Ein Notation in Diderot

Charisee Chiw

April 2014

Abstract

Research scientists and medical professionals use imaging technology to measure a wide variety of biological and physical objects. The increasing sophistication of imaging technology creates demand for equally sophisticated computational techniques to analyze and visualize the image data. Diderot is a domain-specific programming language for scientific visualization. Diderot supports a high-level model of computation based on continuous tensor fields. These tensor fields are reconstructed from discrete image data using separable convolution kernels or from applying differentiation on fields. Diderot's surface language is similar to its mathematical counterpart. For example, the tensor and field operations $\nabla, \cdot, \circledast$ are supported in Diderot.

The current direct-style Diderot compiler is limited by the type of operations it supports. We developed an index notation called "EIN", which is based on index notation but the syntax details are specific to our needs. To address these compiler limitations, we introduce the design and implementation of EIN notation in the Diderot compiler. EIN notation is inspired by Einstein index notation with some extensions to represent the algorithms used in image analysis. The notation allows a concise and expressive way to represent a diverse set of tensor and field operations. We extend the operations supported in the surface language while providing the basis for more operations. The mathematically well-founded compiler can do index-based optimizations supported by tensor calculus to find algebraic identities and simplification, and do domain-specific optimizations.

Contents

Chapter 1

Introduction

Research scientists and medical professionals use imaging technology for a wide variety of physical objects, creating a need for software tools to extract structure and knowledge from the data. That data can represent the underlying biological system. Imaging technology includes computed tomography (CT) and magnetic resonance imaging (MRI), and the hardware can show images of vectors and tensors that represent organs, tissues, brain tissue, and blood flow. The quantity and type of data, however, are not supported optimally by the current tools for image processing.

Diderot is a domain-specific programming language created for scientific visualization and image-analysis that targets applications with large independent sub computations [7]. Many algorithms used for scientific visualization have a large number of independent sub computations, and these techniques require a large quantity of data and computations. Furthermore, scientists may not have the programming knowledge to create an efficient parallel implementation of such techniques. The Diderot language is designed to simplify the parallel methods and allows scientists to develop their own images analysis and visualizations. It also improves performance by supporting efficient execution, especially on parallel platforms.

Diderot supports math notation for computing tensors and fields by providing the ingredients for such operations. Tensors are scalars, vectors, matrices, and outputs to evaluated fields. Applications for Diderot include fiber tractography, particle systems, volume rendering and edge detection. The language improves programmability because of the high level of mathematical programming notation. Additionally, the conceptual transparency of the math involved could be useful for an education context.

Several domain-specific languages exist that provide a similar array of support for scientific computing. Surface language support for common operations allows the programmer to focus on mathematical functions and the compiler to generate code. Spiral is a DSL created for digital signal processing in [18]. TCE is a tensor contraction engine used to represent quantum chemistry in [1]. OptiML is a DSL written for machine learning that utilizes Delite in [24]. Shadie is a DSL for volume visualization in [14]. Some of these DSLs are discussed further in the related works section.

This paper makes several contributions. First, it introduces EIN expressions that are used in the inter representation of the Diderot compiler. This compact notation can help support a larger array of tensor and field operators, and also provide the mathematical basis for more but implementing a new design can provide some challenges. We cover the techniques used to do optimizations based on tensor calculus, which can make the compiler mathematically well-founded.

In Chapter 2, we present the Diderot program structure and design, the mathematical foundation of image-analysis, an overview of the compiler, and standard index notation. Chapter 3 will introduce EIN expressions, demonstrate some advantages, and map the surface language to EIN operators. Chapter 4 will walk through the implementation of EIN expressions in the compiler, optimization techniques, and reconstruction of fields. Chapter 5 will cover code generation limitations, design, and plans. Chapter 6 will discuss related work, Chapter 7 describes future plans and Chapter 8 is the conclusion.

Chapter 2

Background

This chapter provides the background needed for this paper. Section 1 will review the Diderot programming language, the type system, computational notation, program structure and parallelism [7]. Diderot computes operations on continuous tensor fields and supports a variety of tensor operations. We plan to represent these tensor and field operations with an index-inspired notation in the compiler. Section 2 maps the formal definition of tensor and field operators to Einstein index notation. Section 3 offers an overview of the Diderot compiler as well as its limitations Section 4 provides the mathematical background for the reconstruction of fields and the differentiation of fields.

2.1 Diderot

Diderot is a parallel domain specific programming language for scientific visualization and image analysis. Diderot targets applications that require a high intensity for many independent subcomputations. Diderot uses C-like syntax on the surface language to represent mathematical types and operators. This section explores the type system, computational notation and program model.

2.1.1 Types

Diderot supports key types that are needed to express scientific visualization algorithms. The basic concrete values include strings, Booleans, integers and tensors. The type $tensor[\sigma]$ is a tensor with the shape σ. The order of the tensor is the length of σ (i.e., the number of dimensions). A 0-order tensor and a 1-order tensor are specified as $tensor[]$ and $tensor[d]$ respectively. Common tensor types such as 0-order tensors (or scalars) and 1-order tensors (or 2-d,3-d vectors) can be referred to as real, vec2, and vec3.

Diderot provides three different abstract types: images, kernels, and fields. An $image(d)[\sigma]$ has dimension d and shape σ. The image type represents image data. A $kernel\#k$ is used to define continuous reconstruction on discrete data. Kernels are referred here in terms of their continuity class C^k where k is the number of continuous derivatives. Diderot provides references to kernels in the surface language. Some kernels include Tent, a class C^0 kernel, Catmull-Rom cubic spline "ctmr", a class C^1 kernel, and "cubic B-spline3", a class C^2 kernel. Computations directly on images and kernels are unsupported. A field is an abstract representation of functions from vector space to tensors. Fields are defined in the surface language as the convolution between an image and a kernel. $Fields\#k(d)[\sigma]$ represents a continuous tensor field. k is the number of continuous

```
field#1(2)[] f = ctmr ⊛ load(‘‘ddro.nrrd");

strand sample (int ui, int vi) {
  output vec2 pos = ⋯;
  // set isovalue to closest of 50, 30, or 10
  real f0 = 50.0 if f(pos) >= 40.0
            else 30.0 if f(pos) >= 20.0
            else 10.0;
  int steps = 0;
  update {
    if (!inside(pos, f) || steps > stepsMax)
      die;
    vec2 grad = ∇f(pos);
    vec2 delta =  // the Newton-Raphson step
      normalize(grad) * (f(pos) - f0)/|grad|;
    if (|delta| < epsilon)
      stabilize;
    pos = pos - delta;
    steps = steps + 1;
  }
}
```

Figure 2.1: Detecting isocontours

derivatives for the d-dimensional field with shape of the range σ. A field is a function that maps d-dimensional shape to a tensor with shape σ.

2.1.2 Computational notation Tensor and Field Expressions

Diderot supports mathematical notation for computing tensors and fields. The syntax uses Unicode characters to represent constants like π. Tensor manipulations include vector norm $|u|$ and $transpose(u)$. Operations between tensors include basic arithmetic operations, addition, subtraction, and division. Additionally, Diderot supports the product between tensors such as the inner product $u \cdot v$, double dot product $u : v$, cross product $u \times v$, and outer product $u \otimes v$.

Diderot supports various operations on continuous tensor fields. Field values are evaluated by convolving image date with kernel. Convolution is supported as $img \circledast bspln3$. A subset of tensor operations are lifted to work fields, such as addition, subtraction, division, and scaling. Diderot supports the differentiation of fields. The direct-style compiler supports ∇ (scalar fields) and $\nabla\otimes$ (higher-order tensor fields). The probing of a field at a point is represented as $F(x)$. There are also tests to see if a point x lies within the domain of a field (inside(x,F)). Section 2.2.2 provides the mathematical basis for how to evaluate the probing and the differentiation of fields.

