and the c4hexic kernel as

$$\text{krn}_{hex} = \{str : \text{“c4hexic”}, continuity : 3, order : \text{cubic}\}$$

Arguments Arguments have several attributes: *fldty*, *data*, *krn*, and *input*. This refers to the argument type (*fldty*) and data coefficients (*data*). A tensor argument can be represented as the following:

$$\text{ten}_1 = \{fldty : \text{ty}_{tv3}, data : \ldots, krn : \text{Null}, input : \text{Null}\} \tag{5.4}$$

A tensor field argument has a kernel (*krn*) and data file name (*input*). A 2-d scalar field defined by a tent kernel and data file “f1” is represented as

$$\text{fld}_1 = \{fldty : \text{ty}_{fsd2}, krn : \text{krn}_{tent}, input : \text{“f1”}, data : \ldots\}$$

and a 3-d scalar field defined by a c4hexic kernel and data file “f0” is represented as

$$\text{fld}_2 = \{fldty : \text{ty}_{fsd3}, krn : \text{krn}_{kex}, input : \text{“f0”}, data : \ldots\} \tag{5.5}$$

The kernel defined in the field argument (Equation 5.5) is used to initialize the continuity attribute in the field type.

$$\mathrm{ty}_{fsd3'} = \{dim : 3, shape : [], k : \mathrm{krn}_{kex}.continuity\}$$

Application The application object has attributes: *opr*, *oty*, *lhs*, and *rhs*. The object represents the application of an operator (*opr*) to arguments (*lhs*, *rhs*) and the result has the type (*oty*). The addition of two vectors of length 3 results in a vector of length 3.

$$\mathrm{app}_0 = \{opr : \mathrm{opr}_{add}, oty : \mathrm{ty}_{tv3}, lhs : \mathrm{ten}_1, rhs : \mathrm{ten}_1\} \tag{5.6}$$

The application of the gradient (Equation 5.3) of a 3-d scalar field (Equation 5.5) would result in a 3-d vector field and be represented as follows:

$$\mathrm{app}_1 = \{opr : \mathrm{opr}_{grad}, oty : \mathrm{ty}_{fv3d3}, lhs : \mathrm{fld}_2, rhs : \mathrm{None}\} \tag{5.7}$$

We can represent nested operations such as the outer product (Equation 5.2) between the result of app_1 (Equation 5.7) and a tensor (Equation 5.4).

$$\mathrm{app}_2 = \{opr : \mathrm{opr}_{outer}, oty : \mathrm{ty}_{fm33d3}, lhs : \mathrm{app}_1, rhs : \mathrm{ten}_1\} \tag{5.8}$$

5.1.2 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. We introduce the testing frame, define exhaustive and random search, and describe targeted testing.

Table 5.1: Settings in testing frame

variable	description
$coeff\_style$	The coefficients attribute indicate the polynomial order of synthetic data. The data can be linear (x), quadratic (x^2), or cubic (x^3).
$samples$	A diderot program is used to create a nrrd file to represent a tensor field. The program evaluates a polynomial expression at a number of samples.
$space$	Input to the Diderot programs include the number of samples to take and if we randomize the "shear" and "angle" attributes in the Diderot program. Using fewer samples and changing the orientation of the samples creates a stronger field reconstruction test.
num_pos	Generated test programs probe a tensor field at a set number of positions.
krn	The test program uses a specified reconstruction kernel
$layer$	A computation is made up of a set number of nested operators, which can range from one to three.
in_tys, rst_tys	Limitations to argument types and result types, such as only tensor fields.
$random\_limit$	We support the random or exhaustive exploration of test cases. The developer specifies the probability of a single test program being executed, with exhaustive being 100%.

Settings in testing frame The settings indicate data creation factors, test program details, computation details, and type of search to find test cases. Figure 5.2 shows the testing frame object. The settings are initialized by the variables and are defined in Table 5.1.

Scope The scope defines the set of possible programs that can be tested. The scope of the testing is one of three modes:

1. Run all possible test cases described under the testing frame.
2. Target (defined later) a group of test cases.
3. Run a single case, *i.e.*, the addition of two 2-d scalar fields.

The scope is independent of the type of search. The exhaustive search with generate all the operators and arguments under the scope while the random search will randomly choose some of them.

Exhaustive or random search We support two different types of searches; exhaustive and random. Every possible combination of operators and arguments types are generated in

the exhaustive setting. It is not always practical to run an exhaustive test and create tens of thousands of programs. In the random mode, the test generator will randomly choose the test cases to create. 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 5.5.2.

Targeted testing We use the phrase targeting testing to mean limiting the testing scope to subset of test cases with a label. The testing scope can be limited to a single test or a group of tests. Each operator has an identifier. A user can use that identifier (with others) as a label to limit the testing scope. Targeting testing can be helpful when testing a new operator added to the language and only creating test programs that use that operator. Being able to specify a group of relevant tests makes it easier and faster to re-test after making changes to the compiler.

5.1.3 Testing overview

Test generation is parameterized by the testing frame (Section 5.1.2). The **test case generator** creates a single or thousands of test cases that fit the parameters set by the testing frame. The **internal testing typechecker** is used to filter out the test cases that do not make mathematical sense. Lastly, if **single test case** passes the type checker then it is created, executed, and evaluated.

Test case Generator A key part of the implementation process is the creation of test cases. An exhaustive generation of test cases can be found by iterating over the various types and operators in the scope. Each test case is then defined by the application of operator(s) to arguments. Internally, the generator represents a test case by using the application object.

Internal testing typechecker We created an internal typechecker to see if the application of an operator to arguments are okay. As input the internal typechecker receives an operator and arguments. If the computation passes then the method returns an initialized application operator (Section 5.1.1).

For every operator, there is a set of restrictions on the arguments. Given the addition between two tensors, the two arguments must have the same type (as seen in Equation 5.6). Given the addition between a tensor and field then the two arguments must have the same shape and the result has a field type. Lastly, when the arguments are both fields they must have the same dimension.

As another example, consider the application of the gradient operator. The argument can only be a scalar field with continuity k>0. The output type is a vector field (in the 2-d or 3-d case) or a scalar (in the 1-d case) with one less level of continuity than the input argument. The application of the gradient of a 3-d scalar field is represented in Equation 5.7.

Single Test case In this chapter we introduce two models: *DATm* a model that evaluates based on equality and *DAVm* a model that evaluates based on symmetry. The models share the same internal representation, test generator, and internal testing typechecker but they differ in how a single test is executed. We give an overview of *DATm* in the Section 5.2 and describe *DAVm* in Section 5.3.

5.2 Diderot's Automatic Testing model

This section introduces *DATm* as illustrated in Figure 5.3. 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.

For each test case *DATm* creates synthetic data, a list of positions, a ground truth, and

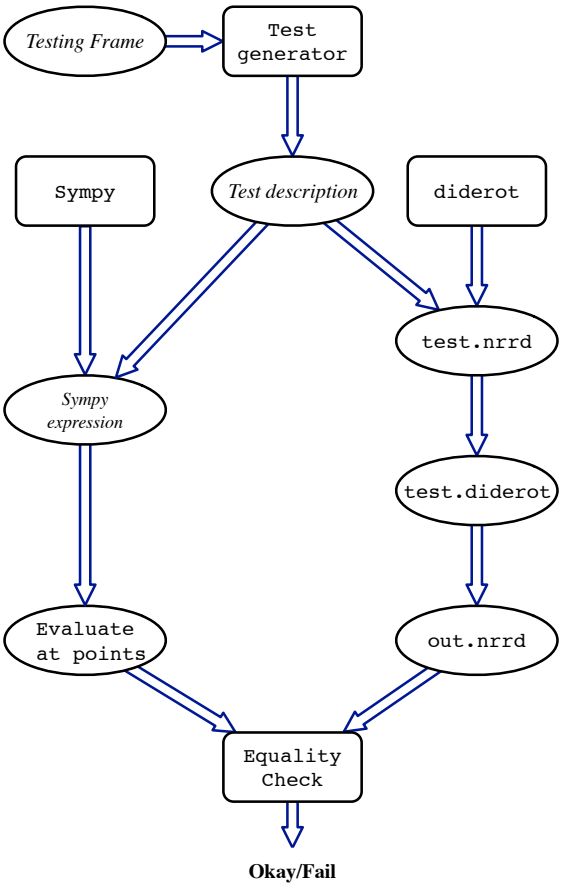

Figure 5.3: Pipeline of Diderot's Automated Testing model. They evaluate based on numerical accuracy and equality.

a Diderot test program (Section 5.2.1). Synthetic data is created for each argument and is represented both by a data file and a symbolic expression (Section 5.2.2). The symbolic expression is used to define the ground truth for each test case (Section 5.2.4). A diderot program a generated for each test case (Section 5.2.3). The output of the executed Diderot program is a data file.

The *test output* is a nrrd file created by Diderot and a Sympy expression. We expect that both outputs reduce to a list of numbers and evaluate the test by comparing them (Section 5.2.5). The *expected behavior* is numerical equality between the output. There are three modes of failure: type error (T), compile error (C), and numerical error (N). A type error indicates a program could not run because there was some unexpected restriction in the type-checker. A compile error indicates a program that otherwise could not compile. A numerical error indicates that the numerical output is not equal to the expected value.

In this chapter we present the pipeline for *DATm*. We provide the steps in generating a single test in Section 5.2.1. We describe the individual steps: data creation (Section 5.2.2), expected solution (Section 5.2.4), generated diderot program (Section 5.2.3), and evaluation (Section 5.2.5). Lastly, we discuss *DATm*'s restrictions (Section 5.2.6) and advantages (Section 5.2.7).

5.2.1 A single test case example

In the following we use a single test case as an example. We compute the outer product between a vector and the gradient of a scalar field. Internally, *DATm* will represent this computation as an application of operators to arguments (reproduced Equation 5.8 here),

$$\text{app}_2 = \{opr : \text{opr}_{outer}, oty : \text{ty}_{fm33d3}, lhs : \text{app}_1, rhs : \text{ten}_1\}$$

After creating the application to represent the computation *DATm* creates, runs, and evaluates a single test case. We present the implementation steps in Table 5.2.

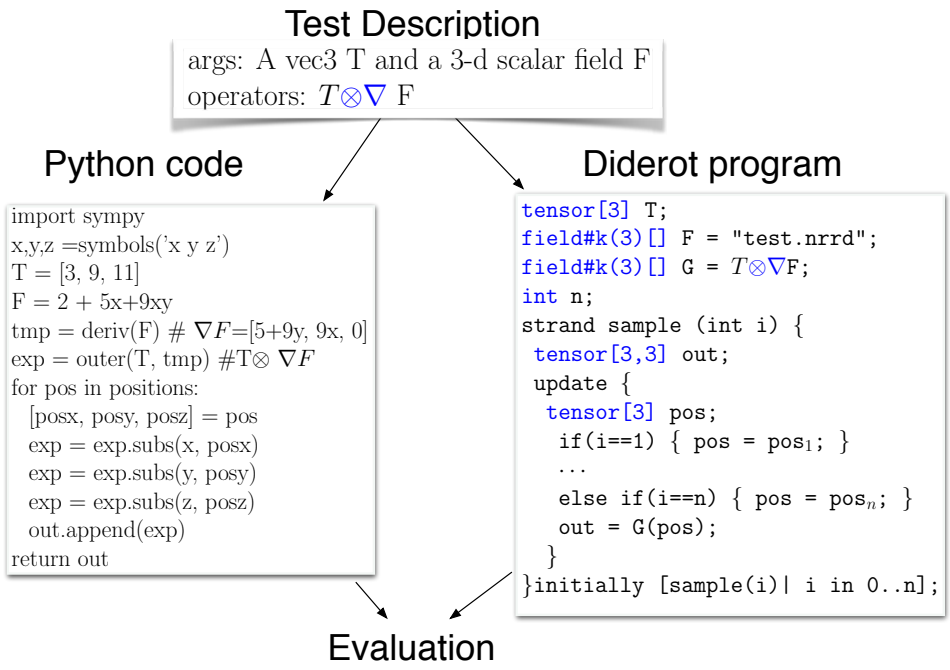

Figure 5.4: Single test case in *DATm*

Table 5.2: Implementation steps for a single test case. We make a reference to variables and names in Figure 5.4.

Description	Reference in code
pull settings from the testing frame	
create positions	"pos" in code
create data files for arguments	test.nrrd in code
generate test program	"Diderot program" box
derive analytical solution	"Python code" box
compare observed data to analytical data	"Evaluation" box

Figure 5.4 illustrates the test case as four different parts: "Test Description," "Python Code," "Diderot program," and "Evaluation." On the left-hand of the figure we show a sketch of "Python code" that create a symbolic expression and generates the expected result. On the right-hand side of the figure we show the "Diderot program" that creates a tensor field and generates the observed result. The symbolic expression is evaluated and the tensor field in the Diderot program is probed at the same series of points We compare the output for both in "Evaluation". The rest of this section describes the individual phases in more detail.

5.2.2 Data Creation

DATm automatically generates the synthetic data used in the testing process. Each argument in a test case is defined by a list of "coefficients." Each argument has a representation in Diderot and in Python.

Coefficients *DATm* creates random numbers to represent the test arguments. The random numbers serve as coefficients to a linear, quadratic, or cubic polynomial expression. For each argument in a test case, we have a list of coefficients. We use that list to create a representation of the argument in Diderot and in Python.

Diderot representation In a Diderot program, a tensor field is defined by an image file (in a nrrd format [71]) and a reconstruction kernel. The data file maintains sampling orientation and discrete data points. We create a nrrd file to represent each argument that is a tensor field.

DATm has a set of Diderot templates to create fields of type **field**#k(d)[κ] (τ as defined by Figure 5.1). The template is initiated by a list of coefficients, number of samples, and orientation. The number of samples and orientation are found in the testing frame.

The program represents the polynomial expression described by the list of coefficients. 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 5.3). It is used later as input to a Diderot test program.

Python representation The operators we are testing are tensor calculus-based operators. Since our operators are based on mathematics we are able to use the Python sympy package to analytically derive the correct solution [68]. For each argument in a test case a symbolic expression is created from the list of coefficients.

5.2.3 Diderot test program

DATm supports the creation of Diderot programs by translating the test description into Diderot code. The test description is internally represented as data structures (defined in Section 5.1.1). In this section we show examples of the internal representation followed by Diderot code.

Arguments Each argument is represented with argument object. The type arguments are translated into Diderot types. For example,the tensor argument

$$\text{ten}_T = \{\mathit{fldty} : \text{ty}_{tv2}, \mathit{data} : [\mathit{expT}_1, \mathit{expT}_2], \mathit{krn} : \text{Null}, \mathit{input} : \text{Null}\}$$

is translated to the following Diderot type and the data value for tensors are inlined.

```
tensor[2]T = [expT1 ,expT2 ];
```

Field objects are translated into Diderot field types.

$$\text{fld}_F = \{\mathit{fldty} : \text{ty}_{fv2d2}, \mathit{krn} : \text{krn}_{kex}, \mathit{input} : \text{“F”}, \mathit{data} : \ldots\}$$

Tensor fields are set equal to the convolution between kernels and the names of the input file (that were created in a previous step).

```
field#k(2)[2]T = load(``F.nrrd")⊛ c4hexic;
```

Application The test description includes the application of the outer product operator between a tensor field and tensor

$$\text{app}_1 = \{opr : \text{opr}_{outer}, oty : \text{ty}_{fm22d2}, lhs : \text{fld}_F, rhs : \text{ten}_T\}$$

then taking the 2-d matrix inverse of the result.

$$\text{app}_2 = \{opr : \text{opr}_{inv}, oty : \text{ty}_{fm22d2}, lhs : \text{app}_1, rhs : \text{Null}\}$$

The application object is translated to the following Diderot code.

```
field#k(2)[2,2]G = inv (F ⊗ T);
```

The field is probed at multiple positions in the field domain.

Rest of program The rest of the program is automatically generated. The core of a test program is a tensor field sampled at a position $(pos_1, pos_2 \ldots pos_{n-1})$. A strand in a diderot program initiates the position value (pos), while the tensor field computation remains the same.

```
if(i==0){pos = pos_1;}
else if(i==1){pos = pos_2;}
...
else if(i==n){pos = pos_{n-1};}
```

An inside test is imposed to be sure the position is in the field domain.

```
if(inside(F,pos)){... }
```

The result of computation is the observed value.

```
tensor[2,2] observed = G(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.

5.2.4 Analytically derived solution

We translate the test description to a symbolic expression that can be evaluated. The test description indicates the argument objects for each tensor field argument. Each argument has a list of numbers (data) that represent coefficients to a polynomial expression. The arguments are represented in python as expressions.

$$\begin{aligned} exp_1 &= x^2 + 3x + 4 \\ exp_2 &= 7x - 1 \end{aligned}$$

Each test description also includes the operators that are used in the test case. Each operator is matched to a function that will apply a manipulation on the polynomial expression based on the argument types. For instance, the expression can be symbolically differentiated.

$$exp_3 = \nabla exp_1 = 2x + 3$$

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

$$exp_4 = exp_3 + exp_2 = 9x + 2$$

After the operators are applied the expression is evaluated at positions.

$$exp_4(x = 1) = 11$$

The computation reduces to a series of number that represent the ground truth for a given test case.

5.2.5 Evaluation

We expect to be able to compare the test output based on equality. If the output is within some error tolerance then the test passes. There are three different possible failure modes.

- Type error.
- Compilation error
- Numerical error

A type error indicates an issue with the Diderot type checker. A type error can occur because the Diderot implementation did not have the language support 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 within the error tolerance of the analytically derived result.

5.2.6 Checking limitations of the Diderot programs

In the past, compiler testing tools have evaluated tests by comparing the output of different versions of a compiler [74], or have asked the human user to supply a criterion that can be checked automatically [23]. 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 with an analytically derived solution. The potential rounding errors that can occur in floating point arithmetic are well-known [27]. In *DATm* it is possible that numerical errors will result in a false positive.

DATm does take some precautions against doing operations with undefined results. As mentioned in Section 5.1.1, the operations used in the Diderot program might have certain restrictions. The argument(s) to the operation might have to be within a certain range of values. If so, the test program generates if statements to check that these conditions are met. If the conditions are not met then that strand is invalid. If not enough strands pass this condition then the test is thrown out.

For example, the square root operation $\sqrt{e}$ is tagged by a condition that limits its argument to positive numbers (Equation 5.1). For each strand, if the expression evaluated for the position is positive then the computation is evaluated and added to the output. Otherwise the strand is not "included" in the output.

```
//strand indicates position value
if(i==0){pos=...}
else if(i==1){pos=...}
//check limitations
if(e(pos)>0){observed = √e(pos);//include item}
else{observed=null//do not include item}
```

Returning to the example (used in Section 5.2.3), an inverse operation involves dividing by the determinant. When dividing we prefer to restrict the denominator value so its magnitude is large than some small value (ϵ). That limitation is indicated in the definition of the inverse operator. *DATm* generates a test on that condition

```
if(|det(F⊗T)|> ε){observed = G(pos);}
```

where the denominator in the division operator must be larger than some small value (ϵ).

5.2.7 Advantages

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* is compromised of three steps:

1. Define the operator with its arity and output type (as shown in Section 5.1.1). Also, initiate attributes placement and limitations, which facilitate the scripting process that creates Diderot test programs.
2. Add a case to the internal type checker (Section 5.1.3).
3. Add a case to apply the operator to polynomial expression(s) (Section 5.2.4)

The new operator can then be tested exhaustively using *DATm* with different argument

types and in combination of existing operators. The slight cost of adding a new operator to *DATm* is well worth the return.

Uncommon programs *DATm* offers more extensive testing of the language than is expected to be found in hand-written Diderot programs. In the past, the testing and development of Diderot relied on hand-written programs centered on commonly used combinations of computations and arguments. This kind of testing biased the discovery of bugs. *DATm* expands the search for bugs from what was convenient and typical usage to what is possible in the Diderot language.

It is worth noting that there are other uncommon field types, that can be exposed to some amount of testing (γ in Figure 5.1). Intermediate steps in the testing process can apply a single operator $op_1(\tau_1, \tau_2) \longrightarrow \tau_3$ where the result type is outside of our scope (*i.e.*, $\tau_3 \notin \tau, \tau_3 \in \gamma$). Application of a second operator (*i.e.*, $op_2(op_1(\tau_1, \tau_2)) \longrightarrow op_2(\tau_3) \longrightarrow \dots$) tests a more extensive range of operations than can be created directly from templates.

As an example, consider the computation

```
tensor[3]t;
field#k(1)[2,2]F;
field#k(1)[3,2,2]G = -(t ⊗ F);
```

Synthetic data can be created for t and F, but currently not for G, which has a rarely used type **field**$\#k(1)[3,2,2]$. Teem [71] 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.

5.3 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 those that *DATm* is able to test. It is possible that this additional complexity could bring to light more bugs. Evaluating the results

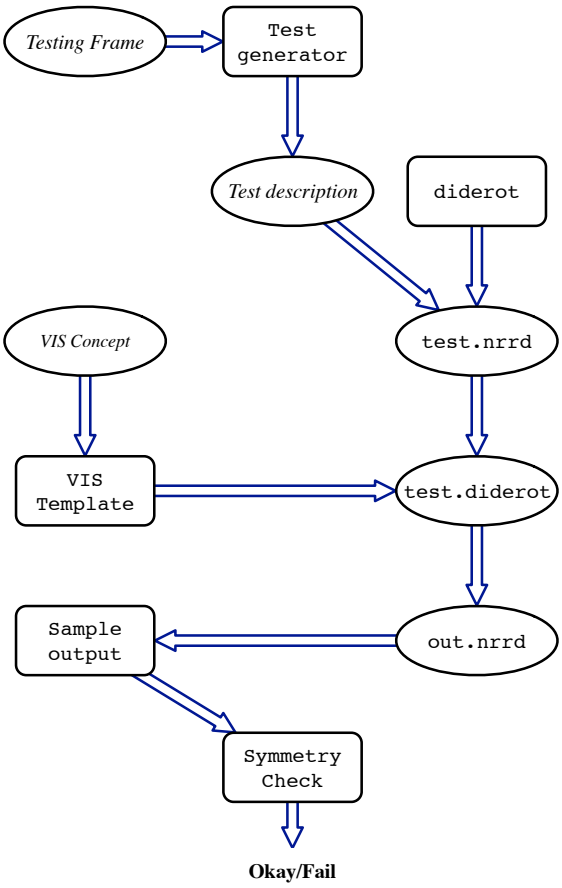

Figure 5.5: vname. They evaluate based on visual verification and symmetry.

of a visualization program based on the numerical output can be difficult (if not impossible), because sometimes we cannot easily know what the algebraic solution is. Therefore, each visualization program had required an "eyeball test", just looking at the resulting image. In an effort to support more types of testing, we have created Diderot's Automated Visualization model *DAVm*, which uses a known property to evaluate an unknown result.

5.3.1 Concept

To test the domain-specific applications of 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 symmetric 3-D field onto a plane, restricted to a spherical domain, should produce a rotationally symmetric 2-D image, regardless of the point of view point and the field operations involved. *DAVm* generates two Diderot programs that will do a volume rendering of the computation and another to sample the output of the volume rendering. Figure 5.6 provides an example of the output from these programs. In the following, we describe how *DAVm* creates a single test case.

5.3.2 Pipeline

Figure 5.5 shows the pipeline for *DAVm*. *DAVm* shares much of the same basic code with *DATm*. Specifically, they have the same internal representation and test generator. They differ in the template used when writing a Diderot program and how test programs are evaluated.

Restrictions The test generator is used to automatically create test cases but require that the result of the computation is a 3-D scalar field, the types of the arguments are restricted to 3-D fields, and the data generated to synthesize tensors and tensor fields is symmetric.

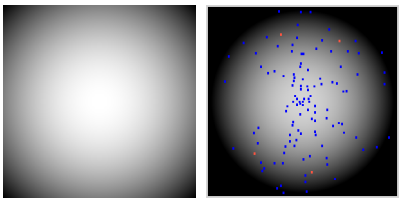

Figure 5.6: Volume rendering of a symmetric 3-D field using Summation projection (left) and Maximal projection (right) with positions that are sampled to evaluate correctness. We sampled 30 groups of 4 points that were equal distance from the center. Highlighted in red is one of the groups.

VIS Template A template is used as base to create Diderot test programs. In *DAVm* we use a Maximally Intensity Projection (MIP) program [43] as the base code for a basic volume rendering program in Diderot. The core of the MIP program gets the maximum value along a ray and stores it into the output variable out.