2.1.3 Program Structure

This section is the overview of the computational model. A Diderot program is implemented by three sections: global definitions, strand definitions and the initialization. Figure 2.1 is an example of Diderot code to detect isocontours.

Global definitions

Global declarations have global values, fields and inputs to the program. These variables are readable by all strands and are not mutable. An input variable can be set outside the program and have a default values. The Diderot compiler synthesizes code and allows command-line setting of input variables. The load function is only used in this section of the program. The load operator gets the image data at the location indicated and the type of the img is checked when the image is loaded. The compiler maps the image values to reals. Global variable field F (line 1) is defined with convolution of kernels and images. In this case, a Field is created from the image at "ddro.nrrd " and kernel ctmr.

Strands

A strand definition is defined in the computational agents that implement the program. The strands maintain the computational core of the application. Each stand has parameters, a state and an update method. A strand has its own independent state, either active, stable, or dead, and never sees the state of another strand. The update method is called only when the strand is active. A strand can also have a stabilize method that offers conditions for stability. Otherwise, a strand stabilizes when it is available and not going to change. The methods can include local variables with scoping rules that are similar to C's. Some strand variables have output variables which report the strands state at the program's output.

Diderot targets algorithms that use a large number of independent computations. The update method for each strand is in lines 10-line 20 in Figure 2.1. Each strand updates its position using a Newton-Raphson step. If the position is outside the field or has taken the maximum number of steps, then the strand dies(line 11-line 12). If the Newton-Raphson step is less, then some epsilon than the strand stabilizes (line 16-line 17). Otherwise the strand stays active.

Line integral convolution(LIC) as described in [6] is another visualization technique that requires a large number of computations. This technique attempts to visualize a vector field by blurring a noise texture. Streamlines are paths everywhere tangent to the vector field, computed by numerical integration. For each pixel in the output, we define a strand that computes a streamline. Diderot assigns one strand per path in fiber tractography algorithms and one strand per ray in volume renders.

Initialization

The initialization section of the program defines the initial set of strands. The set of strands are initialized with parameters at the start of the program. The comprehension syntax is similar to Haskell or Python. A grid of values can also be specified by using a "[]" and the structure is preserved in the output.

2.1.4 Parallelism Model

Diderot uses a bulk synchronous parallelism model as mentioned in [22]. At each iteration, all the strands execute in one step. During the step, each strand's update method is called. There may be some time between the end of a strand's update computation and the next step. At the end of the step, each strand either stabilizes or dies. Diderot only looks at the state of each strand at

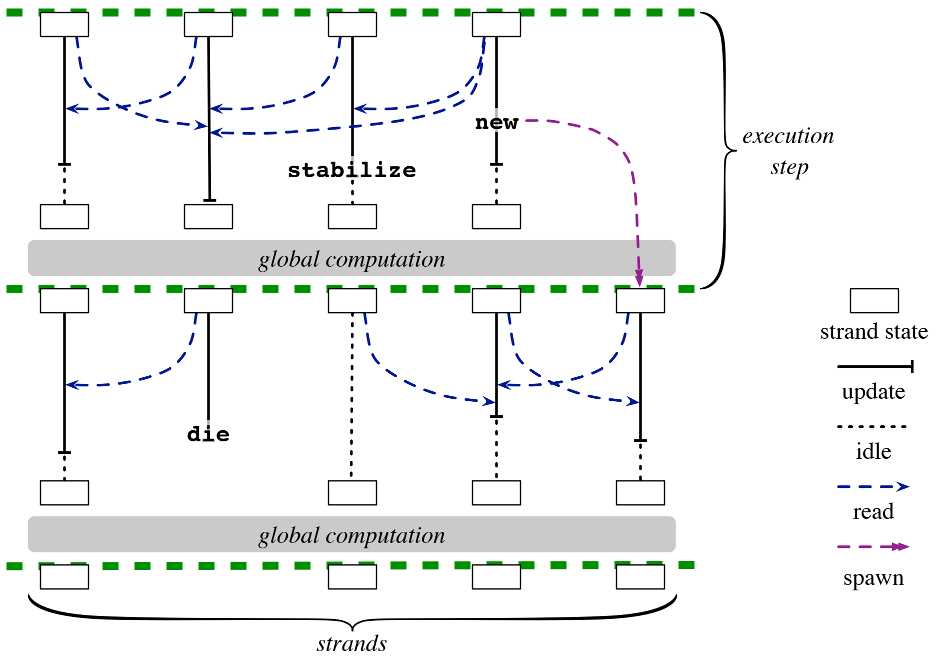

Figure 2.2: Diderot Parallelism Model

the beginning of the next step. When all the strands stabilize or die then there are no more steps taken.

2.1.5 Examples

Figure 2.3 is an example of curvature code. Curvature is a physical property that computes the change in normal due to motion along the surface. The technique requires computations on derived fields, differentiation of fields, and tensor operations.

```
field2(3)[]# F = bspln3 ⊛ load(a.nrrd);
field0(2)[3] RGB = tent ⊛ load(xfer);
···
  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;
    // find material RGBA
    vec3 matRGB =
        RGB([max(-1.0, min(1.0, 6.0*k1)),
             max(-1.0, min(1.0, 6.0*k2))]);
    ···
  }
```

Figure 2.3: Computing the surface color based on implicit surface curvatures

2.2 Einstein Index Notation

Diderot surface language operators can be represented in index notation. Diderot presumes an orthornormal basis, which allows us to take advantage of certain identities. It is necessary to refer to the components of tensors for these identities. Section 2.2.1 will introduce Einstein index notation for basic tensor and field expressions, and Section 2.2.2 will review concepts supported by an orthonormal basis. Then, Section 2.2.3 and 2.2.4 present various tensor operators and higher-order field operators in index notation. The paper uses the following notation

u,**v**,and **w** Vectors **T**,**A**, and **B** General Tensors

φ Scalar Field **F**,**G** General Fields

M_{ij} Index Notation

2.2.1 Index Notation

Einstein index notation or the summation convention can be used to concisely describe tensor and field operations. The subscripts are denoted with lower-case letters ijk. An index is either a summation index or a free index. Each are bound over a range. Consider the expression for 3-by-3 matrix **M**,

$$\boldsymbol{M}_{ij} = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{bmatrix}$$

Indices i, j are free indices. A dummy index is repeated exactly twice and has an implicit summation over of an index. Consider the expression,

$$M_{ii} = M_{00} + M_{11} + M_{22}$$

Index i is a summation index. A multi-index, or a list of indices is denoted with α, μ, ν. T_α can be a tensor of any arbitrary shape, including **M**.

The relationship between the indices can also be expressed with $\mathcal{E}_{ijk}$ and δ_{ij}. $\mathcal{E}_{ijk}$ is used when computing the tensor product between tensors and the Curl. The δ_{ij} expression stands for the kronecker delta function. It can be used when defining the identity.

$$\mathcal{E}_{ijk} = \begin{cases} 1 & \textbf{cyclic} \\ -1 & \textbf{anti-cyclic} \\ 0 & \textbf{otherwise} \end{cases} \quad \text{and} \quad \delta_{ij} = \begin{cases} 1 & \textbf{i=j} \\ 0 & \textbf{otherwise} \end{cases}$$

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

$$\mathcal{E}_{ijk}\mathcal{E}_{ilm} \Longrightarrow \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl} \tag{2.1}$$

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

$$\delta_{ij}T_j \Longrightarrow T_i \qquad \delta_{ij}F_j \Longrightarrow F_i \qquad \nabla_j\delta_{ij} \Longrightarrow \nabla_i \tag{2.2}$$

2.2.2 Orthornormal basis

Let us define an orthonormal basis β with unit basis vectors as $b_i, b_j, ...$ The following properties can be found in [15]. Each basis vector is linearly independent and normalized such that

$$b_i \cdot b_j = \begin{cases} 1 & \textbf{i=j} \\ 0 & \textbf{otherwise} \end{cases} \tag{2.3}$$

$$b_i \cdot b_j = \delta_{ij} \tag{2.4}$$

Specifically, for a 3-d space we have

$$b_i \times b_j = \mathcal{E}_{ijk} b_k \tag{2.5}$$

Then any vector $\mathbf{u}$ can be defined by a linear combination of these basis vectors and have the following properties.

$$\mathbf{u} = \Sigma_i u_i b_i \tag{2.6}$$

$$\mathbf{u} \cdot b_j = u_j \tag{2.7}$$

A matrix can be expressed as

$$A_{ij} = b_i \cdot A b_j \tag{2.8}$$

Any tensor of order or rank can be expressed as

$$A_{i_1, i_2, .. i_n} b_{i_1} \otimes b_{i_2} \otimes .. \otimes b_{i_n} \tag{2.9}$$

2.2.3 Tensor Operations

Simple tensor operators such as scaling and addition are described in Figure 2.4.

Surface Language	Index Representation
-A	$-A_\alpha$
A+ B	$A_\alpha + B_\alpha$
A- B	$A_\alpha - B_\alpha$
s* A	sA_α
A/s	$\frac{A_\alpha}{s}$
Identity	δ_{ij}
Trace(**A**)	A_{ii}

Figure 2.4: Simple Tensor Operators in Index Notation

This section will illustrate examples of tensor operators $(\cdot, :, \otimes, \times)$ in index notation. Figure 2.5 summarizes these operators. Some additional properties for tensors are as follows

$$(\mathbf{u} \otimes \mathbf{v} \otimes \mathbf{w}) : (\mathbf{x} \otimes \mathbf{y}) = (\mathbf{v} \cdot \mathbf{x})(\mathbf{w} \cdot \mathbf{y})\mathbf{u} \tag{2.10}$$

$$(\mathbf{u} \otimes \mathbf{v})\mathbf{w} = \mathbf{u}(\mathbf{v} \cdot \mathbf{w}) \tag{2.11}$$

Tensor Product $[\mathbf{A} \otimes \mathbf{B}]_\beta$
The components of the tensor product between two vectors **A** and **B** can be defined along their orthonormal basis β as

$$b_i \cdot (A \otimes B) b_j$$

Using properties provided by eqns (11,7), we find that

$$b_i \cdot A(B \cdot b_j)$$

$$(b_i \cdot A) B_j$$

$$A_i B_j$$

The tensor product is expressed in index notation as $A_i B_j$

Cross Product $[\mathbf{A} \times \mathbf{B}]_\beta$
This operation can be written as

$$\Sigma_{ij} A_i b_i \times B_j b_j$$

$$\implies \Sigma_{ij} A_i B_j (b_i \times b_j)$$

With eqn (5) and an even permutations of (i,j,k) we find that

$$\implies \Sigma_{ij} A_i B_j (\mathcal{E}_{ijk} b_k)$$

$$\implies \Sigma_{jk} A_j B_k \mathcal{E}_{jki} b_i$$

The Cross product is written in index notation as $\mathcal{E}_{ijk} A_j B_k$

Inner Product $[\mathbf{A} \cdot \mathbf{B}]_\beta$
The dot product between two vectors is described as

$$A_i b_i \cdot B_j b_j$$

Taking advantage of property (4,2) we find that

$$\implies \Sigma_{ij} A_i B_j b_i \cdot b_j$$

$$\implies \Sigma_{ij} A_i B_j \delta_{ij}$$

$$\implies \Sigma_i A_i B_i$$

The dot product is expressed in index notation as $A_i B_i$
Generally, the inner product is expressed as $A_{\mu i} B_{i\nu}$

Double-dot Product $[\mathbf{A} : \mathbf{B}]_\beta$
This example illustrates the double-dot product with third-order tensor **A** and a second-order tensor **B**. With properties (10, 4, 2) we find that

$$\begin{aligned}
&\Sigma_{ijklm} A_{ijk} B_{lm} (b_i \otimes b_j \otimes b_k) : (b_l \otimes b_m)\\
&\implies \Sigma_{ijklm} A_{ijk} B_{lm} (b_j \cdot b_l)(b_k \cdot b_m) b_i\\
&\implies \Sigma_{ijklm} A_{ijk} B_{lm} \delta_{jl} \delta_{km} b_i\\
&\implies \Sigma_{ijk} A_{ijk} B_{jk} b_i
\end{aligned}$$

The double-dot product is written in index notation as $A_{\mu jk} B_{jk\nu}$

Surface Language	Index Representation
$\mathbf{A} \cdot \mathbf{B}$	$A_{\mu i} B_{i\nu}$
$\mathbf{A} : \mathbf{B}$	$A_{\mu ij} B_{ij\nu}$
$\mathbf{A} \otimes \mathbf{B}$	$A_i B_j$
$\mathbf{A} \times \mathbf{B}$	$\mathcal{E}_{ijk} A_j B_k$

Figure 2.5: Tensors Operations

2.2.4 Field Operators

A field can be written as **F** or **G**, and a scalar field with φ. In Diderot, a subset of tensor operators have been lifted to work on fields. A selection of those are shown in index notation in Figure 2.6. Diderot supports four basic differentiation operations;$(\nabla, \nabla\cdot, \nabla\times, \nabla\otimes)$. This section will map the

Fields	
Surface Language	Index Representation
s+**F**	$s + \varphi$
F-s	$\varphi - s$
-**F**	$-F_\mu$
F+**G**	$F_\mu + G_\mu$
F-**G**	$F_\mu - G_\mu$
s***F**	sF_μ
$\frac{\mathbf{F}}{s}$	$\frac{F_\mu}{s}$

Figure 2.6: Tensor Operators on Fields

formal definition of these operators to index notation. Figure 2.7 summarizes the representation of these operations in index notation. Mathematicians and physicists define differentiation with the following set of equations [15, 16]

$$[\nabla \phi]_\beta = \Sigma_i \frac{\partial}{\partial x_i} \phi \quad b_i \tag{2.12}$$

$$[\nabla \cdot f]_\beta = \Sigma_i \frac{\partial}{\partial x_i} f \cdot b_i \tag{2.13}$$

$$[\nabla \times f]_\beta = \Sigma_i b_i \times \frac{\partial}{\partial x_i} f \tag{2.14}$$

$$[\nabla \otimes f]_\beta = \Sigma_i \frac{\partial}{\partial x_i} f \otimes b_i \tag{2.15}$$

Gradient $[\nabla \phi]_\beta$
As described in equation (12) $\Sigma_i \frac{\partial}{\partial x_i}\phi \quad b_i$
By definition of basis vectors, we can expand this expression

$$[\frac{\partial}{\partial x_1}\phi, \frac{\partial}{\partial x_2}\phi, ..]$$

The index notation representation of the Gradient is $\frac{\partial}{\partial x_i}\phi$.

Divergence $[\nabla \cdot \mathbf{F}]_\beta$
Consider a vector field $\mathbf{u}$ in equation (13) by using properties (4,2).

$$\implies \Sigma_{ij} \frac{\partial}{\partial x_i} u_j \quad b_j \cdot b_i$$

$$\implies \Sigma_{ij} \frac{\partial}{\partial x_i} u_j \quad \delta_{ij}$$

$$\implies \Sigma_i \frac{\partial}{\partial x_i} u_i$$

This is written in index notation as $\frac{\partial}{\partial x_i} F_i$.
Generally, we can express a tensor field $\mathbf{A}$ with shape $[\alpha_i, \alpha_j..\alpha_n$,k] with

$$\Sigma_{\alpha,k} A_{\alpha_i \alpha_j ....\alpha_n k} b_{\alpha_i} \otimes .. \otimes b_{\alpha_n} \otimes b_k$$

in equation (13) as

$$\implies \Sigma_{\alpha,k,l} \frac{\partial}{\partial x_l} (\Sigma_{ij..n} A_{\alpha_i ...\alpha_n k} b_{\alpha_i} \otimes .. \otimes b_{\alpha_n} \otimes b_k) b_l$$

$$\implies \Sigma_{\alpha,k,l} \frac{\partial}{\partial x_l} (\Sigma_{ij..n} A_{\alpha_i ...\alpha_n k} b_{\alpha_i} \otimes .. \otimes b_{\alpha_n} (b_k \cdot b_l)$$

$$\implies \Sigma_{\alpha,k,l} \frac{\partial}{\partial x_l} (\Sigma_{ij..n} A_{\alpha_i ....\alpha_n k} b_{\alpha_i} \otimes .. \otimes b_{\alpha_n} (\delta_{kl})$$

$$\implies \Sigma_{\alpha,k} \frac{\partial}{\partial x_k} (\Sigma_{ij..n} A_{\alpha_i ...\alpha_n k} b_{\alpha_i} \otimes .. \otimes b_{\alpha_n}$$

This is written in index notation as $\frac{\partial}{\partial x_k} F_{\alpha k}$.