```
out = max(out, G(pos));
```

Another template can be used to do a summation projection volume rendering by using:

```
out += out;
```

As in *DATm*, the test description is translated into Diderot code. Type objects are translated into Diderot types and the application objects are also translated into Diderot operators.

Sample output After writing, compiling, and running the Diderot test program we sample the output. We sample the output at groups of points that are equal distant from the center of the image. For each point, the x-coordinate is chosen randomly, and the corresponding y-coordinate is chosen using the distance formula. We expect that probing the output at points equal distance from the center have the same value. We show an example of the output and sampling an output in Figure 5.6.

Symmetry Check The correct numerical answers for these visualization programs are unknown (the projections would involve potentially unwieldy symbolic integrations) but we can still detect bugs. We test the symmetry of the results by sampling the output at points we expect to have the same value and then compare them. If they are within some threshold (ϵ) then the test passes. To establish ϵ we generated a few hundred programs and used an eyeball test to determine when the difference between the value of points is meaningful.

5.4 Bugs

DATm has found various bugs in the Diderot compiler when *DATm* was developed. As previously discussed in Section 5.2.5, there are three different failures modes: type errors, compile errors, and numerical errors. *DATm* found eight type errors (Figure 5.3), seven compile time errors (Figure 5.4), and five numerical errors (Figure 5.5). While the number of bugs is not large, some of the compile-error and numerical-error bugs were caused by unique mistakes in the compiler.

DATm discovered several bugs that could only arise with specific combinations of operators. Bugs of this nature are unlikely to be found by handwritten tests and are difficult to identify in the code. The type-error bugs account for missing features and unexpected restrictions in the implementation. In the following we describe five bugs: a numerical bug, three compilation bugs, and a type error bug. The first four bugs are examples of bugs that would have been especially difficult to find without *DATm*.

Numerical error caused by complicated transformation This bug (B1) is an example of a numerical error in the output, which was exposed by computing the transpose of the Jacobian

```
field#k(3)[3,3]G = transpose(∇⊗(V));
```

The Diderot compiler must generate code to map derivatives from image-index space to world space (Section 2.3). 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.

Bug exposed by testing nested operators Programs with nested operators offer a more rigorous test of the compiler than simple programs. The bigger computations are optimized inside the compiler. Creating bigger computations with nested operators is a way to test the optimization steps inside the compiler. The trace of the modulate between a negation of A and B (B15) can be computed as

```
G = trace(modulate(-A, B));
```

The application of the trace operator on modulate is okay. It is the use of a third operator that triggers to use of the *Split* method (Section 3.3.3). The computation raised a compilation error because the case was not handled correctly by *Split*.

Design issue in EIN IR This bug (B9) emphasizes the need to test every operator in combination with each other. The cross product $\times$ and slice operations worked correctly on their own, but combining them creates a bug. When they were combined in a Diderot test program

```
(U × V)[1];
```

it created a compilation error, because the necessary rewrites were not supported. The computation exposed a design issue in the IR of the compiler.

Parameter ids A careless error (B10) occurs when we compute the differentiation of the subtraction of a tensor and tensor field.

```
field#k(d)[d']F;
tensor[d']T;
field#k(d)[d']G = T - F;
field#k(d)[d', d]J = ∇⊗G;
```

The differentiation of a scalar T is zero. While the EIN expression that represented this variable went to zero the EIN parameter remained. The parameter ids need to be reset, but were not done correctly.

Language expressivity An unexpected benefit of implementing *DATm* was the check to Diderot's advertised expressivity [44]. Previous work offered a type specification for the Diderot language. As an example, consider the type specification for the trace operation on fields:

```
trace : field#k(d)[i,i] -> field#k(d)[]
```

But in actuality, the implementation provided trace with the following more restrictive type:

```
trace : field#k(d)[d,d] -> field#k(d)[]
```

This restriction (B16) did not cause numerically incorrect results, but did limit the expressiveness of the language.

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 the goal. *DATm* has found cases 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 complete language.

5.5 Results and Performance

In this section, we present four experiments that evaluate *DATm*. The first is an experiment that measures the difference between exhaustive and random search. The second experiment measures the time it takes to implement a single test case for different argument types. The third experiment uses *DAVm*. It demonstrates that it is possible to do other types of tests with an extension of the testing model. The fourth experiment applies *DATm* to different

Table 5.3: The list of type-error bugs are categorized by the number of *nested* operators needed to discover the bug, the cause of the bug, and description. The cause of the error could be because of missing features E or other O.

Number	Nested	Cause	Description
B1	1	O	Restriction on trace
B2	1	O	Restriction on transpose
B3	1	O	Restriction on slice
B4	2	O	Layout in using computations inline
B5	1	E	Restriction on determinant
B6	1	E	Restriction on modulate
B7	1	E	Missing $\times$ between tensor and field
B8	1	E	Generality $\otimes$ between tensor and field

Table 5.4: The list of compilation bugs are categorized by the number of *nested* operators needed to discover the bug, cause of the bug, and description. The cause of the error could be because of transformation rewrite R, or other O.

Number	Nested	Cause	Description
B9	2	O	Design in EIN IR indices Ex. $(U \times V)[1]$
B10	2	R	Rewrite Parameter Id Ex.$\nabla(T + F)$
B11	2	R	Wrapping summation in rewrite rule
B12	2	R	Check dimension in Optimization
B13	2	R	Translating to vector code. Vectorize EIN without variable index
B14	2	R	Optimization not generalized
B15	3	R	Redundant indices in a single term Ex. $trace(modulate(-U, V))$

Table 5.5: The list of numerical bugs are categorized by the number of *nested* operators needed to discover the bug, the cause of the bug, and description. The cause of the error could be because of transformation rewrite R, or other O.

Number	Nested	Cause	Description
B16	2	R	Rewrite field reconstruction indices Ex. transpose($\nabla \otimes V$)
B17	2	R	Data accessed incorrectly. Reverse order for second order F
B18	1	O	Algebraic error creating EIN operator. Ex. $\|\|\varphi\|\|$
B19	1	O	Creating the concatenation operator
B20	2	R	Determinant(concat(F,G))

snapshots of the compiler. This experiment demonstrates that many bugs were not being caught until we developed *DATm*.

5.5.1 Experimental Framework

The experiments were run on an Apple iMac with a 2.7 GHz Intel core i5 processor, 8GB of 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, unless stated otherwise. Quadratic coefficients were used to create synthetic data. The nrrd file is created by taking 70 samples and not randomizing the sampling orientation. The generated test programs used c4hexic kernels to reconstruct tensor fields and seven different positions to probe the fields.

5.5.2 Exhaustive vs. Random Testing

The first experiment compares 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 generated. We report the effect of changing the setting by representing the time required for testing and the number of test cases explored.

Our experiment does a single exhaustive search and four random searches for up to three nested operators. An exhaustive search attempts all possible test programs. A random search has a certain probability of trying each test program. Table 5.6 records the timing measurement from doing these different searches and varying the number of nested operators. The measurements range from seconds to hours. Table 5.6 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.

By creating a large range of test cases it is possible to find hidden and unique bugs.

Table 5.6: 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 time measurement (in minutes) for running *DATm* with each type of search.

	Total	Timing to run *DATm* with given probability.					
No.of Operators	No. of Programs	0 %	0.5 %	1%	5%	10%	100%
1	695	0.25	0.3	0.55	2.6	4.65	32.53
2	18,819	7.62	14.43	18.35	64.2	121.23	1099.16
3	495,626	58.83	246.02	393.2	-	-	-

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.

5.5.3 Breakdown of a single test case

Our experiment runs a random search and generates 1% of test cases. The frame used two layers of operators and linear elements. We run *DATm* twice: first using any type of argument and second restricting arguments to tensor fields. We measure the time spent generating a single test case in *DATm*. We previously showed the implementation steps in Table 5.2, but we add more individual steps for a better analysis.

Figure 5.8 shows the average breakdown of time spent in the core part of *DATm*. The **data** step creates data files for (field) arguments and is separated into four parts: compiling the C compiler (41%), compiling the diderot compiler (11%), running (4%), and other (3%). The **test** step generates the test program and is separated into four parts: writing (1%), compiling the C compiler (25%), compiling the diderot compiler (1%), running (1%), and other (2%). The **analytical solution** uses a Sympy package to get ground-truth solution (9%). The **evaluate** step compares the observed data and the correct solution (1%). Lastly, the **other** category makes up a small portion of the time (1%) but includes pulling settings from the testing frame, creating positions, reading data from a file, and tracking results.

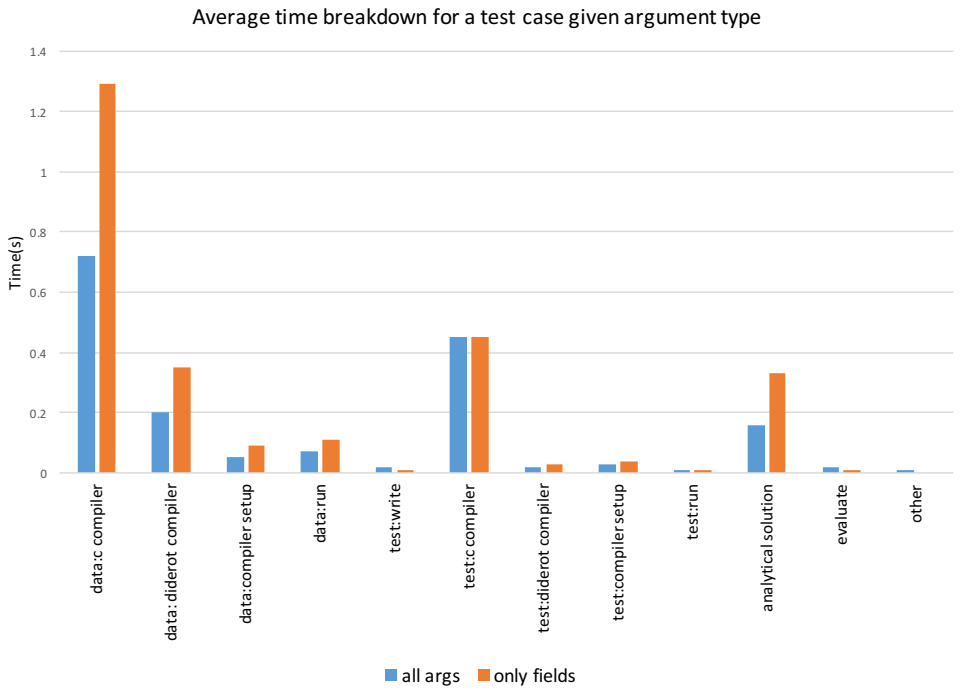

Figure 5.7: The average time spent creating and testing given different argument types. The average time spent for different argument types. Time is in seconds

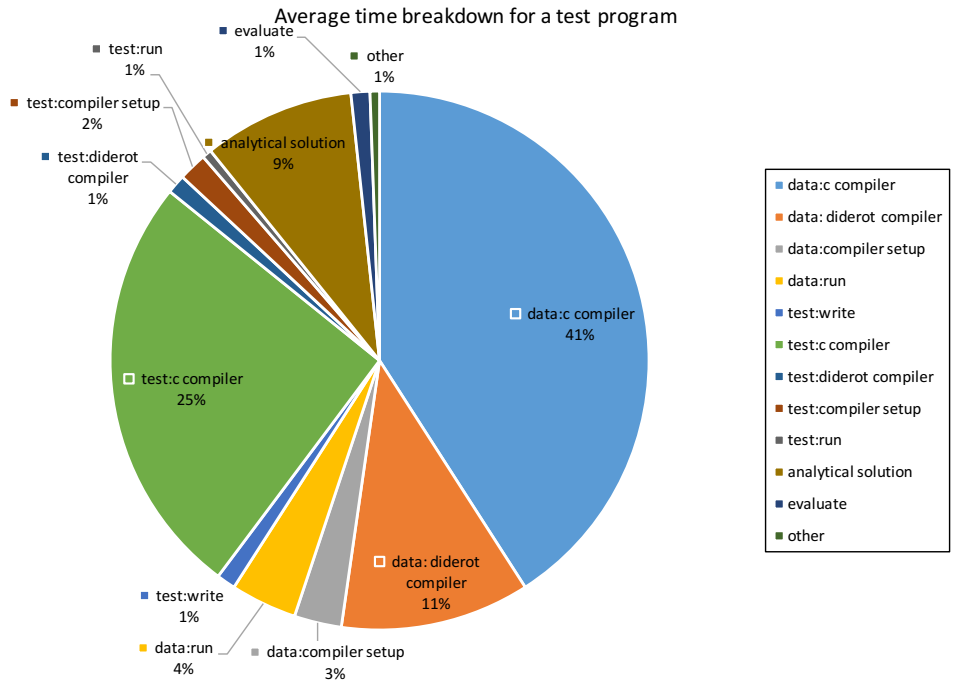

Figure 5.8: The average breakdown of a single test case. Time is in seconds

This experiment shows how the time in a single test case is spent. It can be useful to show places for improvement. On average, 88% of the time is spent on the Diderot programs. 66% is spent on the C compiler, 12% is spent on the Diderot compiler, 5% on running Diderot programs, and the remaining on the setup. Other parts of the testing infrastructure compromise just about 12%. Not substantial but it could indicate room for improvement.

We run *DATm* twice: first using any type of argument and second restricting arguments to only tensor fields. We created 179 tests in the first setting and 47 in the last. Figure 5.7 shows the results from applying *DATm* in these two different settings. The average total time spent per test in the core part of *DATm* is 1.8 and 2.7 seconds for all arguments to just tensor fields, respectively.

When only using field arguments the average time is longer. A Diderot template is used to create synthetic data for tensor field arguments. That step is not necessary for tensor arguments so when using a tensor arguments we expect the average time to be less.

5.5.4 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 5.6 is an example of the output from these tests. 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 Table 5.7.

DAVm is a prototype to illustrate *DATm* can be used for other types of testing. *DAVm* has not found any errors in Diderot, but it was created after Diderot has already been extensively tested 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, and it takes longer to execute a visualization test program. As a result, *DAVm*

Table 5.7: Results from running *DAVm*. The time is given in minutes for both an exhaustive and random search.

No. of Operators	No. of Programs	Time given probability. 1 %	100 %
1	15	.01	16
2	216	3.13	344
3	3,151	81.63	-

does not create nearly as many tests as *DATm* for a given test size.

5.5.5 Snapshots of the Diderot compiler

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 pulled off the Diderot repository at four-month intervals, starting from March 2015. The experiment is a post evaluation of the state of the compiler. We used the same settings as before and we used exhaustive search with two nested operators, which generated almost 19,000 tests.

The results are organized by three different categories, "failed," "compilation error," or "passed." The "failed" description means that there was a bug because the numerical result was not correct, or there was an run-time 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 test programs that fall into these different categories. Figure 5.9 provides the results from the experiment.

Over time, the number of tests that pass increases. Until *DATm* is introduced, the compilation errors decrease but the numerical errors increase. It is unclear if the compilation errors became numerical errors or there were other errors introduced during development and not caught. The experiment does not indicate the number of bugs and we expect that many

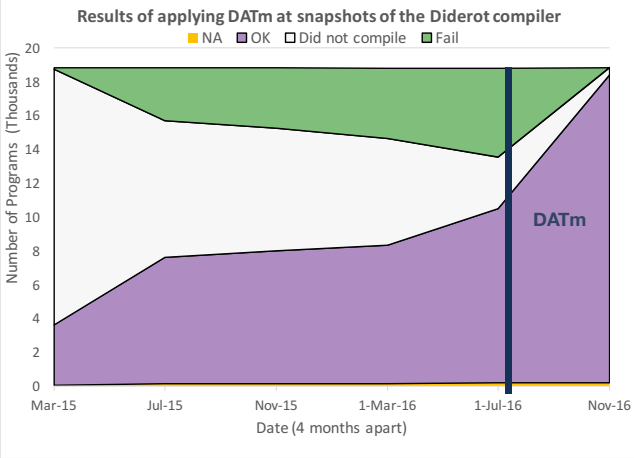

Figure 5.9: Results when running *DATm* over time. Categories"OK", "Fail", "Did not compiler", and "NA" indicates a program passed, failed, it did not compile, or was thrown out, because conditions set by the operators were not met, respectively. The vertical line marks when *DATm* was introduced.

of the tests fail because of the same compiler bug. Various compilation errors may be due to earlier versions of the implementation not fully supporting the newer language features. 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 but there is a small gap of programs that did not compile. After *DATm* revealed a bug (B15) when using modulate we realized modulate should only be supported between vectors and restricted it in the Diderot typechecker. At the time of the experiment the *DATm* type checker (Section 5.1.3) applied the modulate operator to arbitrary tensors and generic fields. Therefore, *DATm* correctly created Diderot test programs that applied modulate to generic arguments, but Diderot could no longer compile them. The gap then best illustrates the mismatch in the expected support (*DATm* typechecker) and the observed support (Diderot language).

5.6 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, which would be unmanageable to write by hand. These programs include 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. *DATm* can track failed tests and enables reproducibility. It is possible to make copies of every program that fails, but that would be unmanageable. In lieu of that, we use labels to apply targeted testing (Section 5.1.2). 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 kinds of errors range from type checking to those deep inside the compiler. Some of these bugs were difficult to find because they only arise with a unique combination of operators (Section 5.4) and a developer is unlikely to try those combinations.

Types of Test Programs An earlier section introduced a way to do visual verification on programs with unknown algebraic solutions. Our application of *DAVm* demonstrated how volume rendering templates can be used to generate and run test programs. The templates can be slightly altered to do 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 the Diderot compiler by creating thousands of tests and evaluating the results based on ground truth. We have described the details of implementing *DATm* as well as an extension to create visual verification tests. We have provided examples of bugs that *DATm* was able to find. The use of *DATm* has made the compiler more reliable and correct. There are other types of improvements we can make

towards the testing process, such as including other types of testing strategies (see future work Section 9.1.1).

CHAPTER 6

EXTENDING DIDEROT

The scientific community use PDEs to model a range of problem. The people in this domain are interested in visualizing their results, but existing mechanisms for visualization can not handle the full richness of computations in the domain. We did an exploration to see how Diderot could be used to solve this problem. This chapter describes how we use Diderot to visualize data created from solving PDEs.

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). These PDEs can be used to describe complex phenomena like turbulent fluid flow. The solution to PDEs or the output to software that solves PDEs are sometimes referred to as "FEM fields". The Diderot language does not know how to solve PDEs, or represent FEM fields, but it does have the computational model to visualize fields. The work in this chapter takes a step towards using Diderot to visualize FEM fields. With this work 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 transition from solving a PDE to visualizing its solution, scientists may turn to standard visualization tools to analyze their data. The problem with this approach is that there is not a universal solution to accurately visualize every PDE. For example, 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 believe that Diderot can be useful.

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. 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).

Our work demonstrates a modest step towards visualizing FEM data with Diderot. This chapter is organized as follows. Section 6.1 offers some background about this problem area and provides a motivating example. Section 6.2 describes the implementation details to our approach. Section 6.3 demonstrates an application of our approach by providing one example of the Helmholtz equation and interpolating a function. We end with a discussion in Section 6.4.

6.1 Motivation

The Finite Element Method [10] gives a general framework for computing solutions to differential equations. In Section 6.1.1, we more closely describe FEM fields and the software used to create them. In Section 6.1.2 we describe the state of using visualization toolkits to visualize a certain domain of FEM fields. We provide an example of a problem that can not be visualized with the current state of visualization.

6.1.1 Background

FEM fields are created from a solving a PDE on a finite element mesh, which involves discretizing the domain into small finite mesh elements and using a set of basis functions (derived from the mesh) to span the domain space. These fields approximate numeric solutions to PDEs. Some of the related work is discussed in more detail in Chapter 8.

FEM Fields The Python code builds on the description of the problem. 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 = UnitSquareMesh(2, 2)
```

The code creates a 2x2 mesh of a square. Each smaller square is divided into two triangles for a total of eight elements. The mesh is one of the arguments when defining a function space.

```
V = FunctionSpace(m, "P", K)
```

"P" refers to a family of finite element spaces (other families 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.

```
f = Function(V).interpolate(Expression(exp))
```

In this chapter, when we wish to create simple examples we choose to generate fields from expressions.