Curl $[\nabla \times \mathbf{F}]_\beta$
Consider property (5) with vector field **u** and equation (15) becomes

$$\implies \Sigma_{ij} b_i \times \frac{\partial}{\partial x_i} u_j \quad b_j$$

$$\implies \Sigma_{ij} \frac{\partial}{\partial x_i} u_j \quad b_i \times b_j$$

$$\implies \Sigma_{ij} \frac{\partial}{\partial x_i} u_j \mathcal{E}_{ijk} b_k$$

Using tmp variables $i \implies n \implies j, j \implies o \implies k,$ and $k \implies m \implies i$

$$\implies \Sigma_{ij} \frac{\partial}{\partial x_j} u_k \mathcal{E}_{jki} b_i$$

The 3-d Curl is written in index notation as $\frac{\partial}{\partial x_j} F_k \mathcal{E}_{ijk}$.

Higher-order differentiation $[\nabla \otimes \mathbf{F}]_\beta$
Equation (15) describes this operation for a generic field f

$$\Sigma_i \frac{\partial}{\partial x_i} f \otimes b_i$$

This operator on a vector field is written as

$$\Sigma_{ij} \frac{\partial}{\partial x_j} f_i \quad b_i \otimes b_j$$

The general representation of a tensor field in equation (15) can be expressed as

$$\Sigma_{\alpha k} \frac{\partial}{\partial x_k} (\Sigma_\alpha A_{\alpha_i \alpha_j \ldots . \alpha_n} b_{\alpha_i} \otimes .. \otimes b_{\alpha_n}) \otimes b_k$$

In index notation we can generally represent the shape of field **f** with α. This operation is written in index notation as $\frac{\partial}{\partial x_k} F_\alpha$.

Fields	
Surface Language	Index Representation
$\nabla \mathbf{F}$	$\frac{\partial}{\partial x_i} \varphi$
$\nabla \cdot \mathbf{F}$	$\frac{\partial}{\partial x_i} F_{\alpha i}$
$\nabla \otimes \mathbf{F}$	$\frac{\partial}{\partial x_i} F_\alpha$
$\nabla \times \mathbf{F}$	$\frac{\partial}{\partial x_j} F_k \mathcal{E}_{ijk}$

Figure 2.7: Differentiation Operators in index Notation

2.3 The Diderot Compiler

2.3.1 Current Organization of the Compiler

The compiler is organized into three different “phases”. They are referred to as the front-end, optimization and lowering, and code generation. The front-end level includes parsing, type checking and simplification. The optimization and lowering phase includes three intermediate representations: High-IL, Mid-IL and Low-IL. Finally, the code generation phase produces block structured code. Figure 2.8 illustrates the organization of the compiler and the optimization and lowering phase is highlighted.

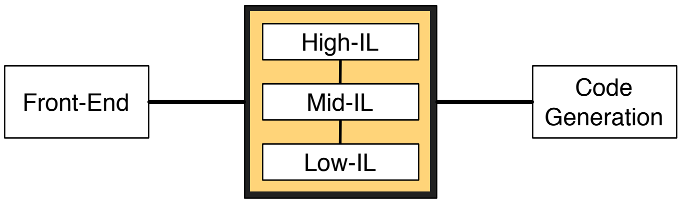

Figure 2.8: Diderot Compiler Organization

2.3.2 How it Currently Works

The front-end phase scans and parses the Diderot source program. The type-checker forms the parse tree to a typed AST representation. Most of the operators supported in Diderot have instances of multiple types. For example, the addition operator is used between integers, reals, tensors and fields. An addition operator for each type of tensor and field would be overly complicated. Instead, the type checker has polymorphic operators that can work on arbitrary size tensors and fields. At the end of the type checking process, the operators have specific monotypes. The typed AST representation produces the Simple AST representation. At the end of this phase, operators are applied only to variables and temporary values are created.

The next phase is the optimization and lowering that has three intermediate representations. Each intermediate representation has a block-structured Static Single Assignment representation and has the same control-flow graph(CFG) [9], but they differ in a few ways. Each representation has its own types, operations, and optimizations. The transition between the representations creates lower-level operations from their counterparts.

High-IL is the first intermediate representation in the lowering and optimization phase. The High-IL optimizations lift static computations to a global level and eliminate dynamic field, kernel and image values. The differentiations of fields is pushed down to the kernels.

Mid-IL supports a different representation of fields. In the transitions of High-IL to Mid-IL, the fields and probes are compiled away into lower-level code. The Mid-IL representation supports transformation between world-space to image-space positions, loading voxels from images, and evaluating kernels. Higher-order tensors are flattened. Mid-IL operators expand kernel evaluation into their piecewise polynomial equivalent and expand the address calculation for voxel addresses.

Low-IL supports basic vector and scalar operations and memory objects.

Finally, Low-IL is translated to Tree-IL in the code generation phase. This process turns SSA assignment into expression trees and turns the CFG into block structured code. Separate backends are created to work for different target interfaces which includes sequential C code with vector extensions, parallel C code, OpenCL, and CUDA. The target representation is used to augment the code with type definitions and runtime support. The output is passed to the host system's compiler.

2.3.3 Limitations

The direct-style notation used in the compiler can provide some challenges in expressing operations. The compiler represents a fixed set of operators. There are several desirable operations that the direct-style compiler does not support such as $\nabla\times$, and $\nabla\cdot$. Adding support for these operators requires a number of intermediate operators. Each additional field operator requires a case-by-case analysis in order support necessary field normalization.

The Diderot surface language has a promise of generality that is not supported by the implementation. The compiler only supports a subset of shapes for tensor operators. To illustrate this consider the inner product. The Direct-Style operators for the inner product include "dot", "MulVecMat", "MulMatVec", "MulMatMat","MulVecTen3", "MulTen3Vec." The compiler could not compute the inner product between higher-order tensors, because it lacks the representation.

2.4 Image Analysis

Tensor fields are defined by reconstruction of discrete image data with separable kernels. Fields combine kernels and discrete data to create smooth tensor fields over continuous world-space abstract functions that are evaluated at any location to produce a tensor. Fields are functions from R^d to tensors. The continuous fields are important because of the features that lie between pixels.

Section 2.4.1 introduces the notation used to describe the reconstruction of fields. Section 2.4.2 examines the reconstruction of fields from image data and kernel and explore the process that allows Diderot to lift tensor operations to work on fields. Section 2.4.3 will provide mathematical background for the differentiation of fields. The existing design in Diderot supported one type of differentiation of fields. This section introduces support for other higher order field operators, which ultimately expands the type of operations supported in the surface language.

2.4.1 Notation

Diderot describe Fields in terms of their domain range, and continuity of function. A field is defined as an image convolved with a kernel. In this section, we define the terms "pure fields" and "derived fields". A pure field is simply a field reconstructed with an image and kernels. A derived field is a field created as a result of field operations. Section 2.2 introduced the notation used to write fields directly and differentiation operations in index notation. In order to represent field reconstruction, we introduce several new expressions in the table below.