Software Computer scientists build software to solve PDEs that represent a wide range of problems. On the surface the software (or programming language) ideally represents a high-level math notation that is easy to understand but under the hood it is more complicated. The translation between the notation and computer code involves several steps and pieces of software. Solving a PDE involves the discretization of differential equations and uses the finite element method to provide an approximate solution. Optimizing the translation from PDE equation to approximate solution is pursed by many groups including the *FEniCS Project* [29] and *Firedrake* [61].

The FEniCS [29, 53] project is an automated system to find solutions for partial differential equations using the finite element method. 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) [3], 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, such as the The FEniCS Form com-

piler (*FFC*) [4], to generate low-level code. FFC can implement tensor reduction for finite element assembly [48] and aims to accept input from any multilinear variational form and any finite element to generate efficient code.

Firedrake [61] is a similar system that is also used to solve PDEs. In addition to UFL, it uses a modified version of FFC [4] (and currently working on *TSFC*), *FIAT* [47] (and replacing it with *FINAT*), a *PyOP2* interface [62], and *COFFEE* [54]. 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 the abstract syntax trees generated by FFC.

6.1.2 Creating and Visualizing FEM data

There are a number of approaches to supporting scientific visualization. A common way is to use a toolkit such as the Visualization Toolkit (VTK) [65], ParaView [2], and the Insight Toolkit (ITK) [40] or other languages and pieces of software we introduce in Section 8. In this domain it is important for the visualization tool to understand how the data is represented and that restraint limits the options available to the FEM user. Toolkits are commonly used to visualize the solutions created by FEM software.

Existing practices to visualize FEM are insufficient. The strategy to solve PDEs is very different from the algorithms used to visualize the result. 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[1]. 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

1. The images in this chapter are created when Firedrake did an L^2 projection, but now Firedrake uses interpolation to generate linear output.

```
exp = "x[0]*x[0]*(1-x[0])"
m = UnitSquareMesh(2, 2)
V = FunctionSpace(m, "P", K)
f = Function(V).interpolate(Expression(exp))
```

Figure 6.1: The above code is written in Python and used to define a field by interpolating an analytically defined expression given the function space. We define a polynomial expression $x^2(1-x)$ and a unit square mesh (m). The function space V is defined by a mesh (m), the family of finite element spaces (P), and the order of the polynomial (K). The field (f) is defined by interpolating an analytically defined expression given the function space (V). 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).

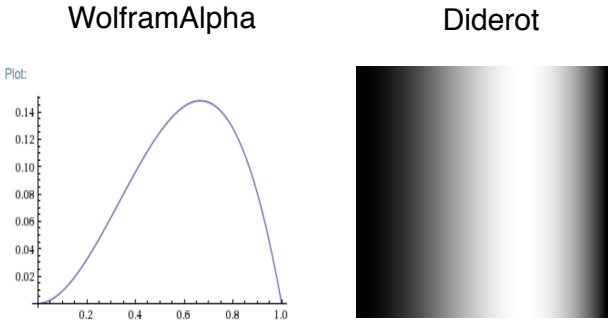

Figure 6.2: Fields created and visualized by a single source. We defined a function $F(x) = x^2(1-x)$ where $x \in [0,1]$. We expect the maximal point to be at x= $\frac{2}{3}$. WolframAlpha can quickly and easily graph the results. In Diderot we synthesized a field by taking samples of a function defined by F and saved the result to *out.nrrd*. Then a second Diderot program was used to sample *out.nrrd* and visualize the result.

accurately represent the PDE solution.

In this section, we present the problem with the current state of visualizing FEM data by providing a simple example of a how a bug can occur. We create a field using higher-order data with a small number of cells. 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 6.2.

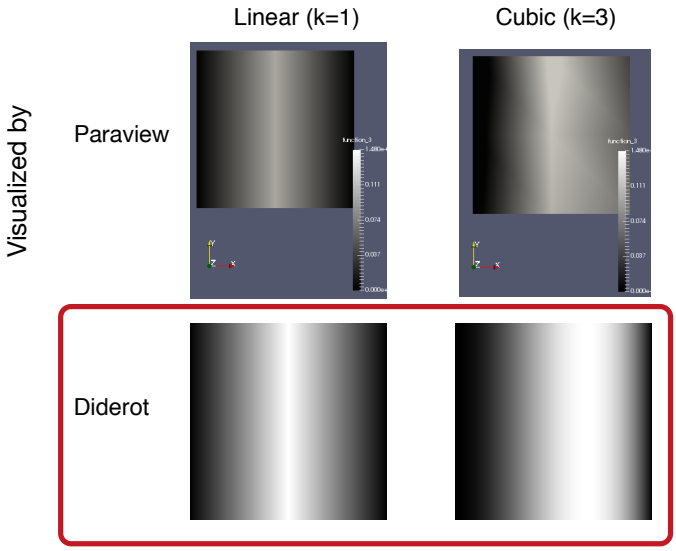

Figure 6.3: Fields created by Firedrake using linear or cubic elements. We defined a function $F(x) = x^2(1-x)$ where $x \in [0,1]$. We expect the maximal point to be at x= $\frac{2}{3}$. In the following grayscale images, the maximal points are indicated by the brighter spots. The two left images use 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 (top right) created with Paraview is incorrect, and the image (bottom right) created with Diderot is correct.

Creating FEM data In the following example, we use Firedrake to create FEM data. Instead of solving a PDE, we define a field by interpolating the expression given the function space. We show the Python code in Figure 6.1.

We define a polynomial expression $x^2(1-x)$. 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.

Visualizing FEM data We chose to visualize the expression several different ways in order to provide a means of comparison. Figure 6.2 and Figure 6.3 present these visualizations. The images in Figure 6.2 offer the ground truth for this example. Figure 6.3 uses Firedrake data created with linear and cubic elements[2]. Regardless of the visualization strategy we expect the images (left) created with linear elements to be an inaccurate representation of the solution. On the other hand, we expect the images created with cubic elements to be correct. The image (bottom right) created with Diderot is correct while the image (top right) created with Paraview is not.

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.

6.2 Our Approach

It is not a long term solution but we created our current prototype to establish communication between Firedrake and Diderot. Firedrake creates FEM data by solving a PDE or interpolating an expression over a function space. Diderot visualizes the results and provides

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.

a few programs that can be initialized. Firedrake provides a way to evaluate a field at a position and provides Diderot with a call back. In the Diderot program a field is defined for FEM data instead of using field reconstruction and convolution. Everything else in the Diderot program is the same. Lastly, when possible we try to evaluate the visualization based on established visualization principles. We worked in collaboration with the Firedrake team [61] at the Imperial College of London.

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 Diderot syntax offers a limited way to represent tensor fields created by FEM data. The FEM community might want to include more details in the field definition such as continuity or mesh. 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 could only represent regular imaging grids. We addressed this limitation by allowing the Diderot compiler to declare a field that is defined by FEM. Instead of creating a field by using field reconstruction and convolution the user can use the following Diderot line to indicate the new kind of field data.

```
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. The following is a sample of Python code that would be written in a Firedrake program.

```
res = 200 # resolution to MIP program
step = 0.01 # step size to ray tracer
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 fields (represented as a FEM field) at arbitrary physical points.It determines which cell to look at. 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.The generated 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 [46]. 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.

6.3 Demonstration

We provide two examples to demonstrate results of this 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.

```
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));}
```

Figure 6.4: The above is Diderot code. 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. The Diderot program will generate code that will communicate to Firedrake's point evaluation capability to reduce F(pos).

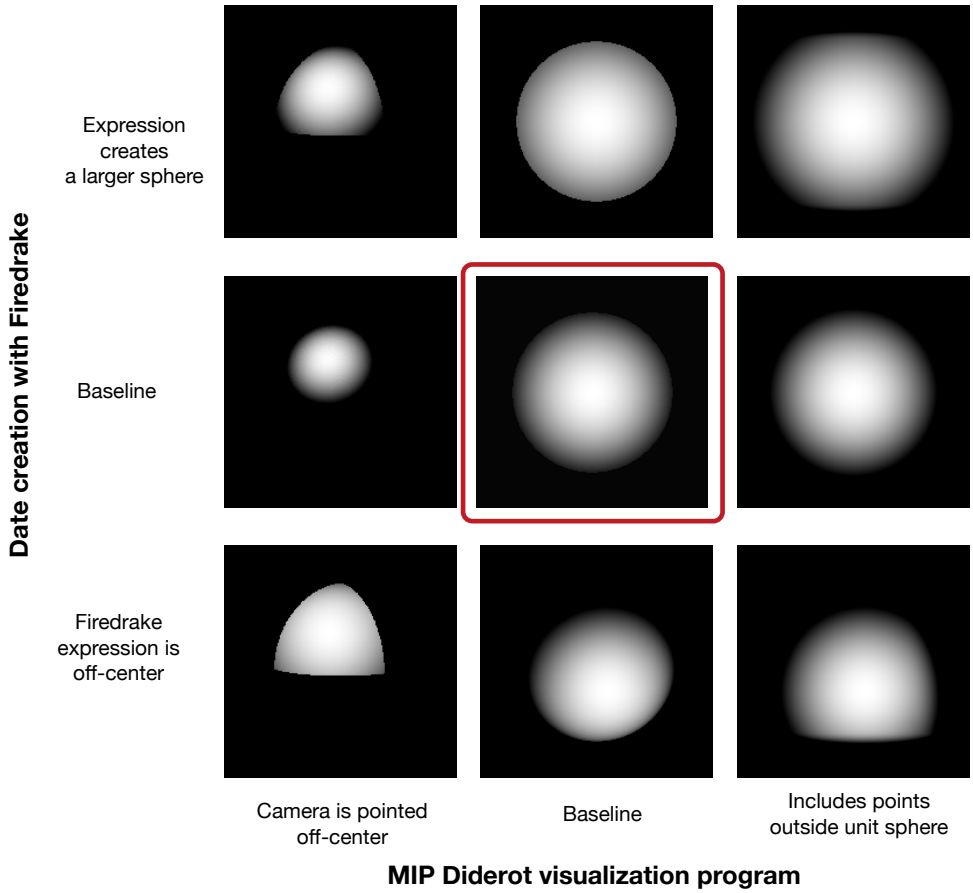

Figure 6.5: Creates sphere with Firedrake. Included are examples of bugs. The centered image is correct.

6.3.1 Communication between Diderot and Firedrake

Python code We create the field defined by interpolating the expression given the function space using Firedrake. We show the Python code under the "Firedrake Field" box in Figure 6.6. We make a call to the Diderot library by passing the field and initializing the resolution and step size (for the Diderot program).

Diderot code Diderot visualizes the result by using an augmented MIP program. MIP or maximum intensity projection is a minimal volume visualization tool for 3-d scalar images. The Diderot code for the Diderot program is presented in Figure 6.4.

Results We visualize the FEM field using a Diderot implementation of a MIP program. We vary 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 6.5. We 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 result is as expected (symmetric sphere). The other images in the figure are subject to bugs, because the function space defined by Firedrake does not match the image space probed by Diderot (or where the camera is pointed). This example illustrates the care that is needed when setting up the Firedrake Diderot pipeline.

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 6.6. As can be expected, the image created by Firedrake is essentially

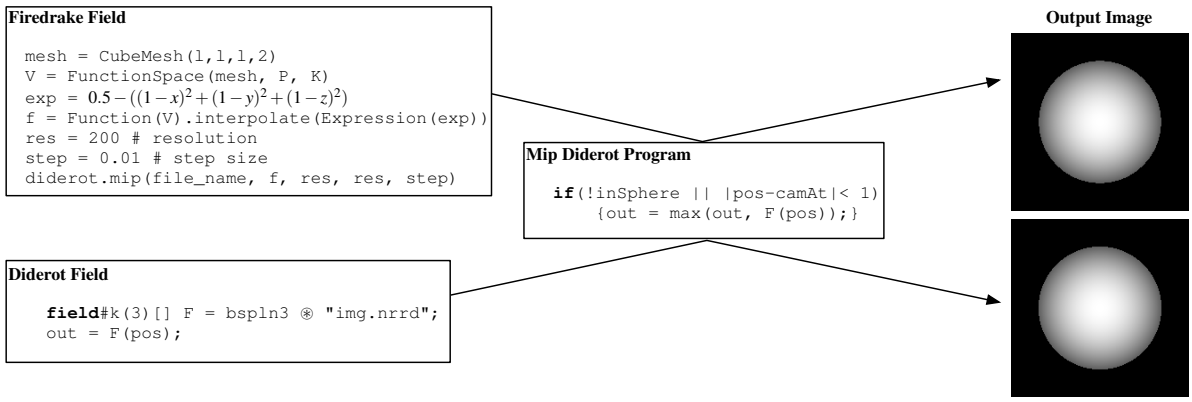

Figure 6.6: This figure compares data created by two different sources and visualized with the same tool. The fields are created by Firedrake and Diderot. In the Firedrake code we create a cube mesh (m) of length 2 on each side. The function space V is defined by a mesh (m), reference element (P), and uses cubic polynomials. The expression creates a symmetrical sphere centered at [1,1,1] and shifted by 0.5. Field (f) is defined by interpolating the expression given the function space (V) . We make a call to Diderot library by passing the field and initializing the resolution and step size.

the same as that created by Diderot.

6.3.2 PDE Example

The Helmholtz problem is a symmetric problem and is a classic example of a PDE. Consider the Helmholtz equation on a unit square Ω with boundary Γ.

$$-\nabla^2 u + u = f$$

$$\nabla u \cdot \vec{n} = 0 \text{ on } \Gamma.$$

The solution to the equation is some function u $\in V$ for some suitable function space V that satisfies both equations. After transforming the equation into weak form, applying a test

```
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'})
# 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, stepSize, type)
```

Figure 6.7: We present the Python code to solve the Helmholtz problem. We omit some details and instead provide comments for clarity.

function V, and integrating over the domain we get the following variational problem[3].

$$\int_\Omega \nabla u \cdot \nabla v + uvdx = \int_\Omega vfdx \tag{6.1}$$

We choose function f[4]

$$f = (1.0 + 8.0\pi^2)cos(2\pi x)cos(2\pi y) \tag{6.2}$$

The Python code (shown in Figure 6.7) solves the PDE and connects the solution to Diderot. 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. The results are shown in Figure 6.8. The image on the bottom right is a difference image and illustrates the difference between the Diderot results. The data with higher-order elements and visualized with Diderot is the more clear and refined. There is a smoothness captured with the higher-order data that is

3. The approach to solve PDEs with FEM is described in more detail in by Brenner and Scott[10].

4. The example and code are provided by Firedrake

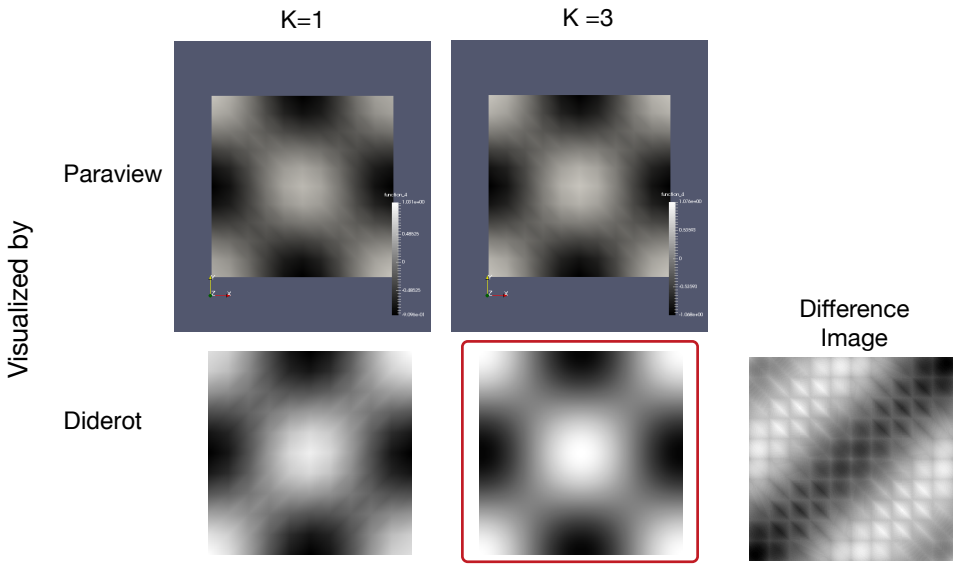

Figure 6.8: The data for the Helmholtz problem is created by Firedrake and visualized by Paraview (top) and Diderot (bottom).

not in the linear data.

6.4 Discussion

The Finite-element community has difficulty visualizing fields that use higher order elements. 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 can 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 6.3.1) this approach can still lead to errors. It takes care, by the user to ensure that a Diderot program is set-up to correctly visualize the Firedrake data.

Our work provides motivation to further develop Diderot and incorporate it with the

FEM community. We have successfully established communication between Firedrake and Diderot but it is limited. A call is made to Firedrake every time a field needs to be evaluated, which makes the communication process expensive. If we move some of the evaluation to Diderot it might be possible to alleviate some of the redundancy that is created with our current method, making the communication process less expensive. In the future it then may be possible to use more mature visualizations for FEM data. We describe ideas for future work in Chapter 9.

CHAPTER 7

APPLICATIONS

We have discussed the contribution of our work in the previous chapters. We have measured the impact of applying our implementation techniques inside the compiler in Chapter 3. We have reported the results from applying our testing models in Chapter 5. We have demonstrated the use of Diderot to visualize fields in another domain in Chapter 6.

In this chapter, we connect some of the work in the dissertation. In Section 7.1, we describe how our 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. In Section 7.3 we describe some of programs that were enabled by our implementation.

7.1 Adding operators to Diderot

Our work has made it easier and faster to add 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 an operator to the language we try to leverage our existing work. This includes a concise representation in an expressive IR (Chapter 2), the generic implementation of operators (Chapter 3), and a robust testing model (Chapter 5). In the following, we illustrate the process of extending Diderot by working through an example of adding a new operator, concat, to the language.

Goal A user can define new tensors by concatenating tensors together. A Diderot program

```
tensor[d1]S;
tensor[d2]T;
tensor[d1,d2]M = [S,T];
```

A user can refer to components of tensor fields by using the slice operation as shown in the following code. A Diderot program

```
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];
```

We would like to provide a way to define new tensors fields by concatenating components together. Using the tensor field variables F and G defined earlier in the program the Diderot code should support the line

```
field#k(d)[d1,d1]H = [F, G];
```

Design and implementation We can use EIN expressions as building blocks to represent field concatenation. In EIN each field term is represented by an expression and it is enabled with a delta function

$$\underset{init}{\Longrightarrow} H = \lambda F, G.\left\langle F_j \delta_{0i} + G_j \delta_{1i} \right\rangle_{i:2,j:2}(\text{F,G})$$

After substitution the new EIN operator would be

$$\underset{subst}{\Longrightarrow} H = \lambda A, B.\left\langle A_{j0} \delta_{0i} + B_{j1} \delta_{1i} \right\rangle_{\hat{i}\hat{j}}(\text{A,B}).$$

In the compiler we choose to create generic versions of an EIN operator that can be instantiated to certain types.

$$\lambda F, G.\langle F_\alpha \delta_{0i} + G_\alpha \delta_{1i} \rangle_{i:2\hat{\alpha}}(\text{F,G})$$
$$\lambda F, G, H.\langle F_\alpha \delta_{0i} + G_\alpha \delta_{1i} + H_\alpha \delta_{2i} \rangle_{i:3\hat{\alpha}}(\text{F,G,H})$$

To implement this operator we need to add to cases to the Diderot typechecker and add the generic representations but not much else. Since we are solely using existing EIN expressions

to represent this computation, we can rely on the existing code to handle the EIN syntax.

Testing Once we have added a new operator to the surface language, it is natural to write some test programs by hand. The tests we wrote by hand were straightforward, but limited and unhelpful since it missed bugs. We can apply a more rigorous approach by using the testing system, *DATm*. We add the concat operator to *DATm* (by creating a new operator object) and used targeted testing to only generate test cases that use the concat operator. *DATm* created and ran 126 test programs that use the concat operator.

We found two bugs in our implementation. One was in the creation of the EIN operator and the second was a deeper bug. The second bug arose when computing the determinant of the concatenation of a field. The bug was caused by the rewriting system. Our rewriting 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 mathematically incorrect to reduce constant indices, because they are not equivalent to variable indices. After fixing the bug we ran the tests again.

7.2 Exploiting Higher-order Operators

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 [41]. The work in this dissertation is designed to make it easier for programmers to illustrate features in their data with Diderot.

Our work in the EIN IR makes it easier to develop algorithms that rely heavily upon higher-order operations. This section describes three visualization features [44, 16] that illustrate the impact and application of designing and implementing the EIN IR.

For scalar fields representing material properties over some scanned region (such as CT or MRI data), the boundaries between materials are important features. To visualize the

continuous boundary as a connected feature of the spatial domain, we can use one of the original definitions of edge detection due Canny [12]:a boundary is the locations in a scalar field F where the gradient magnitude $|\nabla$ F$|$ is maximal with respect to motion along the gradient direction ∇ F$/|\nabla$ F$|$ [12]. One could equivalently state that the gradient of the gradient magnitude ∇ $|\nabla$ F$|$ is orthogonal to the gradient direction *i.e.*,∇F$/|\nabla$F$|\bullet\nabla(|\nabla$F$|)$=0. The computation amounts to showing the zero-crossings of the derived scalar field C.

```
field#2(3)[]C = -∇(|∇F|) • ∇F/|∇F|;
```

The new syntax supports tensor operators (magnitude, inner product, and division) on tensor fields. This allows the user the write the core computation of this concept in high-level Diderot code.

Vector fields arise in the analysis of fluid flow; properties of the derivatives of the vector field characterize important features (such as vortices) in the flow. For example, the curl $\nabla \times V$, indicates the axis direction and magnitude of local rotation. One definition of vortices identifies them with places where the flow direction $\frac{\mathbf{V}}{|\mathbf{V}|}$ aligns with the curl direction $\frac{\nabla\times\mathbf{V}}{|\nabla\times\mathbf{V}|}$ [26]. *Normalized helicity* measures the angle between these directions:

```
field#3(3)[]H = (V/|V|) • (∇×V/|∇×V|);
```

in terms of the vector field V. The new syntax allows the computation to be used directly in a field definition.