W	world-space
I	image-space
$\mathbf{V}$	image data
$\mathbf{p}$	position in W
$\mathbf{x}$	position in I
$\mathbf{n}$	integer position of x
$\mathbf{f}$	fraction position of x
h	kernel
$\mathbf{s}$	support of h
$\mathbf{M}^{-1}$	transform matrix from W to I
$\mathbf{P}$	transform matrix from I to W

An image is discrete data that is sampled across a regular grid. An image is denoted with $\mathbf{V}$. Kernels are independent of data. A piecewise polynomial kernel is represented with h and support with $\mathbf{s}$. The first differentiation of a kernel is represented with h'. A field is probed in the surface language at a world-space position $\mathbf{p}$. $\mathbf{M}^{-1}$ is the transformation matrix from world-space to image-space. The world-space position $\mathbf{p}$ is transformed into an image-space position $\mathbf{x}$. The reconstruction of the field is represented with the integer and fraction position of $\mathbf{x}$. If differentiation is used then we use transform matrix $\mathbf{P}$, where $\mathbf{P}$=$[\mathbf{M}^{-T}]_{IW}$

This section introduces differentiation on reconstructed fields. The examples need to refer to specific components on fields. For that purpose, we denote a specific axis or component with an underline. For instance, a 3-d vector field $\mathbf{F}$ can be referred to directly in index notation or by it's components as

$$\mathbf{F} \qquad \Longrightarrow F_i \qquad \Longrightarrow \begin{bmatrix} F_{\underline{0}} \\ F_{\underline{1}} \\ F_{\underline{2}} \end{bmatrix}.$$

The differentiation expression $\frac{\partial}{\partial x_i}$ is used in every index notation differentiation expression. When illustrating specific components, this expression can get cluttered. For this purpose, we define constants ($\underline{x}$, $\underline{y}$,$\underline{z}$) to represent a specific partial derivative. For a 3-d space, the specific components of the partial derivative can be expressed as

$$\frac{\partial}{\partial x_i} \Longrightarrow \frac{\partial}{\partial x_{\underline{0}}}, \frac{\partial}{\partial x_{\underline{1}}}, \frac{\partial}{\partial x_{\underline{2}}} \Longrightarrow \frac{\partial}{\partial \underline{x}}, \frac{\partial}{\partial \underline{y}}, \frac{\partial}{\partial \underline{z}}$$

The upper subscript denotes the level of differentiation.

$$\frac{\partial}{\partial x_{\underline{1}}^2} \Longrightarrow \frac{\partial}{\partial \underline{y}^2}$$

2.4.2 Reconstruction of Pure Fields

A field application gets compiled down into code where world-space coordinates are mapped to image space. Then, the image-space convolves the image values in the neighborhood of the position as shown in Figure 2.9. Reconstructing continuous signals from discrete data could involve linear interpolation or convolution with more complicated kernels. This section will show the reconstruction of pure fields. Probing a field produces a tensor, which is the shape of the range of the image

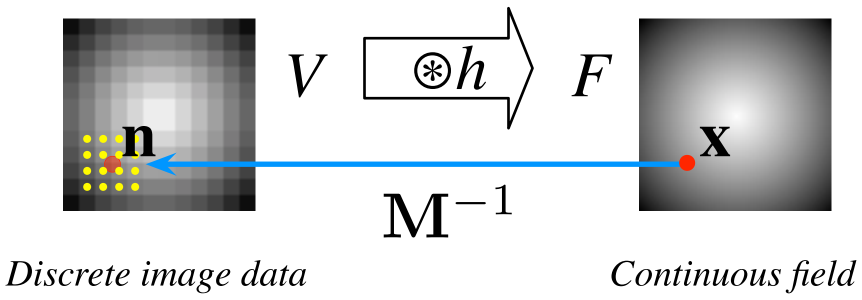

Figure 2.9: Probe of a scalar field

field. The neighborhood of positions will range from 1-**s** to **s**, where **s** is the support of the kernel. This section looks at the reconstruction of a 1-D scalar field and a 2-D vector field.

Scalar field A 1-D scalar field is written in Diderot syntax as

```
field#k(1)[] φ = V ⊛ h;
real s= φ(p);
```

The first step is to transform the world-space position to image-space.

$$\begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$

Take the integer and fractional position as

$$\mathbf{n} = \lfloor \mathbf{x} \rfloor$$

$$\mathbf{f} = \mathbf{x} - \mathbf{n}$$

The probe is expanded as

$$\Longrightarrow \textstyle\sum_{i=1-s}^{s} \mathbf{V}(\mathbf{n}+i)h(\mathbf{f}-i)$$

The kernel is represented as a piecewise polynomial with 2**s** segments. It is computed from $1-\mathbf{s}$ to **s**. If the position is outside that range the kernel polynomial is computed to zero.

Vector field A 2-D vector field is written in the Diderot syntax as

```
field# k(2)[2]F = V ⊛ h;
tensor [2] t= F(p);
```

We transform the world-space position **p** to image space position **x** as

$$\mathbf{x} = \mathbf{M}^{-1}\mathbf{p}$$

$$\mathbf{n} = \lfloor \mathbf{x} \rfloor$$

$$\mathbf{f} = \mathbf{x} - \mathbf{n}$$

The field is expanded as

$$\Longrightarrow \begin{bmatrix} \sum_{ij:1-s}^{s} & \mathbf{V}_{\underline{1}}([\mathbf{n}+i,\mathbf{n}+j]) & h(\mathbf{f}-i)h(\mathbf{f}-j) \\ \sum_{ij:1-s}^{s} & \mathbf{V}_{\underline{2}}([\mathbf{n}+i,\mathbf{n}+j]) & h(\mathbf{f}-i)h(\mathbf{f}-j) \end{bmatrix}$$

2.4.3 Reconstruction of Derived Fields

A derived field is created when operations are applied to fields. When a tensor operator is lifted to operate on a derived field then the expansion is much like it is shown in the previous section. The probed fields are evaluated to a tensor result, and then the arithmetic operations are used. In applications such as edge detection, volume shading or curvature, the differentiation of fields would be computed. The differentiation of fields are more complicated and are derived differently than pure fields.

This section introduces differentiation on reconstructed fields for operators $\nabla, \nabla\cdot, \nabla\otimes, \nabla\times$. While two different differentiation operations can produce the same shape, their method of computation can be different. The resulting shape will depend on the shape of the range of the image field and the level of differentiation. Section 2.2 provided the basis of mapping the formal definition of these operators to index notation. Presented here is an example of each operator by specifying their components. The differentiation operators will be introduced here in direct-style notation, index notation, and then by representing the field components.