Material properties, such as diffusivity and conductivity, vary locally in magnitude, orientation, and 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. For example, neuroscientists study the architecture of human brain white matter with diffusion tensor fields computed from MRI [8]. A popular measure of diffusion anisotropy, "fractional anisotropy" can be directly expressed in Diderot as:

```
field#4(3)[3,3]E = T - trace(T)*identity[T]/3;
```

```
field#4(3)[]A = sqrt(3.0/2.0)*|E|/|T|
```

which measures the magnitude of the purely anisotropic *deviatoric tensor* relative to the tensor field T itself.

Subsequent visualization or analysis 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 ∇ H, and ∇ A by hand is cumbersome and error-prone, whereas EIN allows Diderot to easily handle these and more, such as second derivatives like $\nabla \otimes \nabla$ A.

7.3 Compilation of Tensor Calculus

Enriching Diderot with higher-order operations inspired a new generation of Diderot programs. Unfortunately the compiler suffered from space blow-up and many of these new programs could not be compiled. To address the size issue, we developed compilation techniques (Chapter 3) that reduce to size of the internal representation and enable compilation. The following are some examples of visualization programs that faced the size issue when they were first written, but can now be compiled with Diderot.

Curvature of a surface is defined by the relationship between small positional changes on the surface and changes in the surface normal [45] (we show the program code in Figure 1.1). We can enhance clarity and produce effective renderings in the image created by the program by measuring *Crest Lines*. *Crest Lines* are places were the surface curvature is maximal along the curvature direction [51]. We can find the crest lines by building on the curvature code (as mentioned in Chapter 2) but the program stopped and could not finish compiling until we developed the techniques mentioned in this dissertation.

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 [31, 69]. 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. Initially the programs that use a high-level of math could not compile and suffered from a space blow-up.

The compilation issue was not just if a program would compile but how long it would take to compile. A long compile time could negatively impact a user's experience when using Diderot and hinder rapid prototyping. The experiment section (Chapter 3.4) used several benchmarks with different orders of mathematical expressivity and showed that the techniques are effective at improving compile time. Implementing mathematical expressiveness in Diderot includes addressing the technical issues that arose. By developing the compiler it made possible to use the improved programmability of the Diderot language.

CHAPTER 8

RELATED WORK

In this section, we discuss related work organized by area.

8.1 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[38]. Similar to Diderot, shaders support the ability to perform computations on continuous fields and their derivatives. Shadie is limited to direct-volume rendering applications, whereas Diderot supports other visualization applications.

- *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[55]. Scout is designed for a different class of algorithms than Diderot. Specifically, algorithms that are defined in terms of computations over discrete voxels, such as stencil algorithms, rather than over a continuous tensor fields.

- *Delite* is a framework for implementing embedded parallel DSLs on heterogeneous processors [11, 13]. While the Delite project and Diderot project share the idea that parallel DSLs are an effective way to provide portable parallelism, they differ in the way that the DSL is presented to the user. Delite embeds the DSL in Scala, which limits the notational flexibility of the design, whereas the Diderot syntax is designed to fit its application domain.

- *Vivaldi* is a DSL that supports parallel volume rendering applications on heterogeneous systems [20]. It has a fixed volume rendering vocabulary and does not have the flexible notation that Diderot provides. Specifically, Vivaldi does not support the full range

of higher-order field operators. *ViSlang* is a system to develop and integrate DSLs for visualization [64].

8.2 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 [32]. 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, 7, 21, 28, 66, 67]. 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 notational definitions are created to suppress summation[1]. These include using notation to differentiate between types of indices, using a no-sum operator [7], and differentiate between indices that repeat exactly twice[28, 67]. EIN notation is a compiler IR so the goal is for the IR to supply enough information to represent the wide range of operations. EIN notation uses an explicit summation symbol so it can explicitly set boundaries for diverse operations. The notation leads to more book-keeping but allows more expressivity.

8.3 Intermediate representations and optimizations

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; and the programming model can abstract away from hardware and operating system features. By using a domain specific language, the end-user can write code that is familiar (to them) and let the system focus on generating high-performance code. This notion of developing a high-level mathematical programming model is also emphasized in Diderot. There are various domain-specific languages that provide a link between

mathematical algorithms and programming. This section focuses on four domain-specific compilers that are more closely related to our work (Spiral [60], TCE [49], COFFEE [54], and UFL [3]).

The tensor contraction engine (*TCE*) created by Hartono, Albert *et al.* created a high-level Mathematica style language for the quantum chemistry domain[24, 37, 49]. The class of computations are multi-dimensional summations over products of several arrays. The computations have a large number of nested loops and an explosively large parameter search space. The calculations can require a larger space than available physical memory. To address this issue, Hartono *et al.* developed algebraic transformations to reduce operation counts. Like TCE, the Diderot language supports summations of products over several arrays. Unlike TCE, Diderot supports a high level of expressiveness between tensors and tensor fields and field differentiation.

Spiral is a DSL created for digital signal processing [35, 57, 60]. Its design encapsulates significant mathematical knowledge of algorithms used in digital signal processing. Püschell *et al.* addresses the goal of doing the right transformation at the right level of abstraction. Its implementation uses three levels of IR: SPL to represent the signal processing language, ΣSPL to express loops and index mapping, and *C-IR* for code level optimization. ΣSPL does loop merging and create complicated terms that are simplified with a set of rewriting rules [36]. EIN and ΣSPL both use summation expressions to represent loops, but the Spiral IR expresses algorithms, while the EIN IR expresses general tensor calculus. As with Spiral, Diderot allows its users to focus more on the mathematics and allowing the system to generate high-performance code for their platform. Although Spiral provides a powerful mathematical model, it targeted at the somewhat different domain of signal processing.

The Unified Form Language (*UFL*) is a domain-specific language for representing weak formulations of partial differential equations [3]. UFL is most closely similar to EIN. At its core both UFL and EIN support tensor algebra, high-level expressions with domain-driven abstraction, and offer differentiation (automated vs. symbolic). The projects address

different 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. Therefore, UFL avoids optimizations that a form compiler might want to perform 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[54]. The computation made by COFFEE is key to finding numerical solutions to partial differential equations. COFFEE computes the contribution of a single cell in a discretized domain for a 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. To enable better performance COFFEE applies optimizations. COFFEE applies optimizations on a scalarized tensor expression tree. Like COFFEE, EIN does loop-invariant code motion and expression splitting on tensor expressions. Unlike COFFEE, EIN applies optimizations at a high level to exploit the mathematical properties of the computations on higher-order tensors before flattening

8.4 Evaluating a Visualization

Verifiable visualization allows us to apply a verification process to visualization algorithms [5, 72]. 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 compares them to experimentally observed results in order to provide confidence in behavior [33].

We use the idea of verification [5, 72] as a guiding metric for evaluating 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." [72]. 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. We evaluate the result of test programs written in *DATm* (Chapter 5) based on the previously stated idea.

Kindlmann and Scheideggar introduce three algebraic design principles: The principle of visual-data correspondence, the principle of unambiguous data depiction, and the principle representation invariance [46]. 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 (as defined by [46]). We use the Principle of Representation Invariance to compare two different sources of data in Chapter 6.

8.5 Testing

8.5.1 Domain-Specific Testing

There are a variety of domain-specific languages and frameworks that provide similar features to those that are supported in Diderot, such as the 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 [63, 73]. Wu *et al.* introduces a framework, *DUFT*, to generate unit tests engines for DSLs [73] 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 [63]. Language developers define assertions from the

specification of the DSL and defines a translation into the DSL. Unlike *DATm*, *mbeddr* generates random models that might not pass type-checking rules.

There is work on testing specific properties of general purpose languages. *Herbie* is a tool can be used automatically to improve accuracy in floating point arithmetic [59]. The work introduces a method to evaluate the average accuracy of a floating-point expression and to localize the source of rounding error. It randomly samples inputs, generates rewriting candidates, and discovers rewrites to improve accuracy. Lindig [52] automatically tests consistency of C compilers, specifically C calling conventions, using randomly generated programs. His tool evaluates if the functions (that are being tested) correctly receive parameters. *DATm* creates programs that are semantically more expressive than the programs created by this tool.

8.5.2 Types of Testing

Randomized differential testing (RDT) is way of testing by examining two comparable systems [56, 74, 14, 27]. 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. *Csmith* [74] 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 *DATm*, Csmith effectively looks for deep optimization bugs in atypical combinations of language features. Unlike *DATm*, Csmith evaluates results by comparing various C compilers, and so has no ground truth.

Metamorphic testing is used to evaluate test programs when the correct solution is unknown. With metamorphic testing the test cases are evaluated based on some known property. Donaldson [27] *et al.* applies metamorphic testing to shader-language compilers by using value injection. When comparing images, they note that small differences in rendered images can occur even when there is no compiler bug and that makes it is hard to create a correct solution. *Mettoc* creates a family of programs and compares them using an

equivalence relation [70]. When running *DAVm*, while we do not know what the correct output of the generalized visualization programs is, we do know a property of the the result (symmetry).

Palka et al. generates random and type-correct programs for the Glasgow Haskell compiler [58]. The output of optimized and unoptimized versions of the compiler are compared. *QuickCheck* [23] is a widely used testing tool that allows Haskell programmers to test properties of a program. It is an embedded language for writing properties and can creates test cases that satisfy a condition. The test case generator is limited to a number of candidate test cases. Chen *et al.* [14] 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.

8.5.3 Choosing test cases

There is extensive work on how to create and choose test cases. McKeeman [56] describes test case reduction and test quality with differential testing. When generating test cases *EasyCheck* [22] focuses on traversal strategy. Bernardy, *et al.* [9] present a scheme that leads to a reduction in the number of needed test cases. In addition to fixing types, they also fix some arguments passed to functions, effectively avoiding meaningless tests. *Feat*[30] exhaustively enumerates on the possible values of data types. A *Test set diversity metric* [34] is applied to ensure a diverse set of test cases by using information distance (regardless of datatype) when automatically creating tests.

CHAPTER 9

CONCLUSION

The work in this dissertation has been fundamental to the development of Diderot. 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. We do not currently evaluate testing coverage or measure the how many lines of the Diderot compiler are being tested. In the future it might be possible to use existing tools to mark and measure unused paths in the Diderot compiler.

Smarter Test generation We measured the time it takes to implement a single test case (Section 5.5.3) and we know that the majority of the time is spent on the C compiler. While the portion of time spent in the testing infrastructure is not substantial it indicates that there can be room for improvement. Making the testing process faster makes it more feasible to do a larger search for test cases and as a result apply a more robust testing approach to the language.

One way to cut back on time can be to focus on efficient test case generation. *DATm* generates small Diderot programs, and there is not much need for minimization or reducing the program size. There could be a need to evaluate the distribution of test case. If we wanted to do a random search in a deep test space it might not be feasible to test every test case we find. Instead the tests should be evaluated based on existing cases or by applying some size metric [34, 9].

Clear Bug Reporting There is currently no way to automatically distinguish between the various types of test failures. A single bug could cause multiple test failures and the bug log reports each (failed) test consecutively. Instead the Diderot compiler could generate error messages with labels. *DATm* could read the labels and use it categorize the test results. The bug log would then be organized by errors and produce a more informative report.

Types of Testing It has been valuable to have an exhaustive approach to generating test programs, since testing has exposed interesting and 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 (because of extensive language changes), we could possibly create a family of test programs and do some variation of differential testing [56]. In the future, it would be interesting to evaluate the effectiveness of different testing approaches on the Diderot compiler [14].

Parallel Testing The time it takes to run a large number of tests is a limiting factor in the usefulness of *DATm*. It takes seconds per test (depending on the test's complexity and input argument types), 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 multiple 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. In (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 a generic number of arguments with a mix of tensors and field components.

There are 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 possible approaches to this problem. For one, we could redesign EIN with sharing in mind. Second, we could make sharing visible during normalization. Sharing values would be effective when applying rewrites that clearly replicate expressions but it also 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 not visualized FEM data with Diderot programs that use higher-order code. Partly, because the EIN IR (discussed in Chapter 2) cannot yet represent FEM data. In the future, we look forward to being able to apply more

complicated visualization programs to FEM fields.

The callback that Firedrake offers, to evaluate a field at a point, is also limited. As far as we know, Firedrake does not offer a call back to take third or fourth derivatives of fields (which is necessary in some visualization programs as shown in Chapter 7). Additionally, the call back can only evaluate some fields. There is no callback supported for some niche problems such as extruded meshes, manifolds, and mixed element meshes. These type of structures could possibly require more syntax consideration on the Diderot side as well.

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. The change in coordinates from a reference element to one being computed involves the calculation of the Jacobian matrix, its determinant, and its inverse.

These operations can be similar to previous calls, leading to redundant and expensive computations. Diderot does not know how to evaluate a FEM field at a point or have the syntax to find the right cell, but it can represent tensor computations. If we are able to move some of the computations into Diderot then Diderot can catch these redundant computations and the entire process is less expensive.

Describing a FEM field The current Diderot syntax to define a field is written as follows.

```
fem#k(d)[σ] f;
field#k(d)[σ]F = toField(f);
```

The syntax could be more informative for someone reading the Diderot code. Instead of using the phrase "**fem**" it might be more descriptive to provide different syntax to define a field created by FEM data. We could create two new datatypes; a mesh (mesh) and a field pointer (fptr). A mesh could note the reference element (elem) and polynomial order (o).

```
input string elem;
input int o;
```

```
input mesh(o) m = mesh(elem, o);
```

The field pointer (fptr) is a pointer to the field data.

```
input fptr#p_q(d)[σ] f;
```

The syntax could reflect implementation restrictions. For example, the continuity of the field created (noted in code as p) might be distinct from the number of derivatives available (noted in code as q) by callback mechanisms. The field will then be defined by composing the mesh and field pointer.

```
field#q(d)[σ]F = toField(m, f)
```

The syntax is more informative to someone reading the code than the current syntax.

9.1.4 Writing directly in EIN IR

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. It is worth noting that the suggested syntax is for writing in the EIN IR and not to support traditional Einstein Index notation on the surface.

Translation The translation process would convert the surface level notation to the EIN IR. In this section we will use notation $t[\alpha_0, \alpha_1, \dots]$ to indicate the EIN term for tensor variable t has indices initialized with α. We use the term "$Ein*$" to indicate that the syntax is now in EIN notation. We use "EinAdd" and "EinMult" to mean add and multiply, respectively. Consider, adding two permutations of the same tensor:

```
tensor[2,3,3] t;
tensor[2,3,3] out = EinAdd(t[a,b,c], t[a,c,b]);
```

A direct mapping of the variable t in term t$[a, b, c]$ can create EIN expression T_{ijk} and a permutation of the variable in t$[a, c, b]$ creates term T_{ikj} in the EIN operator:

$$out = \lambda(T)\langle T_{ijk} + T_{ikj} \rangle_{i:a,j:b,k:c}(t)$$

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 explicit summation symbol (Σ) instead of using an implicit summation convention. Consider writing several product operations between two tensors (T and M) with code

```
tensor[d1, d2, d2] T;
tensor[d2, d2] M;
//inner product
tensor[d1,d2,d2] inner = EinMult(T[a,b,d], M[d,c]);
//double dot product
tensor[d1] double = Σbc(EinMult( T[a,b,c], M[b,c]));
//augmented product
tensor[d1] augmented = Σc(EinMult(T[a,b,c], M[c,b]));
```

Like the EIN IR, we use an explicit summation symbol (Σ_c) to clearly mark variable bindings. This type of translation should be clear to the user and compiler.

$$\begin{aligned}
\text{inner} &= \lambda(T, M)\langle \textstyle\sum_l T_{ijl} * M_{lk} \rangle_{i:d_1,j:d_2,k:d_2}(\mathrm{T}, \mathrm{M}) \\
\text{double} &= \lambda(T, M)\left\langle \textstyle\sum_{jk} T_{ijk} * M_{jk} \right\rangle_{i:d_1}(\mathrm{T}, \mathrm{M}) \\
\text{augmented} &= \lambda(T, M)\left\langle \textstyle\sum_{jk} T_{ijk} * M_{kj} \right\rangle_{i:d_1}(\mathrm{T}, \mathrm{M})
\end{aligned}$$

Discussion An existing operator (slice), and another (currently being developed) operator (cons) could possibly represent these computations by projecting components and then composing them together. However the surface level syntax and internal representations (of slice and cons) would likely be much larger than that created by writing directly in EIN. Writing directly in EIN can then offer a more readable and concise way to refer to tensor and field components.

Like the EIN IR, we use an explicit summation symbol (Σ_c) on the surface language to clearly mark variable index (c) boundaries. Alternatively, we could rely on the translation to recognize the redundant indices in the product term and create a summation operator. However, that could be expensive and confusing to the user.

9.1.5 Indicating covariant and contravariant indices

There is a distinction between 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 contra-variant indices with an upper and lower index M^{μ}_{ν}.

Diderot's coordinate system assumes an orthonormal basis and so the distinction between indices does not matter. In the future, supporting co/contra-variant indices in EIN may be useful for future language features. We may wish to make that distinction in the surface language.

tensor [μ ; ν] M;

There is currently no support to distinguish between covariant and contra-variant 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

[1] K. Ahlander. Einstein summation for multi-dimensional arrays. *Computers and Mathematics with Applications*, 44:1007–1017, October – November 2002.

[2] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. In *Visualization Handbook*, pages 717–731. Academic Press, Inc., Orlando, FL, USA, 2005.

[3] Martin S. Alnaes, Anders Logg, Kristian B. Oelgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40, February 2014.

[4] A.Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. FFC: the fenics form compiler. In A. Logg, K.-A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 11, pages 227–238. Springer, 2012.

[5] Ivo Babuska and J. Tinsley Oden. Verification and validation in computational engineering and science: Basic concepts. *Computer Methods in Applied Mechanics and Engineering*, 193(36-38):4057–4066, 9 2004.

[6] H. P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, New York, NY, revised edition, 1984.

[7] A. Barr. The Einstein summation notation, introduction to cartesian tensors and extensions to the notation. Draft paper; available at http://zeus.phys.uconn.edu/m̃cintyre/workfiles/Papers/Einstien-Summation-Notation.pdf.

[8] P J Basser and C Pierpaoli. Microstructural and physiological features of tissues elucidated by quantitative-diffusion-tensor MRI. *Journal of Magnetic Resonance, Series B*, 111(3):209–219, 1996.

[9] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, pages 125–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] Susanne C Brenner and Ridgway Scott. *The mathematical theory of finite element methods*, volume 15. Springer, 2007.

[11] K. J. Brown, A. K. Sujeeth, Hyouk J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Parallel Architectures and Compilation Techniques*, pages 89–100. IEEE, October 2011.

[12] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–714, 1986.

[13] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 45(10):835–847, October 2010.

[14] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, May 14-22, 2016*, pages 180–190, 2016.

[15] Charisee Chiw. Ein notation in diderot. Master's thesis, University of Chicago, April 2014. available from http://diderot-language.cs.uchicago.edu/papers/chiw-masters.pdf.

[16] Charisee Chiw, Gordon L Kindlman, and John Reppy. EIN: An intermediate representation for compiling tensor calculus. In *Proceedings of the 19th Workshop on Compilers for Parallel Computing*, July 2016.

[17] Charisee Chiw, Gordon Kindlmann, and John Reppy. Datm: Diderots automated testing model. *IEEE/ACM 39th International Conference on Software Engineering (IEEE/ACM 12th International Workshop on Automation of Software Test)*, page (to appear), 2017.

[18] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel dsl for image analysis and visualization. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 47(6):111–120, June 2012.

[19] Charisee Chiw and John Reppy. Properties of normalization for a math based intermediate representation. arXiv:1705.08801, 2017.

[20] H. Choi, W. Choi, T.M. Quan, D.G.C. Hildebrand, H. Pfister, and W. Jeong. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2407–2416, December 2014.

[21] Tai L. Chow. *Mathematical Methods for Physicists : A Concise Introduction*. Cambridge University Press, Cambridge, 2000.

[22] Jan Christiansen and Sebastian Fischer. Easycheck: Test data for free. In *Proceedings of the 9th International Conference on Functional and Logic Programming*, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[24] D. Cociorva, J. W. Wilkins, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Loop optimization for a class of memory-constrained computations. In *Proceedings*

of the 15th International Conference on Supercomputing, pages 103–113, New York, NY, USA, 2001. ACM.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[26] D. Degani, Y. Levy, and A. Seginer. Graphical visualization of vortical flows by means of helicity. *American Institute of Aeronautics and Astronautics Journal*, 28:1347–1352, August 1990.

[27] Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing, International Conference on Software Engineering 2016, Austin, Texas, USA, May 16, 2016*, pages 44–47, 2016.

[28] Kees Dullemond and Kasper Peeters. *Introduction to Tensor Calculus*. Kees Dullemond and Kasper Peeters, 1991.

[29] Todd Dupont, Johan Hoffman, Claus Johnson, Robert C Kirby, Mats G Larson, Anders Logg, and R Scott. *The FEniCS project*. Chalmers Finite Element Centre, Chalmers University of Technology, 2003.

[30] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 61–72, New York, NY, USA, 2012. ACM.

[31] David Eberly, Robert Gardner, Bryan Morse, Stephen Pizer, and Christine Scharlach. Ridges for image analysis. *Journal of Mathematical Imaging and Vision*, 4(4):353–373, 1994.

[32] Albert Einstein. The foundation of the general theory of relativity. In A. J. Kox, Martin J. Klein, and Robert Schulmann, editors, *The Collected papers of Albert Einstein*, volume 6, pages 146–200. Princeton University Press, Princeton NJ, 1996.

[33] Tiago Etiene, Carlos Scheidegger, L Gustavo Nonato, Robert M Kirby, and Cláudio T Silva. Verifiable visualization for isosurface extraction. *Visualization and Computer Graphics*, 15(6):1227–1234, 2009.

[34] R. Feldt, S. Poulding, D. Clark, and S. Yoo. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. *ArXiv e-prints*, June 2015.

[35] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. *Operator Language: A Program Generation Framework for Fast Kernels*, pages 385–409. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[36] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 315–326, New York, NY, USA, 2005. ACM.

[37] Xiaoyang Gao, Swarup Kumar Sahoo, Chi-Chung Lam, J. Ramanujam, Qingda Lu, Gerald Baumgartner, and P. Sadayappan. Performance modeling and optimization of parallel out-of-core tensor contractions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 266–276, New York, NY, USA, 2005. ACM.

[38] Miloš Hašan, John Wolfgang, George Chen, and Hanspeter Pfister. Shadie: A domain-specific language for volume visualization. Draft paper; available at http://miloshasan.net/Shadie/shadie.pdf, 2010.

[39] Gerhard A. Holzapfel. *Nonlinear Solid Mechanics*. John Wiley and Sons, West Sussex, England, 2000.

[40] Luis Ibanez and Will Schroeder. *The ITK Software Guide*. Kitware Inc., 2005.

[41] Richard L. Van Metter Jacob Beutel, Harold Kundel. *Medical image processing and analysis.* SPIE Press., 2000.

[42] Christopher Johnson and Charles Hansen. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004.

[43] Gordon Kindlmann. Diderot-language/examples. Available from https://github.com/Diderot-Language/examples.

[44] Gordon Kindlmann, Charisee Chiw, Nicholas Seltzer, Lamont Samuels, and John Reppy. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):867–876, January 2016.

[45] Gordon Kindlmann, Ross Whitaker, Tolga Tasdizen, and Torsten Moller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of the 14th IEEE Visualization 2003*, pages 67–, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Gordon L. Kindlmann and Carlos Eduardo Scheidegger. An algebraic process for visualization design. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2181–2190, 2014.

[47] Robert C Kirby. Algorithm 839: Fiat, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software*, 30(4):502–516, 2004.

[48] Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. Optimizing the evaluation of finite element matrices. *SIAM Journal on Scientific Computing*, 27(3):741–758, October 2005.

[49] Sandhya Krishnan, Sriram Krishnamoorthy, Gerald Baumgartner, Chi-Chung Lam, J. Ramanujam, P. Sadayappan, and Venkatesh Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. *Journal of Parallel and Distributed Computing*, 66(5):659–673, 2006.

[50] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2014.

[51] R. Lengagne, P. Fua, and O. Monga. Using crest lines to guide surface reconstruction from stereo. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 1, pages 9–13 vol.1, Aug 1996.

[52] Christian Lindig. Random testing of C calling conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, pages 3–12, New York, NY, USA, 2005. ACM.

[53] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Springer Publishing Company, Incorporated, 2012.

[54] Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. Cross-loop optimization of arithmetic intensity for finite element local assembly. *Architecture and Code Optimization*, 11(4):57:1–57:25, 2014.

[55] Patrick McCormick, Jeff Inman, James Ahrens, Jamaludin Mohd-Yusof, Greg Roth, and Sharen Cummins. Scout: A data-parallel programming language for graphics processors. *Journal of Parallel Computing*, 33:648–662, November 2007.

[56] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[57] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 125–134, 2013.

[58] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 91–97, New York, NY, USA, 2011. ACM.

[59] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, 2015. ACM.

[60] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[61] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):24:1–24:27, December 2016.

[62] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. Pyop2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis, 2012 SC Companion:*, pages 1116–1123. IEEE, 2012.

[63] Daniel Ratiu and Markus Voelter. Automated testing of DSL implementations: Experiences from building mbeddr. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 15–21, New York, NY, USA, 2016. ACM.

[64] Peter Rautek, Stefan Bruckner, Meister Eduard Gr"oller, and Markus Hadwiger. Vislang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2388–2396, 2014.

[65] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware, Inc., Clifton Park, New York, 3rd edition, 2004.

[66] James. Simmonds. *A Brief on Tensor Analysis*. Springer-Verlag, New York, 1982.

[67] Ivan Stephen Sokolinkoff. *Tensor Analysis*. John Wiley and Sons, New York, 1960.

[68] Sympy is a python library. *SymPy website at http://www.sympy.org/en/index.html*.

[69] Schultz T, Theisel H, and Seidel HP. Crease surfaces: from theory to extraction and application to diffusion tensor mri. *IEEE Transactions on Visualization and Computer Graphics*, 16:109–119, 2010.

[70] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *17th Asia Pacific Software Engineering Conference, Sydney, Australia, November 30 - December 3, 2010*, pages 270–279, 2010.

[71] Teem Library. *Teem website at http://teem.sf.net*.

[72] Cludio T Silva Tiago Etiene, Robert M Kirby. *An Introduction to Verification of Visualization Techniques*, volume 7. Morgan and Claypool Publishers, Clifton Park, New York, 3rd edition, December 2015.

[73] Hui Wu, Jeff Gray, and Marjan Mernik. *Unit Testing for Domain-Specific Languages*, pages 125–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 46(6):283–294, June 2011.

APPENDIX A

PROOFS

A.1 Type Preservation Proof

The following is a proof for Theorem 4.1.1

Given a derivation d of the form $e \xrightarrow[rule]{} e'$ 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

Case R1.$(e_1 \odot_n e_2)@x \xrightarrow[rule]{} (e_1@x) \odot_n (e_2@x)$

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 $\odot_n = /$.

if $\odot_n = *$

Find $\Gamma, \sigma \vdash ((e_1 * e_2)@x)$

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

The LHS has the following type.

$\Gamma, \sigma \vdash (e_1 \odot_n e_2)@x : (\sigma)\mathcal{T}$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash (e_1@x) \odot_n (e_2@x) : (\sigma)\mathcal{T}$.

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d \qquad \Gamma, \sigma \vdash e_2 : (\sigma)\mathcal{F}^d[\text{TYINV}_{11}]}{\Gamma, \sigma \vdash e1 * e2 : (\sigma)\mathcal{F}^d[\text{TYINV}_7]} \qquad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash (e1 * e2)@x : (\sigma)\mathcal{T}}$$

From that we can make the RHS derivations.

Find $\Gamma, \sigma \vdash ((e_1@x) * (e_2@x))$

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 [TYJUD$_7$],

and $\Gamma, \sigma \vdash e_1@x * e_2@x : (\sigma)\mathcal{T}$ by [TYJUD$_{11}$]

T(R1 for $\odot_n = *$)

if $\odot_n = /$

Find $\Gamma, \sigma \vdash ((\frac{e_1}{e_2})@x)$

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

$\Gamma, \sigma \vdash (e_1 \odot_n e_2)@x : (\sigma)\mathcal{T}$ ([TYINV$_7$])

Find $\Gamma, \sigma \vdash (e_1$ and $e_2)$

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d, \qquad \Gamma, \sigma \vdash e_2 : ()\mathcal{F}^d[\text{TYINV}_{12}]}{\Gamma, \sigma \vdash (\dfrac{e1}{e2}) : (\sigma)\mathcal{F}^d[\text{TYINV}_7]} \qquad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash (\dfrac{e1}{e2})@x : (\sigma)\mathcal{T}}$$

Find $\Gamma, \sigma \vdash (\frac{(e_1@x)}{(e_2@x)})$

Given that $\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d, \Gamma, \sigma \vdash e_2 : ()\mathcal{F}^d$

then $\Gamma, \sigma \vdash e_1@x : (\sigma)\mathcal{T}$ by [TYJUD$_7$],

$\Gamma, \sigma \vdash e_2@x : ()\mathcal{T}$ by [TYJUD$_7$],

and $\Gamma, \sigma \vdash \frac{e_1@x}{e_2@x} : (\sigma)\mathcal{T}$ by [TYJUD$_{12}$].

T(R1 for $\odot_n = /$)

`T(R1) OK`

Case R2.$(e_0 \odot_2 e_1)@x \xrightarrow[rule]{} (e_0@x) \odot_2 (e_1@x)$

$\odot_2 = + \mid -$

Find $\Gamma, \sigma \vdash ((e_1 \odot_2 e_2)@x)$

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

The LHS has the following type.

$\Gamma, \sigma \vdash (e_0 \odot_2 e_1)@x : (\sigma)\mathcal{T}$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash (e_0@x) \odot_2 (e_1@x)\!:\!(\sigma)\mathcal{T}.$

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma \vdash e_1, e_2 : (\sigma)\mathcal{F}^d[\text{TYINV}_{10}]}{\Gamma, \sigma \vdash e_1 \odot_2 e2 : (\sigma)\mathcal{F}^d[\text{TYINV}_7]} \qquad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash (e1 \odot_2 e2)@x : (\sigma)\mathcal{T}}$$

From that we can make the RHS derivations.

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 $[\text{TYJUD}_7]$

and $\Gamma, \sigma \vdash e_1@x \odot_2 e_2@x : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_{10}]$

Case R3.$(\odot_1 e_1)@x \underset{rule}{\longrightarrow} \odot_1 (e_1@x)$

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 $\odot_1 = M(.)$.

if $\odot_1 = -$,

Find $\Gamma, \sigma \vdash ((-e_1)@x)$

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

The LHS has the following type.

$\Gamma, \sigma \vdash (\odot_1 e_1)@x :(\sigma)\mathcal{T}$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \odot_1(e_1@x)\!:\!(\sigma)\mathcal{T}.$

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d[\text{TYINV}_{10}]}{\Gamma, \sigma \vdash -e_1 : (\sigma)\mathcal{F}^d[\text{TYINV}_7]} \qquad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash (-e_1)@x : (\sigma)\mathcal{T}}$$

From that we can make the RHS derivations.

Find $\Gamma, \sigma \vdash (-(e_1@x))$

Given that $\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d$

then $\Gamma, \sigma \vdash e_1@x : (\sigma)\mathcal{T}$ by [TYJUD$_7$]

and $\Gamma, \sigma \vdash -e_1@x : (\sigma)\mathcal{T}$ by [TYJUD$_{10}$]

T(R3 for $\odot_1 = -$)

if $\odot_1 = M(e_1)$

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.

The LHS has the following type.

$\Gamma, \sigma \vdash (\odot_1 e_1)@x :(\sigma)\mathcal{T}$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \odot_1(e_1@x):(\sigma)\mathcal{T}$.

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d([\text{TYINV}_9])}{\Gamma, \sigma \vdash M(e_1) : (\sigma)\mathcal{F}^d[\text{TYINV}_7]} \qquad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash M(e_1)@x : (\sigma)\mathcal{T}}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d$

then $\Gamma, \sigma \vdash e_1@x : (\sigma)\mathcal{T}$ by [TYJUD$_7$]

and $\Gamma, \sigma \vdash M(e_1@x) : (\sigma)\mathcal{T}$ by [TYJUD$_9$]

T(R3 for $\odot_1 = M$)

`T(R3) OK`

Case R4.$(\sum_{i=1}^{n} e_1)@x \xrightarrow[rule]{} \sum_{i=1}^{n}(e_1@x)$. Included in the earlier prose.

Case R5.$(\chi)@x \xrightarrow[rule]{} \chi$

We will do a case analysis on the structure on the left-hand-side

where χ=$\{\mathbf{lift}_d(e_1)|\delta_{ij} \mid \mathcal{E}_\alpha\}$.

First we will prove T(d) for χ=$\mathbf{lift}_d(e_1)$ then χ=$\delta_{ij} \mid \mathcal{E}_\alpha$.

case $\chi = \mathbf{lift}_d(e_1)$

Find $\Gamma, \sigma \vdash ((\chi(e_1))@x)$

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

The LHS has the following type.

$\Gamma, \sigma \vdash (\chi)@x :(\sigma)\mathcal{T}$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \chi{:}(\sigma)\mathcal{T}.$

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d([\text{TYINV}_8])}{\Gamma, \sigma \vdash (\mathbf{lift}_d(e_1)) : (\sigma)\mathcal{F}^d[\text{TYINV}_7]} \qquad \Gamma, \sigma \vdash x : \mathbf{Ten}[d]}{\Gamma, \sigma \vdash (\mathbf{lift}_d(e_1))@x : (\sigma)\mathcal{T}}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : (\sigma)\mathcal{F}^d$

then $\Gamma, \sigma \vdash e_1@x : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_7]$

and $\Gamma, \sigma \vdash \mathbf{lift}_d(e_1@x) : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_8]$

T(R5 where $\chi = \mathbf{lift}_d(e_1)$)

For the case $\chi = \delta_{ij} \mid \mathcal{E}_\alpha$

Given that $\Gamma, \sigma \vdash \chi : \tau$ then $\Gamma, \sigma \vdash \chi@x : \tau$ by $[\text{TYJUD}_7]$

T(R5 where $\chi = \delta_{ij} \mid \mathcal{E}_\alpha$)

`T( R5) OK`

Case R6. $\frac{\partial}{\partial x_i} \diamond (e_1 * e_2) \xrightarrow[rule]{} e_1(\frac{\partial}{\partial x_i} \diamond e_2) + e_2(\frac{\partial}{\partial x_i} \diamond e_1)$. Included in the earlier prose.

Case R7. $\frac{\partial}{\partial x_i} \diamond (\frac{e_1}{e_2}) \xrightarrow[rule]{} \frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2}$. Included in the earlier prose.

Case R8. $\frac{\partial}{\partial x_i} \diamond (\sqrt{e_1}) \xrightarrow[rule]{} \mathbf{lift}_d(1/2) * \frac{\frac{\partial}{\partial x_i} \diamond e_1}{\sqrt{e_1}}$

Find $\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond (\sqrt{e_1}))$

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 $[\text{TYJUD}_4]$

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (\sqrt{e_1}) : (i)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \mathbf{lift}_d(1/2) * \frac{\frac{\partial}{\partial x_i} \diamond e}{\sqrt{e_1}} : (i)\mathcal{F}^d.$

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d [\text{TYINV}_9]}{\Gamma, \sigma \vdash \sqrt{e_1} : ()\mathcal{F}^d [\text{TYINV}_4]}}{\Gamma, \sigma \vdash \dfrac{\partial}{\partial x_i} \diamond (\sqrt{e_1}) : (\sigma)\mathcal{F}^d \text{ and } \sigma = \{i : d\} (\text{Claim})}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$

then $\Gamma, \sigma[i \mapsto (1, d)] \vdash \frac{\partial}{\partial x_{i:d}} \diamond e_1 : (i)\mathcal{F}^d([\text{TYJUD}_4])$

and $\Gamma, \sigma \vdash \sqrt{e_1} : ()\mathcal{F}^d([\text{TYJUD}_9])$

Additionally, $\Gamma, \sigma \vdash \mathbf{lift}_d(-) : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_8]$

Given that $\Gamma, \sigma \vdash \sqrt{e_1} : ()\mathcal{F}^d$ and $\Gamma, \sigma[i \mapsto (1, d)] \vdash \frac{\partial}{\partial x_{i:d}} \diamond e_1 : (i)\mathcal{F}^d$

then $\Gamma, \sigma[i \mapsto (1, d)] \vdash \frac{\frac{\partial}{\partial x_{i:d}} \diamond e_1}{\sqrt{e_1}} : (i)\mathcal{F}^d$ by $[\text{TYJUD}_{12}]$

and $\Gamma, \sigma[i \mapsto (1, d)] \vdash \mathbf{lift}_d(1/2) * \frac{\frac{\partial}{\partial x_i} \diamond e}{\sqrt{e_1}} : (i)\mathcal{F}^d$ by $[\text{TYJUD}_{11}]$

Case R9. $\frac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1)) \xrightarrow[rule]{} (-\mathbf{sine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Find $\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1)))$

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

and the $\mathbf{cosine}(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \mathbf{cosine}(e_1) : ()\mathcal{F}^d$ then $\Gamma_i \vdash \nabla_i \diamond (\mathbf{cosine}(e_1)) : (i)\mathcal{F}^d$ by $[\text{TYJUD}_4]$

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1)) : (i)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash (-\mathbf{sine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d.$

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d[\text{TYINV}_9]}{\Gamma, \sigma \vdash \mathbf{cosine}(e_1) : ()\mathcal{F}^d}}{\Gamma, \sigma[i \mapsto (1,d)] \vdash \dfrac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1)) : (i)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$

then $\Gamma, \sigma[i \mapsto (1,d)] \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (i)\mathcal{F}^d$ by [TYJUD$_4$],

$\Gamma, \sigma \vdash \mathbf{sine}(e_1) : ()\mathcal{F}^d$ by [TYJUD$_9$],

$\Gamma, \sigma \vdash -\mathbf{sine}(e_1) : ()\mathcal{F}^d$ by [TYJUD$_{10}$],

and $\Gamma, \sigma[i \mapsto (1,d)] \vdash (-\mathbf{sine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d$ by [TYJUD$_{11}$]

`T(R9) OK`

Case R10. $\frac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1)) \xrightarrow[rule]{} (\mathbf{cosine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$. Included in the earlier prose.

`T(R10) OK`

Case R11. $\frac{\partial}{\partial x_i} \diamond (\mathbf{tangent}(e_1)) \xrightarrow[rule]{} \frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{cosine}(e_1) * \mathbf{cosine}(e_1)}$

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

and the $\mathbf{tangent}(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \mathbf{tangent}(e_1) : ()\mathcal{F}^d$ then $\Gamma_i \vdash \nabla_i \diamond (\mathbf{tangent}(e_1)) : (i)\mathcal{F}^d$ by [TYJUD$_4$]

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (\mathbf{tangent}(e_1)) : (i)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{cosine}(e_1) * \mathbf{cosine}(e_1)} : (i)\mathcal{F}^d$.

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d[\text{TYINV}_9]}{\Gamma, \sigma \vdash \mathbf{tangent}(e_1) : ()\mathcal{F}^d}}{\Gamma, \sigma[i \mapsto (1,d)] \vdash \dfrac{\partial}{\partial x_i} \diamond (\mathbf{tangent}(e_1)) : (i)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$

then $\Gamma, \sigma[i \mapsto (1,d)] \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (i)\mathcal{F}^d$ by [TYJUD$_4$],

$\Gamma, \sigma \vdash \mathbf{cosine}(e_1) * \mathbf{cosine}(e_1) : ()\mathcal{F}^d$ by [TYJUD$_9$], [TYJUD$_{11}$],

and $\Gamma, \sigma \vdash \frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{cosine}(e_1) * \mathbf{cosine}(e_1)} : ()\mathcal{F}^d$ by [TYJUD$_{12}$]

`T( R11) OK`

Case R12. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arccosine}(e_1)) \xrightarrow[rule]{} (\frac{-\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Similar approach to R13 `T( R12) OK`

Case R13. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1)) \xrightarrow[rule]{} (\frac{\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Find $\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1)))$

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

and the $\mathbf{arcsine}(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \mathbf{arcsine}(e_1) : ()\mathcal{F}^d$ then $\Gamma_i \vdash \nabla_i \diamond (\mathbf{arcsine}(e_1)) : (i)\mathcal{F}^d$ by [TYJUD$_4$]

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1)) : (i)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash (\frac{\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d$.

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d([\text{TYINV}_9])}{\Gamma, \sigma \vdash \mathbf{arcsine}(e_1) : ()\mathcal{F}^d}}{\Gamma, \sigma[i \mapsto (1, d)] \vdash \dfrac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1)) : (i)\mathcal{F}^d}$$

Since $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$ then $\Gamma, \sigma[i \mapsto (1, d)] \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (i)\mathcal{F}^d$ by [TYJUD$_4$]

Find $\Gamma, \sigma \vdash (\mathbf{lift}_d(1))$

$\Gamma, \sigma \vdash \mathbf{lift}_d(1) : (\sigma)\mathcal{F}^d([\text{TYJUD}_8])$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$

then $\Gamma, \sigma \vdash e_1 * e_1 : ()\mathcal{F}^d$ by [TYJUD$_{11}$],

$\Gamma, \sigma \vdash \mathbf{lift}_d(1) - (e_1 * e_1) : ()\mathcal{F}^d$ by [TYJUD$_{10}$],

$\Gamma, \sigma \vdash \sqrt{\mathbf{lift}_d(1) - (e_1 * e_1)} : ()\mathcal{F}^d$ by [TYJUD$_9$],

$\Gamma, \sigma \vdash \frac{\mathbf{lift}_d(1)}{\sqrt{\mathbf{lift}_d(1) - (e_1 * e_1)}} : ()\mathcal{F}^d$ by [TYJUD$_{12}$],

and $\Gamma, \sigma[i \mapsto (1,d)] \vdash (\frac{\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1)-(e*e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1) : ()\mathcal{F}^d$ by [TYJUD$_{11}$]

`T( R13) OK`

Case R14. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arctangent}(e_1)) \xrightarrow[rule]{} \frac{\mathbf{lift}_d(1)}{\mathbf{lift}_d(1)+(e_1*e_1)} * (\frac{\partial}{\partial x_i} \diamond e_1)$

Similar approach to R13 `T( R14) OK`

Case R15. $\frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e_1)) \xrightarrow[rule]{} \mathbf{exp}(e_1) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Find $\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e_1)))$

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

and the $\exp(e_1)$ term results in a scalar.

Claim: $\Gamma \vdash \exp(e_1) : ()\mathcal{F}^d$ then $\Gamma_i \vdash \nabla_i \diamond (\exp(e_1)) : (i)\mathcal{F}^d$ by [TYJUD$_4$]

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e_1)) : (i)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \mathbf{exp}(e_1) * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d$.

The type derivation for the LHS is the following structure.

$$\frac{\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d([\text{TYINV}_9])}{\Gamma, \sigma \vdash \mathbf{exp}(e_1) : ()\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$

then $\Gamma, \sigma[i \mapsto (1,d)] \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (i)\mathcal{F}^d$ by [TYJUD$_4$],

$\Gamma, \sigma \vdash \mathbf{exp}(e_1) : ()\mathcal{F}^d$ by [TYJUD$_9$],

and $\Gamma, \sigma[i \mapsto (1,d)] \vdash \mathbf{exp}(e_1) * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d$ by [TYJUD$_{11}$]

`T(R15) OK`

Case R16. $\frac{\partial}{\partial x_i} \diamond (e_1^n) \xrightarrow[rule]{} \mathbf{lift}_d(n) * e_1^{n-1} * (\frac{\partial}{\partial x_i} \diamond e_1)$

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 $\Gamma_i \vdash \nabla_i \diamond (e_1^n) : (i)\mathcal{F}^d$ by [TYJUD$_4$]

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (e_1^n) : (i)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \mathbf{lift}_d(n) * e_1^{n-1} * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d.$

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d, \Gamma, \sigma \vdash n : ()\mathcal{T} \text{ and } \sigma = \{i : d\}([\text{TYINV}_9])}{\Gamma, \sigma \setminus i \vdash (e^n) : (\sigma \setminus i)\mathcal{F}^d [\text{TYINV}_4]}}{\Gamma, \sigma \vdash \dfrac{\partial}{\partial x_{i:d}} \diamond (e^n) : (i)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d$ then $\Gamma, \sigma[i \mapsto (1, d)] \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (i)\mathcal{F}^d$ by $[\text{TYJUD}_4]$.