The ∇ operator is only used on scalar fields and produces a vector field. A 2-d expression of the operator

$$\nabla\varphi \Longrightarrow \tfrac{\partial}{\partial x_i}\varphi \Longrightarrow \begin{bmatrix} \frac{\partial}{\partial \underline{x}^1 y^0} \\ \frac{\partial}{\partial \underline{x}^0 \underline{y}^1} \end{bmatrix} \varphi$$

The $\nabla\times$ operator is only used on vector fields. The resulting tensor depends on the dimension of the field. A 3-d expression of the operator is

$$\nabla \times \mathbf{F} \Longrightarrow \frac{\partial}{\partial x_j} F_k \mathcal{E}_{ijk} \Longrightarrow \begin{bmatrix} \frac{\partial}{\partial \underline{y}} F_{\underline{2}} - \frac{\partial}{\partial \underline{z}} F_{\underline{1}} \\ \frac{\partial}{\partial \underline{z}} F_{\underline{0}} - \frac{\partial}{\partial \underline{x}} F_{\underline{2}} \\ \frac{\partial}{\partial \underline{x}} F_{\underline{1}} - \frac{\partial}{\partial \underline{y}} F_{\underline{0}} \end{bmatrix}$$

The $\nabla\cdot$ operator can be used on any higher-order fields and reduces the order of the tensor field by one. This operator can be expressed on a 2-d vector field as

$$\nabla \cdot \mathbf{F} \Longrightarrow \frac{\partial}{\partial x_i} F_{\alpha i} \Longrightarrow \frac{\partial}{\partial x_i} F_i \Longrightarrow \frac{\partial}{\partial \underline{x}^1 \underline{y}^0} F_{\underline{0}} \quad + \frac{\partial}{\partial \underline{x}^0 \underline{y}^1} F_{\underline{1}}$$

The $\nabla\otimes$ operators can be used on any higher-order field, and increases the order of the tensor field by one. This operator can be expressed on a 2-d field vector field as

$$\nabla \otimes F \Longrightarrow \frac{\partial}{\partial x_j} F_\alpha \Longrightarrow \frac{\partial}{\partial x_j} F_i \Longrightarrow \begin{bmatrix} \frac{\partial}{\partial \underline{x}^1 \underline{y}^0} F_0 & \frac{\partial}{\partial \underline{x}^0 \underline{y}^1} F_0 \\ \frac{\partial}{\partial \underline{x}^1 \underline{y}^0} F_1 & \frac{\partial}{\partial \underline{x}^0 \underline{y}^1} F_1 \end{bmatrix}$$

The product of $\frac{\partial}{\partial x_\mu}$ and $\frac{\partial}{\partial x_\nu}$ expressions is $\frac{\partial}{\partial x_{\mu\nu}}$. The expression can also be evaluated by summing the components.

$$\frac{\partial}{\partial \underline{x}^i \underline{y}^j} \quad \frac{\partial}{\partial \underline{x}^{i'} \underline{y}^{j'}} \quad = \frac{\partial}{\partial \underline{x}^{i+i'} \underline{y}^{j+j'}}$$

The following section will show several examples using the above operators. While the intermediate steps to reconstruct fields may differ, each example shares two common steps: the first and last. The first step is to transform world-space position $\mathbf{p}$ to image-space and then to use world-space positions $\mathbf{n}$ and $\mathbf{f}$. The last step is to transform the result of the computation back to world-space.

When taking the differentiation of fields, the result is a value in image-space and not world-space [16]. Therefore, the result of field differentiation must be transformed back to world-space with transformation matrix $\mathbf{P}$. A multiplication by $\mathbf{P}$ is needed for each index of convolution measured in the derivatives. Consider the result of a third-order differentiation of a field, tensor T_{lmn} with three indices lmn in image-space position. This requires three multiplications by matrix P expressed in index notation as

$$T_{lmn} P_{il} P_{jm} P_{kn}$$

The first index in P is in world-space and the second index is in image-space. Generally, tensor $T_{\mu\alpha}$ with indices α in image-space gets transformed back to image space by

$$T_{\mu\alpha_0\alpha_1..\alpha_n} P_{\beta_0\alpha_0} P_{\beta_1\alpha_1}..P_{\beta_n\alpha_n}$$

We show this transformation in the following two examples, Gradient and Hessian but omit in the rest.

Gradient The Gradient is the first order-approximation of the local variation of the function, or field itself. The Gradient of a field is used to compute the surface normal. The Gradient of a 3-d scalar field is written in Diderot syntax as

```
field#k(3)[3] $\varphi = V \circledast h$;
tensor [3] t= $\nabla\varphi(p)$;
```

Expressed as

$$\begin{array}{lc} t = & (\nabla\varphi)@\mathbf{p} \\ \Longrightarrow & (V \circledast \nabla h)@\mathbf{p} \\ \Longrightarrow & \left(\mathbf{V} \circledast \begin{bmatrix} \frac{\partial}{\partial \underline{x}^1 \underline{y}^0 \underline{z}^0} \\ \frac{\partial}{\partial \underline{x}^0 \underline{y}^1 \underline{z}^0} \\ \frac{\partial}{\partial \underline{x}^0 \underline{y}^0 \underline{z}^1} \end{bmatrix} h\right)@\mathbf{p} \end{array}$$

Reconstructed as

$$\Longrightarrow \begin{bmatrix} \sum_{i,j,k:1-s}^{s} & \mathbf{V}([\mathbf{n}+i,\mathbf{n}+j,\mathbf{n}+k]) & h'(\mathbf{f}-i)h(\mathbf{f}-j)h(\mathbf{f}-k) \\ \sum_{i,j,k:1-s}^{s} & \mathbf{V}([\mathbf{n}+i,\mathbf{n}+j,\mathbf{n}+k]) & h(\mathbf{f}-i)h'(\mathbf{f}-j)h(\mathbf{f}-k) \\ \sum_{i,j,k:1-s}^{s} & \mathbf{V}([\mathbf{n}+i,\mathbf{n}+j,\mathbf{n}+k]) & h(\mathbf{f}-i)h(\mathbf{f}-j)h'(\mathbf{f}-k) \end{bmatrix}$$

Transform the result back to world-space

$$t' = \mathbf{P} \cdot t$$

Hessian The Hessian is a first-order approximation of the local variation of the Gradient. This increases the order of the resulting tensor by two. The Hessian of a 2-d scalar field is written in Diderot syntax as

```
Field#k(2)[] $\varphi = V \circledast h$
tensor [2,2] $t = (\nabla \otimes \nabla\varphi)@$
```

Then goes through the following derivations

$$\begin{array}{lc} t = & (\nabla \otimes \nabla\varphi)@\mathbf{p} \\ \Longrightarrow & (\mathbf{V} \circledast \nabla \otimes \nabla h)@\mathbf{p} \\ \Longrightarrow & \left(V \circledast \begin{bmatrix} \frac{\partial}{\partial \underline{x}^2 \underline{y}^0} & \frac{\partial}{\partial \underline{x}^1 \underline{y}^1} \\ \frac{\partial}{\partial \underline{x}^1 \underline{y}^1} & \frac{\partial}{\partial \underline{x}^0 \underline{y}^2} \end{bmatrix} h\right)@\mathbf{p} \\ \Longrightarrow & \begin{bmatrix} \Sigma_{ij}\mathbf{V}([\mathbf{n}+i,\mathbf{n}+j])h''(\mathbf{f}-i)h(\mathbf{f}-j) & \Sigma_{ij}\mathbf{V}([\mathbf{n}+i,\mathbf{n}+j])h'(\mathbf{f}-i)h'(\mathbf{f}-j) \\ \Sigma_{ij}\mathbf{V}([\mathbf{n}+i,\mathbf{n}+j])h'(\mathbf{f}-i)h'(\mathbf{f}-j) & \Sigma_{ij}\mathbf{V}([\mathbf{n}+i,\mathbf{n}+j])h(\mathbf{f}-i)h''(\mathbf{f}-j) \end{bmatrix} \end{array}$$

There are two indices for differentiation so we multiply the result by P twice

$$t' = \mathbf{P} \cdot t \cdot \mathbf{P}^T$$

Divergence The Divergence of a 2-d vector is written as

```
Field#k(2)[2] F =$V \circledast h$
real s= $(\nabla \cdot F)@p$
```

Expressed as

$$\begin{array}{lc} t = & (\nabla \cdot F)@p \\ \Longrightarrow & (\mathbf{V} \circledast (\nabla \cdot h))@\mathbf{p} \\ \Longrightarrow & ((\mathbf{V}_{\underline{0}} \circledast \frac{\partial}{\partial \underline{x}^1 \underline{y}^0} h) + (V_{\underline{1}} \frac{\partial}{\partial \underline{x}^0 \underline{y}^1} h))@\mathbf{p} \\ \Longrightarrow & \Sigma_{ij}\mathbf{V}_{\underline{0}}(\mathbf{n}+i)h'(\mathbf{f}-i)h(\mathbf{f}-j) + \Sigma_{ij}V_{\underline{1}}(\mathbf{n}+i)h(\mathbf{f}-i)h'(\mathbf{f}-j) \end{array}$$