Given that $\Gamma, \sigma \vdash e_1 : ()\mathcal{F}^d, \Gamma, \sigma \vdash n : ()\mathcal{T}$

then $\Gamma, \sigma \vdash \mathbf{lift}_d(n) : ()\mathcal{F}^d$ by $[\text{TYJUD}_8]$ and $\Gamma, \sigma \vdash e^{n-1} : ()\mathcal{F}^d$ by $[\text{TYJUD}_9]$.

Given that $\Gamma, \sigma \vdash e^{n-1} : ()\mathcal{F}^d$ and $\Gamma, \sigma[i \mapsto (1, d)] \vdash \frac{\partial}{\partial x_i} \diamond e_1 : (i)\mathcal{F}^d$

then $\Gamma, \sigma[i \mapsto (1, d)] \vdash \mathbf{lift}_d(n) * e_1^{n-1} * (\frac{\partial}{\partial x_i} \diamond e_1) : (i)\mathcal{F}^d$ by $[\text{TYJUD}_{11}]$.

`T(R16) OK`

Case R17. $\frac{\partial}{\partial x_i} \diamond (e_1 \odot e_2) \xrightarrow[rule]{} (\frac{\partial}{\partial x_i} \diamond e_1) \odot (\frac{\partial}{\partial x_i} \diamond e_2)$

Find $\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond (e_1 \odot e_2))$

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

Given the subterm: $\Gamma, \sigma/\ i \vdash e_1 \odot e_2 : (\sigma/\ i)\mathcal{F}^d$

then by $[\text{TYJUD}_4]$ we know it's derivative $\Gamma, \sigma \vdash \nabla_i \diamond (e_1 \odot e_2) : (\sigma)\mathcal{F}^d$

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (e_1 \odot e_2) : (\sigma)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond e_1) \odot (\frac{\partial}{\partial x_i} \diamond e_2) : (\sigma)\mathcal{F}^d.$

The type derivation for the LHS is the following structure.

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

$$\frac{\dfrac{\Gamma, \sigma \setminus i \vdash e_1, e_2 : (\sigma \setminus i)\mathcal{F}^d [\text{TYINV}_{10}]}{\Gamma, \sigma \setminus i \vdash e_1 \odot e_2 : (\sigma \setminus i)\mathcal{F}^d [\text{TYINV}_4]}}{\Gamma, \sigma \vdash \dfrac{\partial}{\partial x_i} \diamond (e_1 \odot e_2) : (\sigma)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1, e_2 : (\sigma \setminus i)\mathcal{F}^d$

then $\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (e_1) : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_4]$

and $\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond e_1) \odot (\frac{\partial}{\partial x_i} \diamond e_2) : (\sigma)\mathcal{F}^d$ by $[\text{TYINV}_{10}]$.

`T( R17) OK`

Case R18. $\frac{\partial}{\partial x_i} \diamond (-e_1) \xrightarrow[rule]{} -(\frac{\partial}{\partial x_i} \diamond e_1)$

Find $\Gamma, \sigma \vdash (\frac{\partial}{\partial x_i} \diamond (-e_1))$

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

Given the subterm: $\Gamma, \sigma/\ i \vdash -e_1 : (\sigma/\ i)\mathcal{F}^d$

then by $[\text{TYJUD}_4]$ we know it's derivative $\Gamma, \sigma \vdash \nabla_i \diamond (-e_1) : (\sigma)\mathcal{F}^d$

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (-e_1) :(\sigma)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash -(\frac{\partial}{\partial x_i} \diamond e_1):(\sigma)\mathcal{F}^d$.

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \setminus i \vdash e_1 : (\sigma \setminus i)\mathcal{F}^d [\text{TYINV}_{10}]}{\Gamma, \sigma \setminus i \vdash -e_1 : (\sigma \setminus i)\mathcal{F}^d [\text{TYINV}_4]}}{\Gamma, \sigma \vdash \dfrac{\partial}{\partial x_i} \diamond (-e_1) : (\sigma)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : (\sigma \setminus i)\mathcal{F}^d$

then $\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (e_1) : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_4]$

and $\Gamma, \sigma \vdash -(\frac{\partial}{\partial x_i} \diamond e_1) : (\sigma)\mathcal{F}^d$ by $[\text{TYINV}_{10}]$

`T( R18) OK`

Case R19. $\frac{\partial}{\partial x_i} \sum_{v=1}^{n} e_1 \xrightarrow[rule]{} \sum_{v=1}^{n} (\frac{\partial}{\partial x_i} e_1)$

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

Given the subterm: $\Gamma, \sigma/\ i \vdash \sum_{v=1}^{n} : (\sigma/\ i)\mathcal{F}^d$

then by [TYJUD$_4$] we know it's derivative $\Gamma, \sigma \vdash \nabla_i \diamond (\sum_{v=1}^{n}) : (\sigma)\mathcal{F}^d$

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \sum_{v=1}^{n} e_1 :(\sigma)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \sum_{v=1}^{n} (\frac{\partial}{\partial x_i} e_1):(\sigma)\mathcal{F}^d.$

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma \setminus i, v : n \vdash e_1 : (\sigma \setminus i, v : n)\mathcal{F}^d([\text{TYINV}_3])}{\Gamma, \sigma \setminus i \vdash (\sum_{v=1}^{n} e_1) : (\sigma \setminus i)\mathcal{F}^d[\text{TYINV}_4]}}{\Gamma, \sigma \vdash \frac{\partial}{\partial x_{i:d}} \diamond (\sum_{v=1}^{n} e_1) : (\sigma)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e_1 : (\sigma \setminus i, v : n)\mathcal{F}^d$

then $\Gamma, \sigma \vdash \frac{\partial}{\partial x_{i:d}} \diamond (e_1) : (\sigma, v : n)\mathcal{F}^d$ by [TYJUD$_4$]

and $\Gamma, \sigma \vdash \sum_{v=1}^{n} (\frac{\partial}{\partial x_{i:d}} \diamond (e_1)) : (\sigma)\mathcal{F}^d$ by ([TYJUD$_3$])

`T( R19) OK`

Case R20.$\frac{\partial}{\partial x_i}\chi \xrightarrow[rule]{} 0$

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

Given the subterm: $\Gamma, \sigma/\ i \vdash \nabla\chi : (\sigma/\ i)\mathcal{F}^d$

then by [TYJUD$_4$] we know it's derivative $\Gamma, \sigma \vdash \nabla_i \diamond (\nabla\chi) : (\sigma)\mathcal{F}^d$

Lastly,$\Gamma, \sigma \vdash 0 : (\sigma)\mathcal{F}^d$ by [TYJUD$_8$]. `T( R20) OK`

Case R21.$\frac{\partial}{\partial x_i} \diamond (V_\alpha \circledast H^\nu) \xrightarrow[rule]{} (V_\alpha \circledast h^{i\nu})$

Given$\Gamma, \sigma \vdash V_\alpha \circledast H^v : (\sigma/\ i)\mathcal{F}^d$ by [TYJUD$_2$]

then $\Gamma, \sigma \vdash \frac{\partial}{\partial x_i} \diamond (V_\alpha \circledast H^\nu) : (\sigma)\mathcal{F}^d$ by [TYJUD$_4$].

Lastly, $\Gamma, \sigma \vdash (V_\alpha \circledast H^{i\nu}) : (\sigma)\mathcal{F}^d$ by [TYJUD$_2$].

`T( R21) OK`

Case R22.$--e_1 \underset{rule}{\longrightarrow} e_1$

Find $\Gamma, \sigma \vdash (--e_1)$

Assign generic type $\Gamma, \sigma \vdash e_1 : \tau$

$$\dfrac{\dfrac{\Gamma, \sigma \vdash --e_1 : \tau[\text{TYINV}_{10}]}{\Gamma, \sigma \vdash -e_1 : \tau[\text{TYINV}_{10}]}}{\Gamma, \sigma \vdash e_1 : \tau}$$

From that we can make the RHS derivations.

`T(R22) OK`

Case R23.$-0 \underset{rule}{\longrightarrow} 0$

Find $\Gamma, \sigma \vdash (-0)$

Assign generic type $\Gamma, \sigma \vdash -0 : \tau$

Find $\Gamma, \sigma \vdash (0)$

The LHS has the following type.

$\Gamma, \sigma \vdash -0 :\tau$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash 0{:}\tau$.

The type derivation for the LHS is the following structure.

$$\dfrac{\Gamma, \sigma \vdash 0 : \tau[\text{TYINV}_{10}]}{\Gamma, \sigma \vdash -0 : \tau}$$

From that we can make the RHS derivations.

`T(R23) OK`

Case R24.$e_1 - 0 \underset{rule}{\longrightarrow} e_1$

Find $\Gamma, \sigma \vdash (e_1 - 0)$

Assign generic type $\Gamma, \sigma \vdash e_1 - 0 : \tau$

$\Gamma, \sigma \vdash e - 0 : (\sigma)\tau_0$

$\Gamma, \sigma \vdash 0 : (\sigma)\tau_0$ by $[\text{TYJUD}_1]$

T(R24)

`T(R24) OK`

Case R25. $-e_1 \underset{rule}{\longrightarrow} -e_1$

Similar approach to R24 `T( R25) OK`

Case R26. $\frac{0}{e_1} \underset{rule}{\longrightarrow} 0$

Similar approach to R24 `T( R26) OK`

Case R27. $\frac{\frac{e_1}{e_2}}{e_3} \underset{rule}{\longrightarrow} \frac{e_1}{e_2 e_3}$. Included in the earlier prose.

Case R28. $\frac{e_1}{\frac{e_2}{e_3}} \underset{rule}{\longrightarrow} \frac{e_1 e_3}{e_2}$

Similar approach to R27 `T( R28) OK`

Case R29. $\frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}} \underset{rule}{\longrightarrow} \frac{e_1 e_4}{e_2 e_3}$

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}} : (\sigma)\tau_0$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \frac{e_1 e_4}{e_2 e_3} : (\sigma)\tau_0$.

The type derivation for the LHS is the following structure.

$$\dfrac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma)\tau_0 \quad \Gamma, \sigma \vdash e_2 : ()\tau_0[\text{TYINV}_{12}]}{\Gamma, \sigma \vdash (\frac{e_1}{e_2}) : (\sigma)\tau_0} \quad \dfrac{\Gamma, \sigma \vdash e_3, e_4 : ()\tau_0[\text{TYINV}_{12}]}{\Gamma, \sigma \vdash (\frac{e_3}{e_4}) : ()\tau_0[\text{TYINV}_{12}]}}{\Gamma, \sigma \vdash \frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}} : (\sigma)\tau_0}$$

From that we can make the RHS derivations.

Find $\Gamma, \sigma \vdash (\frac{e_1 e_4}{e_2 e_3})$

Given that $\Gamma, \sigma \vdash e_1 : (\sigma)\tau_0$ and $\Gamma, \sigma \vdash e_2, e_3, e_4 : ()\tau_0$

then $\Gamma, \sigma \vdash e_1 * e_4 : (\sigma)\tau_0$ by [TYJUD_{11}],

$\Gamma, \sigma \vdash e_2 * e_3 : ()\tau_0$ by [TYJUD_{11}],

and $\Gamma, \sigma \vdash \frac{e_1 e_4}{e_2 e_3} : (\sigma)\tau_0$ by [TYJUD_{12}].

`T( R29) OK`

Case R30. $0 + e_1, e_1 + 0 \underset{rule}{\longrightarrow} e_1$

Similar approach to R24 `T( R30) OK`

Case R31. $0e, e0 \underset{rule}{\longrightarrow} 0$

Similar approach to R24 `T( R31) OK`

Case R32.$\sqrt{(e_1)} * \sqrt{(e_1)} \xrightarrow[rule]{} e_1$

Assign generic type $\Gamma, \sigma \vdash \sqrt{(e_1)} * \sqrt{(e_1)} : \tau$

Find $\Gamma, \sigma \vdash (e_1)$

$$\dfrac{\dfrac{\Gamma, \sigma \vdash e_1 : \tau([\text{TYINV}_9])}{\Gamma, \sigma \vdash \sqrt{e_1} : \tau[\text{TYINV}_{11}]}}{\Gamma, \sigma \vdash \sqrt{e_1} * \sqrt{e_1} : \tau}$$

`T(R32) OK`

Case R33.$\mathcal{E}_{ijk} \frac{\partial}{\partial x_{i,j}} \diamond e_1 \xrightarrow[rule]{} \mathbf{lift}_d(0)$

Similar approach to R34 `T(R33) OK`

Case R34.$\mathcal{E}_{ijk}(V_\alpha \circledast h^{jk}) \xrightarrow[rule]{} \mathbf{lift}_d(0)$

Given $\Gamma, \sigma \vdash V_\alpha \circledast h^{jk} : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_2]$

then $\Gamma, \sigma \vdash \epsilon_{ijk} V_\alpha \circledast h^{jk} : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_6]$.

Lastly, $\Gamma, \sigma \vdash \mathbf{lift}_d(0) : (\sigma)\mathcal{F}^d$ by $[\text{TYJUD}_8]$

`T(R34) OK`

Case R35.$\mathcal{E}_{ijk}\mathcal{E}_{ilm} \xrightarrow[rule]{} \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$

We know $\Gamma, \sigma \vdash \mathcal{E}_{ijk}\mathcal{E}_{ilm} : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_6]$.

Given $\Gamma, \sigma \vdash \delta_{jl}\delta_{km} : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_5]$

then $\Gamma, \sigma \vdash \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_{10}]$.

`T(R35) OK`

Case R36.$\delta_{ij} T_j \xrightarrow[rule]{} T_i$

Find $\Gamma, \sigma \vdash (\delta_{ij} T_j)$

Given $\Gamma, \sigma \vdash T_j : (\sigma)\mathcal{T}$ and $\sigma = \{j\}$ by $[\text{TYJUD}_1]$

then $\Gamma, \sigma \vdash \delta_{ij}(T_j) : (\sigma)\mathcal{T}$ by $[\text{TYJUD}_5]$

and $\sigma = \{i\}[\text{TYJUD}_5]$

Find $\Gamma, \sigma \vdash (T_i)$

$\Gamma, \sigma \vdash T_i : (\sigma)\mathcal{F}^d$ and $\sigma = \{i\}[\text{TYJUD}_1]$

`T(R36) OK`

Case R37.$\delta_{ij}F_j \xrightarrow[rule]{} F_i$

Similar approach to R36 `T( R37) OK`

Case R38.$\delta_{ij}V \circledast H^{\delta_{cj}} \xrightarrow[rule]{} V \circledast H^{\delta_{ci}}$

Given $\Gamma, \sigma \vdash V \circledast H^{\delta_{cj}} : (\sigma)\mathcal{F}^d$ and $\sigma = \{j\}$ by [TYJUD$_2$]

then $\Gamma, \sigma \vdash \delta_{ij}(V \circledast H^{\delta_{cj}}) : (\sigma)\mathcal{F}^d$ and $\sigma = \{i\}$ by [TYJUD$_5$]

$\Gamma, \sigma \vdash V \circledast H^{\delta_{ci}} : (\sigma)\mathcal{F}^d$ and $\sigma = \{i\}$[TYJUD$_2$]

`T( R38) OK`

Case R39.$\delta_{ij}V \circledast H^{\delta_{cj}}(x) \xrightarrow[rule]{} V \circledast H^{\delta_{ci}}(x)$

Similar approach to R38 `T( R39) OK`

Case R40.$\delta_{ij}\frac{\partial}{\partial x_j} \diamond e_1 \xrightarrow[rule]{} \frac{\partial}{\partial x_i} \diamond (e_1)$. Included in the earlier prose.

Case R41.$\sum(se_1) \xrightarrow[rule]{} s\sum e_1$. Included in the earlier prose.

Case R42.$\frac{\partial}{\partial x_\alpha} \diamond \frac{\partial}{\partial x_\beta} \diamond e_1 \xrightarrow[rule]{} \frac{\partial}{\partial x_{\beta\alpha}} \diamond e_1$

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

Claim: $\Gamma, \sigma/\ \alpha\beta \vdash e_1 : (\sigma/\ \alpha\beta)\mathcal{F}^d$

The LHS has the following type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_\alpha} \diamond \frac{\partial}{\partial x_\beta} \diamond e_1 : (\sigma)\mathcal{F}^d$

We want to show that the RHS has the same type.

$\Gamma, \sigma \vdash \frac{\partial}{\partial x_{\beta\alpha}} \diamond e_1 : (\sigma)\mathcal{F}^d$.

The type derivation for the LHS is the following structure.

$$\frac{\dfrac{\Gamma, \sigma \vdash e_1 : (\sigma/\ \alpha\beta)\mathcal{F}^d[\text{TYJUD}_4]}{\Gamma, \sigma \vdash (\dfrac{\partial}{\partial x_\beta} \diamond e_1) : (\sigma/\ \alpha)\mathcal{F}^d[\text{TYJUD}_4]}}{\Gamma, \sigma \vdash (\dfrac{\partial}{\partial x_\alpha} \diamond \dfrac{\partial}{\partial x_\beta} \diamond e_1) : (\sigma)\mathcal{F}^d}$$

From that we can make the RHS derivations.

Given that $\Gamma, \sigma \vdash e : \sigma/\ \alpha\beta$ then $\Gamma, \sigma \vdash \frac{\partial}{\partial x_{\beta\alpha}} \diamond e : (\sigma)\mathcal{F}^d$ by [TYJUD$_4$]

`T( R42) OK` T(d) Lemma 4.1.1

A.2 Value Preservation Proof

The following is a proof for Theorem 4.2.1 Given a derivation d of the form $e \longrightarrow e'$ 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

Case Rules R1-R5 use the probe operator.

Value representation of the probe operator is not supported.

Case Rules R6-R21 use the differentiation operator.

Value representation of the differentiation operator is not supported.

Case R22.$--e_1 \xrightarrow[rule]{} e_1$

Claim $--e_1$ evaluates to v.

We need to define v.

Assume that $e_1 \Downarrow v'$

then $\Psi, \rho \vdash -e_1 \Downarrow -v'$ by [VALJUD$_4$],

and $\Psi, \rho \vdash --e_1 \Downarrow --v'$ by [VALJUD$_4$]

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`

Case R23.$-0 \xrightarrow[rule]{} 0$

Claim -0 evaluates to v.

We need to define v.

then $\Psi, \rho \vdash 0 \Downarrow \mathit{Real}()(0)$ by [VALJUD$_1$] , and $\Psi, \rho \vdash -0 \Downarrow \mathit{Real}()(-0)$ by [VALJUD$_4$]

The value of v is $\mathit{Real}()(-0)$

By using algebraic reasoning: $\mathit{Real}()(-0) = \mathit{Real}()(0)$

The last step leads to $0 \Downarrow v$

`V( R23) OK`

Case R24.$e_1 - 0 \underset{rule}{\longrightarrow} e_1$

Included in the earlier prose.

Case R25.$0 - e_1 \underset{rule}{\longrightarrow} -e_1$

Claim $0 - e_1$ evaluates to v.

We need to define v.

Assume that $-e_1 \Downarrow v'$

then $\Psi, \rho \vdash 0 - e_1 \Downarrow \mathit{Real}()(0) + v'$ by ([VALJUD$_1$], [VALJUD$_5$]).

The value of v is $\mathit{Real}()(0) + v'$. By using algebraic reasoning: $\mathit{Real}()(0) + v' = v'$.

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`

Case R26.$\frac{0}{e_1} \underset{rule}{\longrightarrow} 0$

Assume that $e_1 \Downarrow \mathit{Real}()(v2)$ then $\Psi, \rho \vdash \frac{0}{e_1} \Downarrow \mathit{Real}()(\frac{0}{v2})$ by ([VALJUD$_1$], [VALJUD$_5$]).

The value of v is $\mathit{Real}()(\frac{0}{v2})$. By using algebraic reasoning: $\mathit{Real}()(\frac{0}{v2}) = \mathit{Real}()(0)$

Lastly, $\Psi, \rho \vdash 0 \Downarrow \mathit{Real}()(0)$ by ([VALJUD$_1$])

The last step leads to $0 \Downarrow v$

`V(R26) OK`

Case R27.$\frac{\frac{e_1}{e_2}}{e_3} \underset{rule}{\longrightarrow} \frac{e_1}{e_2 e_3}$

Claim $\frac{\frac{e_1}{e_2}}{e_3}$ evaluates to v.

We need to define v.

Assume that $\frac{e_1}{e_2 e_3} \Downarrow v', e_1 \Downarrow v1, e_2 \Downarrow v2, e_3 \Downarrow v3$.

then $\Psi, \rho \vdash \frac{e_1}{e_2} \Downarrow \frac{v1}{v2}$ by [VALJUD$_5$] and $\Psi, \rho \vdash \frac{\frac{e1}{e_2}}{e_3} \Downarrow \frac{\frac{v1}{v2}}{v3}$ by [VALJUD$_5$].

Given that $e_1 \Downarrow v1$ $e_2 \Downarrow v2$ $e_3 \Downarrow v3$

then $\Psi, \rho \vdash e_2 e_3 \Downarrow v2 * v3$ by [VALJUD$_5$] and $\Psi, \rho \vdash \frac{e1}{e_2 e_3} \Downarrow \frac{v1}{v2 * v3}$ by [VALJUD$_5$].

The value of v is $\frac{v1}{v2 * v3}$. By using algebraic reasoning: $v' = \frac{v1}{v2 * v3} = \frac{\frac{v1}{v2}}{v3} = v$.

The last step leads to $\frac{e_1}{e_2 e_3} \Downarrow v$

`V(R27) OK`

Case R28. $\frac{\frac{e_1}{e_2}}{e_3} \underset{rule}{\longrightarrow} \frac{e_1 e_3}{e_2}$

Similar approach to R27 `V( R28) OK`

Case R29. $\frac{\frac{\frac{e_1}{e_2}}{e_3}}{e_4} \underset{rule}{\longrightarrow} \frac{e_1 e_4}{e_2 e_3}$

Similar approach to R27 `V( R29) OK`

Case R30. $0 + e_1, e_1 + 0 \underset{rule}{\longrightarrow} e_1$ Claim $0 + e_1, e_1 + 0$ evaluates to v.

We need to define v.

Assume that $e_1 \Downarrow v'$ then $\Psi, \rho \vdash e_1 + 0 \Downarrow v' + \mathit{Real}()(0)$ by $([\text{VALJUD}_1], [\text{VALJUD}_5])$.