Jacobian The Jacobian is written in Diderot syntax as

```
Field#k(2)[2] $F = V \circledast h$
tensor [2,2] $t = (\nabla \otimes F)$@p
```

and expressed as

$$\begin{array}{lc}
\Longrightarrow & (\nabla \otimes F)@\mathbf{p} \\
\Longrightarrow & (\mathbf{V} \circledast (\nabla \otimes h))@\mathbf{p} \\
\Longrightarrow & \left(\begin{bmatrix} \mathbf{V}_{\underline{0}} \circledast \frac{\partial}{\partial \underline{x}^1 \underline{y}^0} & \mathbf{V}_{\underline{0}} \circledast \frac{\partial}{\partial \underline{x}^0 \underline{y}^1} \\ \mathbf{V}_{\underline{1}} \circledast \frac{\partial}{\partial \underline{x}^1 \underline{y}^0} & \mathbf{V}_{\underline{1}} \circledast \frac{\partial}{\partial \underline{x}^0 \underline{y}^1} \end{bmatrix} h \right) @\mathbf{p} \\
\Longrightarrow & \begin{bmatrix} \Sigma_{ij} \mathbf{V}_{\underline{0}}([\mathbf{n}+i, \mathbf{n}+j]) h'(\mathbf{f}-i) h(\mathbf{f}-j) & \Sigma_{ij} \mathbf{V}_{\underline{0}}([\mathbf{n}+i, \mathbf{n}+j]) h(\mathbf{f}-i) h'(\mathbf{f}-j) \\ \Sigma_{ij} \mathbf{V}_{\underline{1}}([\mathbf{n}+i, \mathbf{n}+j]) h'(\mathbf{f}-i) h(\mathbf{f}-j) & \Sigma_{ij} \mathbf{V}_{\underline{1}}([\mathbf{n}+i, \mathbf{n}+j]) h(\mathbf{f}-i) h'(\mathbf{f}-j) \end{bmatrix}
\end{array}$$

Laplacian The Laplacian takes the second derivative of a field. The Laplacian operation is represented by a combination of basic differentiation operators. The Laplacian of a 3-d scalar field is written as

```
Field#k(3)[] $\varphi = V \circledast h$
real $s = (\nabla \cdot \nabla \varphi)$@p
```

Expressed as

$$\begin{array}{lc}
\Longrightarrow & (V \circledast \nabla \cdot \nabla h)@\mathbf{p} \\
& \Sigma_{ijk} \mathbf{V}([\mathbf{n}+i, \mathbf{n}+j, \mathbf{n}+k]) h''(f-i) h(f-j) h(f-k) \\
\Longrightarrow & +\Sigma_{ijk} \mathbf{V}([\mathbf{n}+i, \mathbf{n}+j, \mathbf{n}+k]) h(f-i) h''(f-j) h(f-k) \\
& +\Sigma_{ijk} \mathbf{V}([\mathbf{n}+i, \mathbf{n}+j, \mathbf{n}+k]) h(f-i) h(f-j) h''(f-k)
\end{array}$$

Discussion

Probing an arbitrary field at a position can generate complicated code. It requires nested summations that are dependent on the dimension of the fields and order/shape of the result. Additionally, this type of computation has high arithmetic intensity. The design of the intermediate representation of this process needs to be concise and unambiguous.

Chapter 3

Design

This chapter introduces index-inspired EIN expressions. Similar to index notation, EIN expressions can be used to represent tensor and field operations in a mathematically sound and compact way. The EIN expressions are designed to represent the tensor and field operations in the Diderot compiler. The compact representation provides an advantage over Diderot's direct-style compiler and are discussed in Section 3.1. The Diderot language offers operations not traditionally represented in index notation. The notation used in EIN expressions can represent the diverse set of tensor and field operations used in Diderot. Section 3.2 will introduce the EIN expression notation. Section 3.3 will walk through the mapping of surface language operators to EIN operators. Finally, Section 3.4 will discuss the differences between index notation (Section 2.4) and EIN expressions.

3.1 Motivation

The direct-style notation that the compiler uses to express operations on tensor and fields is limiting. Index notation can provide a concise way of representing operations and by doing so expands the capabilities of the compiler. A richer set of operations are supported in the surface language and a mathematically well-founded compiler can do optimizations supported by tensor calculus.

3.1.1 Compact Notation

Diderot's surface language makes a promise of generality that is not delivered by the current compiler. Operators in the existing compiler were expressed with direct-style notation. The code generator is not general enough and can only generate code on a fixed set of tensor operations. Therefore, even though arbitrary-sized tensors can be supported in the surface language, only a subset of possibilities could be represented in the compiler.

Index-inspired notation gives us a way to look at the structure of tensors and fields without having to examine the scalar computations. By using EIN operators in the compiler, we can represent operations on arbitrary-sized tensors. The compiler can break complicated and large EIN operators into a basic set of operations that only work on scalars and vectors. Then, it only needs to generate code for operations between scalars and vectors avoiding the complication of creating an explosion of shape-specific operators in the compiler.

3.1.2 Surface Language Operators

The expressive and concise advantage provided by index notation allows for a richer set of operations that can be supported in the surface language. For example, the previous implementation of Diderot only supported the Gradient $\nabla\varphi$ and the Hessian $\nabla \otimes \nabla\varphi$. The new implementation supports differentiation operators Divergence $\nabla \cdot F$, Laplacian $\nabla \cdot \nabla\varphi$, Curl $\nabla \times F$, and the Jacobian $\nabla \otimes F$. EIN expressions also provide a mathematical basis for lifting tensor operations to work on fields such as the dot product $F \cdot G$ and tensor product $F \otimes G$.

3.1.3 Mathematically Well-Founded Optimizations

The representations and optimizations of index-inspired expressions are mathematically well-founded. The notation expresses explicitly the operations on tensors and fields and has a familiarity to the surface language. Index notation can also provide new optimization opportunities. The direct-style compiler is limited by the rewrites it can do because a term rewrite would require a case-by-case analysis of operators. With index notation, the compiler is smarter about tensor calculus, can also do term rewriting based on the mathematics involved, and can find algebraic identities as a result.

3.2 EIN Expressions

An EIN Operator is an EIN expression with an outer subscript. Figure 3.1 represents EIN operators and expressions. Section 3.2.1 describes the types of subscripts. Section 3.2.2 will introduce the basic tensor, field, image and kernel expressions. Operations such as the differentiation of fields and probing are not traditionally expressed in index notation and require some additional notation. Section 3.2.3 will introduce the notation used to represent fields operations.

3.2.1 Types of Subscript

EIN expressions support two different types of indices: variable index i and constant index $\boldsymbol{c}$ which is represented in bold. μ denotes an index and α denotes a multi-index or list of indices. The variable index can be bounded in two different places; outside the EIN operator or by the summation expression. Take this generic EIN operator

$$\left\langle \sum_{\nu} e \right\rangle_i .$$

where $i \epsilon n$

A variable index bounded by the outer index ranges from 0 to n. A variable index bounded by the summation has range ν. This allows the summation to support indices varying from 0-N in the case of tensor operators, and over the support of a kernel. The inner product operation was written in index notation as

$$A_{\alpha j} B_{j\beta}$$

and in EIN expressions with a summation symbol as

$$\langle \Sigma_j A_{\alpha j} B_{j\beta} \rangle_{\alpha\beta}.$$

Specifically, the inner-product of a 3-by-2 matrix and a vector of length 2 is expressed as