By using algebraic reasoning $v' + \mathit{Real}()(0) = v'$

The last step leads to $e_1 \Downarrow v$

`V( R30) OK`

Case R31. $0e, e0 \underset{rule}{\longrightarrow} 0$

Similar approach to R26 `V( R31) OK`

Case R32. $\sqrt{(e_1)} * \sqrt{(e_1)} \underset{rule}{\longrightarrow} e_1$

Included in the earlier prose.

Case R33. $\mathcal{E}_{ijk} \frac{\partial}{\partial x_{i,j}} \diamond e_1 \underset{rule}{\longrightarrow} \mathbf{lift}_d(0)$

Value representation not supported

Case R34. $\mathcal{E}_{ijk}(V_\alpha \circledast h^{jk}) \underset{rule}{\longrightarrow} \mathbf{lift}_d(0)$

Value representation not supported

Case R35. $\mathcal{E}_{ijk}\mathcal{E}_{ilm} \underset{rule}{\longrightarrow} \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$

Included in the earlier prose.

Case R36. $\delta_{ij} T_j \underset{rule}{\longrightarrow} T_i$

Included in the earlier prose.

Case Rules R37-R40 uses field terms

Value representation of the field terms is not supported.

Case R41. $\sum(se_1) \underset{rule}{\longrightarrow} s \sum e_1$

Claim $\sum(se_1)$ evaluates to v.

We need to define v.

Assume that $s \Downarrow v_s$ and $e_1 \Downarrow v_e$

then $\Psi, \rho \vdash s * e_1 \Downarrow v_s * v_e$ by ([VALJUD$_5$])

and $\Psi, \rho \vdash \sum(se_1) \Downarrow \sum(v_s * v_e)$ by [VALJUD$_4$]

The value of v is $\sum(v_s * v_e)$

$v = v_s * \sum(v_e)$ by moving scalar outside summation

We need to show that $s \sum e_1$ evaluates to v.

Given that $s \Downarrow v_s$ and $e \Downarrow v_e$

then $\Psi, \rho \vdash \sum e \Downarrow \sum v_e$ by ([VALJUD$_4$]) and $\Psi, \rho \vdash s \sum e_1 \Downarrow v_s * \sum v_e$ by ([VALJUD$_5$])

The last step leads to $s \sum e_1 \Downarrow v$

`V( R41) OK`

Case R42. $\frac{\partial}{\partial x_\alpha} \diamond \frac{\partial}{\partial x_\beta} \diamond e_1 \xrightarrow[rule]{} \frac{\partial}{\partial x_{\beta\alpha}} \diamond e_1$

Value representation not supported

A.3 Termination

A.3.1 Size reduction

If e $\Longrightarrow$ e' then $\mathcal{S}(e) > \mathcal{S}(e') \geq 0$ (Lemma 4.3.1). The following are a few helpful lemmas that will be referred to in the proof.

Lemma A.3.1. $5^{(1+x)} > (16 + 5^x)$

$5^x > 4.$	*Given* $x >= 1$
$4 * 5^x > 16$	*Multiply by 4*
$5 * 5^x - 5^x > 16$	*Refactor left side*
$5 * 5^x > (16 + 5^x)$	*Add* 5^x
$5^{(1+x)} > (16 + 5^x)$	*Rewritten*

Lemma A.3.2. $5^{([\![e_1]\!]+[\![e_2]\!])} > 5^{([\![e_1]\!])} > 4.$

Lemma A.3.3. $(1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} > [\![e_1]\!](16 + 5^{[\![e_1]\!]}) + 20$

$5^{(1+[\![e_1]\!])} > 16 + 5^{[\![e_1]\!]}$ *Lemma A.3.1*

$[\![e_1]\!]5^{(1+[\![e_1]\!])} > [\![e_1]\!](16 + 5^{[\![e_1]\!]})$ *Multiply by* $[\![e_1]\!]$

$[\![e_1]\!]5^{(1+[\![e_1]\!])} + 5^{(1+[\![e_1]\!])} > [\![e_1]\!](16 + 5^{[\![e_1]\!]}) + 5^{(1+[\![e_1]\!])}$ *Add* $5^{(1+[\![e_1]\!])}$

$(1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} > [\![e_1]\!](16 + 5^{[\![e_1]\!]}) + 5 * 5^{[\![e_1]\!]} > [\![e_1]\!](16 + 5^{[\![e_1]\!]}) + 20$ (*Lemma A.3.2*)

The following is a proof for Lemma 4.3.1 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. This proof does a case analysis to show $\forall d \in Deriv.P(d)$. Case on structure of d

Case R1.$(e_1 \odot_n e_2)@x \xrightarrow[rule]{} (e_1@x) \odot_n (e_2@x)$. Included in the earlier prose.

Case R2.$(e_0 \odot_2 e_1)@x \xrightarrow[rule]{} (e_0@x) \odot_2 (e_1@x)$

$$\begin{aligned}[\![(e_0 \odot_2 e_1)@x]\!] &= 2 + 2[\![e_1]\!] + 2[\![e_2]\!] \\ &> 1 + 2[\![e_1]\!] + 2[\![e_2]\!] \\ &= [\![(e_0@x) \odot_2 (e_1@x)]\!]\end{aligned}$$

P(d)

Case R3.$(\odot_1 e_1)@x \xrightarrow[rule]{} \odot_1 (e_1@x)$

$$\begin{aligned}[\![(\odot_1 e_1)@x]\!] &= 2 + 2[\![e_1]\!] \\ &> 1 + 2[\![e_1]\!] \quad = [\![\odot_1(e_1@x)]\!]\end{aligned}$$

P(d)

Case R4.$(\sum_{i=1}^{n} e_1)@x \xrightarrow[rule]{} \sum_{i=1}^{n}(e_1@x)$

$$\begin{aligned}[\![(\sum_{i=1}^{n} e_1)@x]\!] &= 4 + 4[\![e_1]\!] \\ &> 2 + 4[\![e_1]\!] \\ &= [\![\sum_{i=1}^{n}(e_1@x)]\!]\end{aligned}$$

P(d)

Case R5.$(\chi)@x \xrightarrow[rule]{} \chi$

$[\![(\chi)@x]\!] \;= 2\mathcal{S}(\chi)$

$\qquad\qquad > \mathcal{S}(\chi) \quad = [\![\chi]\!]$

Case R6. $\frac{\partial}{\partial x_i} \diamond (e_1 * e_2) \xrightarrow[rule]{} e_1(\frac{\partial}{\partial x_i} \diamond e_2) + e_2(\frac{\partial}{\partial x_i} \diamond e_1)$

We define $[\![(\frac{\partial}{\partial x_i} \diamond (e_1 * e_2))]\!]$=$s_1 + s_2 + s_3$

where $s_1 = [\![e_1]\!] * 5^{1+[\![e_1]\!]+[\![e_2]\!]}$, $s_2 = [\![e_2]\!] * 5^{1+[\![e_1]\!]+[\![e_2]\!]}$, and $s_3 = 5^{1+[\![e_1]\!]+[\![e_2]\!]}$,

We define $[\![(e_1 \frac{\partial}{\partial x_i} \diamond e_2 + e_2 \frac{\partial}{\partial x_i} \diamond e_1)]\!]$=$t_1 + t_2 + t_3$

where $t_1 = [\![e_1]\!](5^{[\![e_1]\!]} + 1)$, $t_2 = [\![e_2]\!](5^{[\![e_1]\!]} + 1)$, and $t_3 = 3$

Given $4 * 5^{1+[\![e_1]\!]} > 1$ then

$\longrightarrow 5 * 5^{[\![e_1]\!]} > 5^{[\![e_1]\!]} + 1$ by adding $5^{[\![e_1]\!]}$

$\longrightarrow 5^{1+[\![e_1]\!]+[\![e_2]\!]} > 5^{[\![e_1]\!]} + 1$ by refactoring

$\longrightarrow [\![e_1]\!] * 5^{1+[\![e_1]\!]+[\![e_2]\!]} > [\![e_1]\!](5^{[\![e_1]\!]} + 1)$ by multiplying by $[\![e_1]\!]$

$\longrightarrow [\![e_2]\!] * 5^{1+[\![e_1]\!]+[\![e_2]\!]} > [\![e_2]\!](5^{[\![e_1]\!]} + 1)$ by multiplying by $[\![e_2]\!]$

where and so $s_1 > t_1$,$s_2 > t_2$

where Lastly, $5^{1+[\![e_1]\!]+[\![e_2]\!]} > 3$ (Lm A.3.2) and so $s_3 > t_3$

Finally, $[\![\frac{\partial}{\partial x_i} \diamond (e_1 * e_2)]\!] > [\![e_1 \frac{\partial}{\partial x_i} \diamond e_2 + e_2 \frac{\partial}{\partial x_i} \diamond e_1]\!]$

P(d)

Case R7. $\frac{\partial}{\partial x_i} \diamond (\frac{e_1}{e_2}) \xrightarrow[rule]{} \frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2}$

We define $[\![(\frac{\partial}{\partial x_i} \diamond (\frac{e_1}{e_2}))]\!]$=$s_1 + s_2 + s_3$

where $s_1 = [\![e_1]\!]5^{2+[\![e_1]\!]+[\![e_2]\!]}$, $s_2 = [\![e_2]\!]5^{2+[\![e_1]\!]+[\![e_2]\!]}$, and $s_3 = 2 * 5^{2+[\![e_1]\!]+[\![e_2]\!]}$

We define $[\![(\frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2})]\!]$=$t_1 + t_2 + t_3$

where $t_1 = [\![e_1]\!](1 + 5^{[\![e_1]\!]})$, $t_2 = [\![e_2]\!](3 + 5^{[\![e_2]\!]})$, and $t_3 = 6$

Given $5^{2+[\![e_1]\!]+[\![e_2]\!]} > (1 + 5^{[\![e_1]\!]})$(Lm A.3.1)

where then $[\![e_1]\!]5^{2+[\![e_1]\!]+[\![e_2]\!]} > [\![e_1]\!](1 + 5^{[\![e_1]\!]})$ by multiplying by $[\![e_1]\!]$

where so $s_1 > t_1$,$s_2 > t_2$

Given $5^{1+[\![e_1]\!]+[\![e_2]\!]} > 5^{[\![e_2]\!]} + 3$ (Lm A.3.1)

where then $2 * 5^{1+[\![e_1]\!]+[\![e_2]\!]} > 2 * 5^{[\![e_2]\!]} + 6$ by multiplying by 2

where so $s_3 > t_3$

$[\![\text{source}(d)]\!] > [\![\text{target}(d)]\!]$

P(d)

Case R8. $\frac{\partial}{\partial x_i} \diamond (\sqrt{e_1}) \xrightarrow[rule]{} \mathbf{lift}_d(1/2) * \frac{\frac{\partial}{\partial x_i} \diamond e_1}{\sqrt{e_1}}$

$$\begin{aligned}
[\![\frac{\partial}{\partial x_i} \diamond (\sqrt{e_1})]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\
&> [\![e_1]\!](1 + 5^{[\![e_1]\!]}) + 6 \\
&= [\![\mathbf{lift}_d(1/2) * \frac{\frac{\partial}{\partial x_i} \diamond e}{\sqrt{e_1}}]\!]
\end{aligned}$$

P(d)

Case R9. $\frac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1)) \xrightarrow[rule]{} (-\mathbf{sine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$. Included in the earlier prose.

Case R10. $\frac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1)) \xrightarrow[rule]{} (\mathbf{cosine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$

$$\begin{aligned}
[\![\frac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1))]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\
&> [\![e_1]\!](1 + 5^{[\![e_1]\!]}) + 2 \\
&= [\![(\mathbf{cosine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)]\!]
\end{aligned}$$

P(d)

Case R11. $\frac{\partial}{\partial x_i} \diamond (\mathbf{tangent}(e_1)) \xrightarrow[rule]{} \frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{cosine}(e_1) * \mathbf{cosine}(e_1)}$

$$\begin{aligned}
[\![\frac{\partial}{\partial x_i} \diamond (\mathbf{tangent}(e_1))]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\
&> [\![e_1]\!](5^{[\![e_1]\!]} + 2) + 5 \\
&= [\![\frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{cosine}(e_1) * \mathbf{cosine}(e_1)}]\!]
\end{aligned}$$

P(d)

Case R12. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arccosine}(e_1)) \xrightarrow[rule]{} (\frac{-\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)$

$$\begin{aligned}
[\![\frac{\partial}{\partial x_i} \diamond (\mathbf{arccosine}(e_1))]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\
&> [\![e_1]\!](2 + 5^{[\![e_1]\!]}) + 11 \\
&= [\![(\frac{-\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)]\!]
\end{aligned}$$

P(d)

Case R13. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1)) \xrightarrow[rule]{} (\frac{\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)$

$$\begin{aligned}
[\![\frac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1))]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\
&> [\![e_1]\!](2 + 5^{[\![e_1]\!]}) + 10 \\
&= [\![(\frac{\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)]\!]
\end{aligned}$$

P(d)

Case R14. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arctangent}(e_1)) \xrightarrow[rule]{} \frac{\mathbf{lift}_d(1)}{\mathbf{lift}_d(1)+(e_1 * e_1)} * (\frac{\partial}{\partial x_i} \diamond e_1)$

$$\begin{aligned} [\![\frac{\partial}{\partial x_i} \diamond (\mathbf{arctangent}(e_1))]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\ &> [\![e_1]\!](2 + 5^{[\![e_1]\!]}) + 9 \\ &= [\![\frac{1}{1+(e*e)} * (\frac{\partial}{\partial x_i} \diamond e_1)]\!] \end{aligned}$$

P(d)

Case R15. $\frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e_1)) \xrightarrow[rule]{} \mathbf{exp}(e_1) * (\frac{\partial}{\partial x_i} \diamond e_1)$

$$\begin{aligned} [\![\frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e_1))]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\ &> [\![e_1]\!](1 + 5^{[\![e_1]\!]}) + 2 \\ &= [\![\mathbf{exp}(e_1) * (\frac{\partial}{\partial x_i} \diamond e_1)]\!] \end{aligned}$$

P(d)

Case R16. $\frac{\partial}{\partial x_i} \diamond (e_1^n) \xrightarrow[rule]{} \mathbf{lift}_d(n) * e_1^{n-1} * (\frac{\partial}{\partial x_i} \diamond e_1)$

$$\begin{aligned} [\![\frac{\partial}{\partial x_i} \diamond (e_1^n)]\!] &= (1 + [\![e_1]\!])5^{(1+[\![e_1]\!])} \\ &> 5 + [\![e_1]\!](1 + 5^{[\![e_1]\!]}) \\ &= [\![\mathbf{lift}_d(n) * e_1^{n-1} * (\frac{\partial}{\partial x_i} \diamond e_1)]\!] \end{aligned}$$

P(d)

Case R17. $\frac{\partial}{\partial x_i} \diamond (e_1 \odot e_2) \xrightarrow[rule]{} (\frac{\partial}{\partial x_i} \diamond e_1) \odot (\frac{\partial}{\partial x_i} \diamond e_2)$Included in the earlier prose.

Case R18. $\frac{\partial}{\partial x_i} \diamond (-e_1) \xrightarrow[rule]{} -(\frac{\partial}{\partial x_i} \diamond e_1)$

$$\begin{aligned} [\![\frac{\partial}{\partial x_i} \diamond (-e_1)]\!] &= 5^{1+[\![e_1]\!]}(1 + [\![e_1]\!]) \\ &> 1 + [\![e_1]\!]5^{[\![e_1]\!]} \\ &= [\![-(\frac{\partial}{\partial x_i} \diamond e_1)]\!] \end{aligned}$$

P(d)

Case R19. $\frac{\partial}{\partial x_i} \sum_{v=1}^{n} e_1 \xrightarrow[rule]{} \sum_{v=1}^{n} (\frac{\partial}{\partial x_i} e_1)$

$$\begin{aligned} [\![\frac{\partial}{\partial x_i} \sum_{v=1}^{n} e_1]\!] &= (2 + 2[\![e_1]\!]) * 5^{2+2[\![e_1]\!]} \\ &> 2 + 2[\![e_1]\!]5^{[\![e_1]\!]} \\ &= [\![\sum_{v=1}^{n} (\frac{\partial}{\partial x_i} e_1)]\!] \end{aligned}$$

P(d)

Case R20. $\frac{\partial}{\partial x_i}\chi \xrightarrow[rule]{} 0$

$[\![\frac{\partial}{\partial x_i}]\!] = \mathcal{S}\chi 5^{\mathcal{S}\chi}$

$> 2 \qquad = [\![0]\!]$

Case R21. $\frac{\partial}{\partial x_i} \diamond (V_\alpha \circledast H^\nu) \xrightarrow[rule]{} (V_\alpha \circledast h^{i\nu})$

$[\![\frac{\partial}{\partial x_i} \diamond (V_\alpha \circledast H^\nu)]\!] = 5$

$> 1 \quad = [\![(V_\alpha \circledast H^{i\nu})]\!]$

Case R22. $--e_1 \xrightarrow[rule]{} e_1$

$[\![--e_1]\!] = 2 + [\![e_1]\!]$

$> [\![e_1]\!] \qquad = [\![e_1]\!]$

Case R23. $-0 \xrightarrow[rule]{} 0$

$[\![-0]\!] = 2$

$> 1 \quad = [\![0]\!]$

Case R24. $e_1 - 0 \xrightarrow[rule]{} e_1$

$[\![e_1 - 0]\!] = 2 + [\![e_1]\!]$

$> [\![e_1]\!] \qquad = [\![e_1]\!]$

Case R25. $0 - e_1 \xrightarrow[rule]{} -e_1$

Similar approach to R24 `P( R25) OK`

Case R26. $\frac{0}{e_1} \xrightarrow[rule]{} 0$

$[\![\frac{0}{e_1}]\!] = 3 + [\![e_1]\!]$

$> 1 \qquad = [\![0]\!]$

Case R27. $\frac{\frac{e_1}{e_2}}{e_3} \xrightarrow[rule]{} \frac{e_1}{e_2 e_3}$ Included in the earlier prose.

Case R28. $\frac{e_1}{\frac{e_2}{e_3}} \xrightarrow[rule]{} \frac{e_1 e_3}{e_2}$

Similar approach to R27 `P( R28) OK`

Case R29. $\frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}} \xrightarrow[rule]{} \frac{e_1 e_4}{e_2 e_3}$

$[\![\frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}}]\!] = 6 + [\![e_1]\!] + [\![e_2]\!] + [\![e_3]\!]$

$> 4 + [\![e_1]\!] + [\![e_2]\!] + [\![e_3]\!] \quad = [\![\frac{e_1 e_4}{e_2 e_3}]\!]$

Case R30. $0 + e_1, e_1 + 0 \xrightarrow[rule]{} e_1$

$$[\![0 + e_1, e_1 + 0]\!] \quad = 2 + [\![e_1]\!]$$
$$> [\![e_1]\!] \qquad = [\![e_1]\!]$$

Case R31.$0e, e0 \xrightarrow[rule]{} 0$

Similar approach to R30 `P( R31) OK`

Case R32.$\sqrt{(e_1)} * \sqrt{(e_1)} \xrightarrow[rule]{} e_1$

$$[\![\sqrt{(e_1)} * \sqrt{(e_1)}]\!] \quad = 3 + 2[\![e_1]\!]$$
$$> [\![e_1]\!] \qquad = [\![e_1]\!]$$

Case R33.$\mathcal{E}_{ijk} \frac{\partial}{\partial x_{i,j}} \diamond e_1 \xrightarrow[rule]{} \mathbf{lift}_d(0)$

$$[\![\mathcal{E}_{ijk} \frac{\partial}{\partial x_{i,j}} \diamond e_1]\!] \quad = 5 + [\![e_1]\!] 5 [\![e_1]\!]$$
$$> 2 \qquad = [\![\mathbf{lift}_d(0)]\!]$$

Case R34.$\mathcal{E}_{ijk}(V_\alpha \circledast h^{jk}) \xrightarrow[rule]{} \mathbf{lift}_d(0)$

$$[\![\mathcal{E}_{ijk}(V_\alpha \circledast h^{jk})]\!] \quad = 6$$
$$> 2 \quad = [\![\mathbf{lift}_d(0)]\!]$$

Case R35.$\mathcal{E}_{ijk}\mathcal{E}_{ilm} \xrightarrow[rule]{} \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$

$$[\![\mathcal{E}_{ijk}\mathcal{E}_{ilm}]\!] \quad = 9$$
$$> 7 \quad = [\![\delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}]\!]$$

Case R36.$\delta_{ij}T_j \xrightarrow[rule]{} T_i$

$$[\![\delta_{ij}T_j]\!] \quad = 3$$
$$> 1 \quad = [\![T_i]\!]$$

Case R37.$\delta_{ij}F_j \xrightarrow[rule]{} F_i$

Similar approach to R36 `P( R37) OK`

Case R38.$\delta_{ij}V \circledast H^{\delta_{cj}} \xrightarrow[rule]{} V \circledast H^{\delta_{ci}}$

Similar approach to R36 `P( R38) OK`

Case R39.$\delta_{ij}V \circledast H^{\delta_{cj}}(x) \xrightarrow[rule]{} V \circledast H^{\delta_{ci}}(x)$

$$[\![\delta_{ij}V \circledast H^{\delta_{cj}}(x)]\!] \quad = 4$$
$$> 2 \quad = [\![V \circledast H^{\delta_{ci}}(x)]\!]$$

Case R40.$\delta_{ij} \frac{\partial}{\partial x_j} \diamond e_1 \xrightarrow[rule]{} \frac{\partial}{\partial x_i} \diamond (e_1)$

$$[\![\delta_{ij}\frac{\partial}{\partial x_j}\diamond(e_1)]\!] = 2 + [\![e_1]\!]5^{[\![e_1]\!]}$$
$$> [\![e_1]\!]5^{[\![e_1]\!]} = [\![\frac{\partial}{\partial x_i}\diamond(e_1)]\!]$$

Case R41.$\sum(se_1) \xrightarrow[rule]{} s\sum e_1$

$$[\![\sum(se_1)]\!] = 6 + 2[\![e_1]\!]$$
$$> 4 + 2[\![e_1]\!] = [\![s\sum e_1]\!]$$

P(d) Lemma 4.3.1

A.3.2 Termination implies Normal Form

Termination implies normal form (Lemma 4.3.2). 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 $Q(e_x)$ as a shorthand for the claim that if x has terminated and is normal form. Additionally we state $CQ(e_x)$ if there exists an expression that is not in normal form and has terminated. The following is a proof by contradiction.

Define the following shorthand: $\mathrm{M}(e_1) = \sqrt{e_1} \mid exp(e_1) \mid e_1^n \mid \kappa(e_1)$

case on structure e_x

If $e_x = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = V_\alpha \circledast H$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \delta_{ij}$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \mathcal{E}_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \mathbf{lift}_d(e_1)$

Prove Q(e_x) by contradiction.

case on structure e_1

If $e_1 = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = F_\alpha$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = e \circledast e$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = \delta_{ij}$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathcal{E}_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathbf{lift}_d(e)$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = M(e)$ and assuming $Q(e)$ then $Q(e_x)$

Given $M(e_3) = \sqrt{e_3} \mid exp(e_3) \mid e_3^n \mid \kappa(e_3)$

If $e_1 = -e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \frac{\partial}{\partial x_\alpha} \diamond e$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = \sum e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e_3 + e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 - e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 * e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = \frac{e_3}{e_4}$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 @ e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

$Q(e_x)$

e_x=M(e_1)

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

Note. M(e_1) = $\sqrt{e_3} \mid exp(e_3) \mid e_3^n \mid \kappa(e_3)$

If $e_1 = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = V_\alpha \circledast H$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \delta_{ij}$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathcal{E}_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathbf{lift}_d(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = M(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = -e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \frac{\partial}{\partial x_\alpha} e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \sum e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e_3 + e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 - e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 * e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = \frac{e_3}{e_4}$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 @ e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

$Q(e_x)$

$e_x = -e_1$

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

If $e_1 = 0$ then $Q(e_x)$ because we can apply rule$R23$

If $e_1 = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = V_\alpha \circledast H$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \delta_{ij}$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathcal{E}_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathbf{lift}_d(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = M(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = -e$ then $Q(e_x)$ because we can apply rule$R22$

If $e_1 = \frac{\partial}{\partial x_\alpha} e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \sum e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e_3 + e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 - e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 * e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = \frac{e_3}{e_4}$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 @ e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

$Q(e_x)$

$e_x = e_1 + e_2$

Prove Q(x)

case on structure e_1

If $e_x = 0$ then $Q(e_x)$ because we can apply rule $R30$

If $e_x = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = V_\alpha \circledast H$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \delta_{ij}$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \mathcal{E}_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_x = \mathbf{lift}_d(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_x = M(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_x = -e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_x = \frac{\partial}{\partial x_\alpha} e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \sum e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e_3 + e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 - e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 * e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = \frac{e_3}{e_4}$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 @ e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

$Q(e_x)$

case on structure e_2

Proof same as above Q(x)

$e_x = e_1 - e_2$

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

If $e_1 = 0$ then $Q(e_x)$ because we can apply rule $R25$

If $e_1 = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = V_\alpha \circledast H$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \delta_{ij}$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathcal{E}_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathbf{lift}_d(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = M(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = -e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \frac{\partial}{\partial x_\alpha} e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \sum e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e_3 + e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 - e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 * e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = \frac{e_3}{e_4}$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 @ e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

$Q(e_x)$

case on structure e_2

If $e_x = 0$ then $Q(e_x)$ because we can apply rule $R24$

Proof same as above

$Q(x)$

$e_x = e_1 * e_2$

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

If $e_1 = 0$ then $Q(e_x)$ because we can apply rule $R31$

If $e_1 = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = V_\alpha \circledast H$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \delta_{ij}$

case on structure e_2

If $e_2 = T_j$ then $Q(e_x)$ because we can apply rule $R36$

If $e_2 = F_j$ then $Q(e_x)$ because we can apply rule $R37$

If $e_2 = V_\alpha \circledast H$ then $Q(e_x)$ because we can apply rule $R38$

If $e_2 = V_\alpha \circledast H @ e$ then $Q(e_x)$ because we can apply rule $R39$

If $e_2 = \frac{\partial}{\partial x_\alpha} e$ then $Q(e_x)$ because we can apply rule $R40$

else $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathcal{E}_{ij}$

If $e_1 = \mathcal{E}_{ijk}$

case on structure e_2

If $e_2 = \frac{\partial}{\partial x_{ij}}(e)$ then $Q(e_x)$ because we can apply rule $R33$

If $e_2 = V \circledast H_{jk}$ then $Q(e_x)$ because we can apply rule $R34$

If $e_2 = \mathcal{E}_{ijk}$ then $Q(e_x)$ because we can apply rule $R35$

else $Q(e_x)$ because e_x is in normal form.

If $e_1 = \mathbf{lift}_d(e_1)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \sqrt{e_3}$

If $e_2 = \sqrt{e_4}$ then $Q(e_x)$ because we can apply rule $R32$

otherwise $Q(e_x)$ because e_x is in normal form.

If $e_1 = -e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \frac{\partial}{\partial x_\alpha} \diamond e$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = \sum e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e_3 + e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 - e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 * e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = \frac{e_3}{e_4}$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

If $e_1 = e_3 @ e_4$ and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$

$Q(e_x)$

$e_x = \frac{e_1}{e_2}$

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

If $e_1 = \frac{e_3}{e_4}$

If $e_2 = \frac{e_5}{e_6}$ then $Q(e_x)$ because we can apply rule $R27$ otherwise $Q(e_x)$ because we can apply rule R29.

If $e_1 = 0$ then $Q(e_x)$ because we can apply rule $R26$

If $e_1 = c$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = V \circledast H$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \delta_{ij}, \mathcal{E}_{ij}, \mathcal{E}_{ijk}$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = \frac{\partial}{\partial x_\alpha} e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \sum e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \mathbf{lift}_d(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = M(e)$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = -e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e + e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e - e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e * e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e@e$ and assuming $Q(e)$ then $Q(e_x)$

case on structure e_2

If $e_2 = \frac{e_4}{e_5}$ then $Q(e_x)$ because we can apply rule $R28$ otherwise proof same as above

$Q(e_x)$

$e_x = e_1@e_2$

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

If $e_1 = c$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = F_\alpha$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = e \circledast e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \delta_{ij}, \mathcal{E}_\alpha$ then $Q(e_x)$ because we can apply rule $R5$

If $e_1 = \mathbf{lift}_d(e)$ then $Q(e_x)$ because we can apply rule $R5$

If $e_1 = M(e)$ then $Q(e_x)$ because we can apply rule $R3$

If $e_1 = -e$ then $Q(e_x)$ because we can apply rule $R3$

If $e_x = \frac{\partial}{\partial x_\alpha} \diamond e$ and assuming $Q(e)$ then $Q(e_x)$

If $e_1 = \sum e$ then $Q(e_x)$ because we can apply rule $R4$

If $e_1 = e + e$ then $Q(e_x)$ because we can apply rule $R2$

If $e_1 = e - e$ then $Q(e_x)$ because we can apply rule $R2$

If $e_1 = e * e$ then $Q(e_x)$ because we can apply rule $R1$

If $e_1 = \frac{e}{e}$ then $Q(e_x)$ because we can apply rule $R1$

If $e_1 = e@e$ then $Q(e_x)$ because e_x is not a supported type.

$Q(e_x)$

$e_x = \frac{\partial}{\partial x_\alpha} e_1$

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

If $e_1 = c$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = T_\alpha$ then $Q(e_x)$ because e_x is not a supported type.

If $e_1 = F_\alpha$ then $Q(e_x)$ because e_x is in normal form.

If $e_1 = e \circledast e$ then $Q(e_x)$ because we can apply rule $R21$

If $e_1 = \delta_{ij}, \mathcal{E}_\alpha$ then $Q(e_x)$ because we can apply rule $R20$

If $e_1 = \mathbf{lift}_d(e)$ then $Q(e_x)$ because we can apply rule $R20$

If $e_1 = M(e_2)$

case on structure e_2

If $e_2 = Cosine(e)$ then $Q(e_x)$ because we can apply rule $R9$

If $e_2 = Sine(e)$ then $Q(e_x)$ because we can apply rule $R10$

If $e_2 = Tangent(e)$ then $Q(e_x)$ because we can apply rule $R11$

If $e_2 = ArcCosine(e)$ then $Q(e_x)$ because we can apply rule $R12$

If $e_2 = ArcSine(e)$ then $Q(e_x)$ because we can apply rule $R13$

If $e_2 = ArcTangent(e)$ then $Q(e_x)$ because we can apply rule $R14$

If $e_2 = exp(e)$ then $Q(e_x)$ because we can apply rule $R15$

If $e_2 = e^n$ then $Q(e_x)$ because we can apply rule $R16$

If $e_2 = \sqrt{e}$ then $Q(e_x)$ because we can apply rule $R8$

$Q(e_x)$

If $e_1 = -e$ then $Q(e_x)$ because we can apply rule $R18$

If $e_1 = \frac{\partial}{\partial x_\alpha} \diamond e$ then $Q(e_x)$ because we can apply rule $R42$

If $e_1 = \sum e$ then $Q(e_x)$ because we can apply rule $R19$

If $e_1 = e + e$ then $Q(e_x)$ because we can apply rule $R17$

If $e_1 = e - e$ then $Q(e_x)$ because we can apply rule $R17$

If $e_1 = e * e$ then $Q(e_x)$ because we can apply rule $R6$

If $e_1 = \frac{e}{e}$ then $Q(e_x)$ because we can apply rule $R7$

If $e_1 = e@e$ then $Q(e_x)$ because e_x is not a supported type.

$Q(e_x)$

e_x=$\sum(e_1)$

Show Q(x) with proof by contradiction. Assume CQ(Q_x)

case on structure e_1

If $e_1 = c$	then $Q(e_x)$ because we can apply rule $R41$
If $e_1 = T$	then $Q(e_x)$ because we can apply rule $R41$
If $e_1 = T_\alpha$	then $Q(e_x)$ because e_x is in normal form.
If $e_1 = F$	then $Q(e_x)$ because we can apply rule $R41$
If $e_1 = F_\alpha$	then $Q(e_x)$ because e_x is in normal form.
If $e_1 = V_\alpha \circledast H$	then $Q(e_x)$ because we can apply rule $R41$
If $e_1 = \delta_{ij}, \mathcal{E}_\alpha$	then $Q(e_x)$ because e_x is in normal form.
If $e_1 = \mathbf{lift}_d(e)$	and assuming $Q(e)$ then $Q(e_x)$
If $e_1 = M(e)$	and assuming $Q(e)$ then $Q(e_x)$
If $e_1 = -e$	and assuming $Q(e)$ then $Q(e_x)$
If $e_1 = \frac{\partial}{\partial x_\alpha} e$	and assuming $Q(e)$ then $Q(e_x)$
If $e_1 = \sum e_1$	and assuming $Q(e)$ then $Q(e_x)$
If $e_1 = e_3 + e_4$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e_1 = e_3 - e_4$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e_1 = e_3 * e_4$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e_1 = \frac{e_3}{e_4}$	and assuming $Q(e_3)$ and $Q(e_4)$ then $Q(e_x)$
If $e_1 = F@e$	then $Q(e_x)$ because we can apply rule $R41$
If $e_1 = V \circledast h@e$	then $Q(e_x)$ because we can apply rule $R41$
If $e_1 = e@e$	then $Q(e_x)$ because e_x is in normal form.
$Q(e_x)$	

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. □

Case analysis on the source of each rule

Case R1.$(e_1 \odot_n e_2)@x \xrightarrow[rule]{} (e_1@x) \odot_n (e_2@x)$

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

Case R2.$(e_0 \odot_2 e_1)@x \xrightarrow[rule]{} (e_0@x) \odot_2 (e_1@x)$

Let y= $(e_0 \odot_2 e_1)@x$ and since y is not in normal form then `M( R2) OK`

Case R3.$(\odot_1 e_1)@x \xrightarrow[rule]{} \odot_1 (e_1@x)$

Let y= $(\odot_1 e_1)@x$ and since y is not in normal form then `M( R3) OK`

Case R4.$(\sum_{i=1}^{n} e_1)@x \xrightarrow[rule]{} \sum_{i=1}^{n} (e_1@x)$

Let y= $(\sum_{i=1}^{n} e_1)@x$ and since y is not in normal form then `M( R4) OK`

Case R5.$(\chi)@x \xrightarrow[rule]{} \chi$

Let y= $(\chi)@x$ and since y is not in normal form then `M( R5) OK`

Case R6.$\frac{\partial}{\partial x_i} \diamond (e_1 * e_2) \xrightarrow[rule]{} e_1(\frac{\partial}{\partial x_i} \diamond e_2) + e_2(\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (e_1 * e_2)$ and since y is not in normal form then `M( R6) OK`

Case R7.$\frac{\partial}{\partial x_i} \diamond (\frac{e_1}{e_2}) \xrightarrow[rule]{} \frac{(\frac{\partial}{\partial x_i} \diamond e_1)e_2 - e_1(\frac{\partial}{\partial x_i} \diamond e_2)}{e_2^2}$

Let y= $\frac{\partial}{\partial x_i} \diamond (\frac{e_1}{e_2})$ and since y is not in normal form then `M( R7) OK`

Case R8.$\frac{\partial}{\partial x_i} \diamond (\sqrt{e_1}) \xrightarrow[rule]{} \mathbf{lift}_d(1/2) * \frac{\frac{\partial}{\partial x_i} \diamond e_1}{\sqrt{e_1}}$

Let y= $\frac{\partial}{\partial x_i} \diamond (\sqrt{e_1})$ and since y is not in normal form then `M( R8) OK`

Case R9.$\frac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1)) \xrightarrow[rule]{} (-\mathbf{sine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (\mathbf{cosine}(e_1))$ and since y is not in normal form then `M( R9) OK`

Case R10.$\frac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1)) \xrightarrow[rule]{} (\mathbf{cosine}(e_1)) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (\mathbf{sine}(e_1))$ and since y is not in normal form then M(R10) OK

Case R11. $\frac{\partial}{\partial x_i} \diamond (\mathbf{tangent}(e_1)) \xrightarrow[rule]{} \frac{\frac{\partial}{\partial x_i} \diamond e}{\mathbf{cosine}(e_1) * \mathbf{cosine}(e_1)}$

Let y= $\frac{\partial}{\partial x_i} \diamond (\mathbf{tangent}(e_1))$ and since y is not in normal form then M(R11) OK

Case R12. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arccosine}(e_1)) \xrightarrow[rule]{} (\frac{-\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (\mathbf{arccosine}(e_1))$ and since y is not in normal form then M(R12) OK

Case R13. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1)) \xrightarrow[rule]{} (\frac{\mathbf{lift}_d(1)}{\sqrt{(\mathbf{lift}_d(1) - (e * e)}}) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (\mathbf{arcsine}(e_1))$ and since y is not in normal form then M(R13) OK

Case R14. $\frac{\partial}{\partial x_i} \diamond (\mathbf{arctangent}(e_1)) \xrightarrow[rule]{} \frac{\mathbf{lift}_d(1)}{\mathbf{lift}_d(1) + (e_1 * e_1)} * (\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (\mathbf{arctangent}(e_1))$ and since y is not in normal form then M(R14) OK

Case R15. $\frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e_1)) \xrightarrow[rule]{} \mathbf{exp}(e_1) * (\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (\mathbf{exp}(e_1))$ and since y is not in normal form then M(R15) OK

Case R16. $\frac{\partial}{\partial x_i} \diamond (e_1^n) \xrightarrow[rule]{} \mathbf{lift}_d(n) * e_1^{n-1} * (\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (e_1^n)$ and since y is not in normal form then M(R16) OK

Case R17. $\frac{\partial}{\partial x_i} \diamond (e_1 \odot e_2) \xrightarrow[rule]{} (\frac{\partial}{\partial x_i} \diamond e_1) \odot (\frac{\partial}{\partial x_i} \diamond e_2)$

Let y= $\frac{\partial}{\partial x_i} \diamond (e_1 \odot e_2)$ and since y is not in normal form then M(R17) OK

Case R18. $\frac{\partial}{\partial x_i} \diamond (-e_1) \xrightarrow[rule]{} -(\frac{\partial}{\partial x_i} \diamond e_1)$

Let y= $\frac{\partial}{\partial x_i} \diamond (-e_1)$ and since y is not in normal form then M(R18) OK

Case R19. $\frac{\partial}{\partial x_i} \sum_{v=1}^{n} e_1 \xrightarrow[rule]{} \sum_{v=1}^{n} (\frac{\partial}{\partial x_i} e_1)$

Let y= $\frac{\partial}{\partial x_i} \sum_{v=1}^{n} e_1$ and since y is not in normal form then M(R19) OK

Case R20. $\frac{\partial}{\partial x_i} \mathbf{lift}_d(e_1) \xrightarrow[rule]{} 0$

Let y= $\frac{\partial}{\partial x_i} \mathrm{Lift}(e_1)$ and since y is not in normal form then M(R20) OK

Case R20. $\frac{\partial}{\partial x_i} \chi \xrightarrow[rule]{} 0$

Let y= $\frac{\partial}{\partial x_i}$ and since y is not in normal form then M(R20) OK

Case R21. $\frac{\partial}{\partial x_i} \diamond (V_\alpha \circledast H^\nu) \xrightarrow[rule]{} (V_\alpha \circledast h^{i\nu})$

Let y= $\frac{\partial}{\partial x_i} \diamond (V_\alpha \circledast H^\nu)$ and since y is not in normal form then M(R21) OK

Case R22. $--e_1 \xrightarrow[rule]{} e_1$

Let y= $--e_1$ and since y is not in normal form then M(R22) OK

Case R23.$-0 \underset{rule}{\longrightarrow} 0$

Let y= -0 and since y is not in normal form then `M( R23) OK`

Case R24.$e_1 - 0 \underset{rule}{\longrightarrow} e_1$

Let y= $e_1 - 0$ and since y is not in normal form then `M( R24) OK`

Case R25.$0 - e_1 \underset{rule}{\longrightarrow} -e_1$

Let y= $0 - e_1$ and since y is not in normal form then `M( R25) OK`

Case R26.$\frac{0}{e_1} \underset{rule}{\longrightarrow} 0$

Let y= $\frac{0}{e_1}$ and since y is not in normal form then `M( R26) OK`

Case R27.$\frac{\frac{e_1}{e_2}}{e_3} \underset{rule}{\longrightarrow} \frac{e_1}{e_2 e_3}$

Let y= $\frac{\frac{e_1}{e_2}}{e_3}$ and since y is not in normal form then `M( R27) OK`

Case R28.$\frac{e_1}{\frac{e_2}{e_3}} \underset{rule}{\longrightarrow} \frac{e_1 e_3}{e_2}$

Let y= $\frac{e_1}{\frac{e_2}{e_3}}$ and since y is not in normal form then `M( R28) OK`

Case R29.$\frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}} \underset{rule}{\longrightarrow} \frac{e_1 e_4}{e_2 e_3}$

Let y= $\frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}}$ and since y is not in normal form then `M( R29) OK`

Case R30.$0 + e_1, e_1 + 0 \underset{rule}{\longrightarrow} e_1$

Let y= $0 + e_1, e_1 + 0$ and since y is not in normal form then `M( R30) OK`

Case R31.$0e, e0 \underset{rule}{\longrightarrow} 0$

Let y= $0e, e0$ and since y is not in normal form then `M( R31) OK`

Case R32.$\sqrt{(e_1)} * \sqrt{(e_1)} \underset{rule}{\longrightarrow} e_1$

Let y= $\sqrt{(e_1)} * \sqrt{(e_1)}$ and since y is not in normal form then `M( R32) OK`

Case R33.$\mathcal{E}_{ijk} \frac{\partial}{\partial x_{i,j}} \diamond e_1 \underset{rule}{\longrightarrow} \mathbf{lift}_d(0)$

Let y= $\mathcal{E}_{ijk} \frac{\partial}{\partial x_{i,j}} \diamond e_1$ and since y is not in normal form then `M( R33) OK`

Case R34.$\mathcal{E}_{ijk}(V_\alpha \circledast h^{jk}) \underset{rule}{\longrightarrow} \mathbf{lift}_d(0)$

Let y= $\mathcal{E}_{ijk}(V_\alpha \circledast h^{jk})$ and since y is not in normal form then `M( R34) OK`

Case R35.$\mathcal{E}_{ijk}\mathcal{E}_{ilm} \underset{rule}{\longrightarrow} \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$

Let y= $\mathcal{E}_{ijk}\mathcal{E}_{ilm}$ and since y is not in normal form then `M( R35) OK`

Case R36.$\delta_{ij}T_j \underset{rule}{\longrightarrow} T_i$

Let y= $\delta_{ij}T_j$ and since y is not in normal form then `M( R36) OK`

Case R37.$\delta_{ij}F_j \underset{rule}{\longrightarrow} F_i$

Let y= $\delta_{ij}F_j$ and since y is not in normal form then `M( R37) OK`

Case R38.$\delta_{ij}V \circledast H^{\delta_{cj}} \underset{rule}{\longrightarrow} V \circledast H^{\delta_{ci}}$

Let y= $\delta_{ij}V \circledast H^{\delta_{cj}}$ and since y is not in normal form then `M( R38) OK`

Case R39.$\delta_{ij}V \circledast H^{\delta_{cj}}(x) \underset{rule}{\longrightarrow} V \circledast H^{\delta_{ci}}(x)$

Let y= $\delta_{ij}V \circledast H^{\delta_{cj}}(x)$ and since y is not in normal form then `M( R39) OK`

Case R40.$\delta_{ij}\frac{\partial}{\partial x_j} \diamond e_1 \underset{rule}{\longrightarrow} \frac{\partial}{\partial x_i} \diamond (e_1)$

Let y= $\delta_{ij}\frac{\partial}{\partial x_j} \diamond (e_1)$ and since y is not in normal form then `M( R40) OK`

Case R41.$\sum(se_1) \underset{rule}{\longrightarrow} s\sum e_1$

Let y= $\sum(se_1)$ and since y is not in normal form then `M( R41) OK`

Case R42.$\frac{\partial}{\partial x_\alpha} \diamond \frac{\partial}{\partial x_\beta} \diamond e_1 \underset{rule}{\longrightarrow} \frac{\partial}{\partial x_{\beta\alpha}} \diamond e_1$

Let y= $\frac{\partial}{\partial x_\alpha} \diamond \frac{\partial}{\partial x_\beta} \diamond e_1$ and since y is not in normal form then `M( R42) OK`

M(x) Lemma 4.3.3