$$\left\langle \Sigma_{[0:j:1]} A_{ij} B_j \right\rangle_i$$

μ	$::=$	i, j, k	**Index Variables**
	$\mid$	$\underline{c}$	**Index constant**
α, β, η	$::=$	$\vec{\mu}$	**Multi-index, list of indices**
ν	$::=$	$[lb :: i :: ub]$	**Summation-range**
ψ	$::=$	$(\underline{c}, i)$	**Kernel differentiation**
e	$::=$	s	**real**
	$\mid$	T_α	**Tensor**
	$\mid$	$\mathcal{E}_{ijk}$	**Epsilon**
	$\mid$	$\delta_{\mu\mu}$	**Kronecker-deltas**
	$\mid$	$\Sigma_{\bar{\nu}} e$	**Summation**
	$\mid$	$-e$	**Negation**
	$\mid$	$e + e$	**Addition**
	$\mid$	$e - e$	**Subtraction**
	$\mid$	$e * e$	**Multiplication**
	$\mid$	$\frac{e}{e}$	**Division**
	$\mid$	F_α	**Field**
	$\mid$	$\frac{\partial}{\partial \alpha}$	**Derivative**
	$\mid$	$e \diamond e$	**Apply**
	$\mid$	$V_\alpha \circledast h^\alpha$	**Convolution**
	$\mid$	$e(e)$	**Probe**
	$\mid$	$\widetilde{i}$	**Value-of-index**
	$\mid$	$V_\alpha[\bar{e}]$	**Index-Images**
	$\mid$	$h^{\bar{\psi}}[e]$	**Kernel**
$\mathbf{E}$	$::=$	$\langle e \rangle_\alpha$	**EIN Operator**

Figure 3.1: EIN operator E and EIN expression e

where $i\epsilon 3$

A constant index $\underline{c}$ can represent a specific axis of a field or kernel. For instance, the x-direction and y-direction of the field is expressed as $F_{\underline{0}}, F_{\underline{1}}$, respectfully.
The $\widetilde{i}$ notation lifts an index variable to a constant integer.

$$\Sigma_{i=0}^{n} \widetilde{i} = 0 + 1 + 2 ... n.$$

The $\mathcal{E}_{iii}$ and $\delta_{\mu\mu}$ expressions act like their mathematical counterpart.

3.2.2 Basic expressions

Tensors and Fields

EIN expressions describe Tensors T and fields F similar index notation.

Index Images

The expression $V_\alpha[\bar{e}]$ denotes an image field with shape α, and $\bar{e}$ is the list of integer image-space position(s).

Kernels

The $h^{\bar{\psi}}[e]$ expression represents a kernel created for a specific dimension. The kernel evaluates at position e and the level and type of differentiation is expressed in the $\bar{\psi}$. $\bar{\psi}$ is a list of pairs $[(\underline{c}, i_1), (\underline{c}, i_2), ...(\underline{c}, i_m)]$. The first index is a constant index that represents the dimension of the kernel, while the second index is a variable index created by the differentiation operator. ψ pairs are evaluated like kronecker-deltas pairs. i.e. $\psi = (\underline{c}, i) = \delta_{c,i}$ and added together.

$$\psi_{\underline{c}j} = \begin{cases} 1 & \underline{c} = \mathbf{j} \\ 0 & \textbf{otherwise} \end{cases}$$

$h^{\bar{\psi}}$ is evaluated to h^k, where k is the k^{th} derivative of the kernel. h^1,and h^2 are the first, and second derivative of the kernel.

3.2.3 Field operations

Differentiation

Expressions $\diamond$ and $\frac{\partial}{\partial\alpha}$ are used to represent field differentiation. The $\diamond$ operator has two arguments that reduce to a partial and a field expression, $\frac{\partial}{\partial\alpha_1} \diamond F_{\alpha_2}$. The concise representation of differentiation allows us to express a wide variety of differentiation operations and also supports index-base optimizations on the differentiation operators.

Probe

The probe operator evaluates the field at a world-space position $e(e)$.

Convolution

The convolution expression $v_{\alpha_1} \circledast h^{\alpha_2}$ represents the convolution of an image field v, with the range α_1 and a polynomial kernel h. This expression can represent a scalar field $v \circledast h$, and a vector field $v_i \circledast h$. Additionally, the operator can represent the Gradient of a scalar field $v \circledast h^i$, Divergence of a vector field $\Sigma_i v_i \circledast h^i$, Laplacian of a scalar field $\Sigma_i v \circledast h^{i,i}$, and Jacobian of a vector field $v_i \circledast h^j$.

3.3 Operators

This Section will map the Diderot surface language to EIN expressions.

3.3.1 Tensor Operators

Tensor operators in the Diderot surface language are mapped to EIN operators. Operations on vectors include the cross product and the outer product (Figure 3.2). Operations on matrices include the Identity matrix, trace, and transpose (Figure 3.3). Other operations supported on tensors of arbitrary size include negation, addition, subtraction, scale, division, inner product, double dot, and modulate (Figure 3.4). In these cases, the EIN expressions have generic subscripts to represent the shape of their tensor arguments. The tensor types in the surface language are mapped to EIN subscripts in the generic operators. For example, the addition of tensors in Diderot syntax is

```
tensor [σ] A;
tensor [σ] B;
tensor [σ] T = A + B;
```

This computation is expressed with the general EIN operator as

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

$$\text{where } \sigma = \alpha$$

Many tensor operators are written the same as they would be in index notation, including the addition of tensors. The tensor operations that have an implicit summation in index notation will use an explicit Σ in EIN notation. For example, the trace expressed in Diderot syntax as

```
tensor [m,n] M ;
tensor [] t =trace(M);
```

The trace operator is expressed with the general EIN operator as

$$t = \lambda(A)\langle \Sigma_\nu A_{ii}\rangle(M)$$

$$\text{where } \nu = [0 : i : n] \text{ and } m = n$$

One of the benefits of using Index notation is that we can represent a family of operators with a concise expression. This property allows the compiler to generate the inner product and double-dot product between tensors of arbitrary shapes as shown in Figure 3.5 and Figure 3.6. The direct-style compiler represented the inner product $\cdot$ operation between tensors as several product operators: dot product between vectors, inner product of vectors and matrices, and the inner product of matrices. The inner product between tensors can be written in the Diderot Syntax as

```
tensor [σ] A;
tensor [σ'] B;
tensor [ς] T = A · B;
```

The inner product is expressed with the generic EIN operator as

$$\lambda(A,B)\langle \Sigma_i A_{\alpha i} B_{i\beta}\rangle_{\alpha\beta}$$

$$\text{where } \sigma = \alpha :: i, \sigma' = i :: \beta, \text{ and } \varsigma = \alpha\beta$$

The type-checker infers the types to the tensor arguments and specifies the generic operator by initializing the indices in α and β.

Outer Product	$A \otimes B$	$\lambda(A,B)\langle A_i B_j\rangle_{ij}$
Cross Product	$A \times B$	$\langle \Sigma_{jk} \mathcal{E}_{ijk} A_j B_k\rangle_i$

Figure 3.2: Operators on Vectors

Identity	I	$\lambda()\langle \delta_{ij}\rangle_{ij}$
Trace	Trace(A)	$\lambda A\langle \Sigma_i A_{ii}\rangle$
Transpose	A^T	$\lambda(A)\langle A_{ji}\rangle_{ij}$

Figure 3.3: Operators on Matrices

Negative Tensor	-A	$\lambda(A)\langle -A_\alpha\rangle_\alpha$
Addition	A+B	$\lambda(A,B)\langle A_\alpha + B_\alpha\rangle_\alpha$
Subtraction	A-B	$\lambda(A,B)\langle A_\alpha - B_\alpha\rangle_\alpha$
Scale	s*A	$\lambda(s,A)\langle sA_\alpha\rangle_\alpha$
Divide	$\frac{A}{s}$	$\lambda(s,A)\langle \frac{A_\alpha}{s}\rangle_\alpha$
InnerProduct	A · B	$\lambda(A,B)\langle \Sigma_i(A_{\alpha i} B_{i\beta})\rangle_{\alpha\beta}$
Double Dot	$A : B$	$\lambda(A,B)\langle \Sigma_{ij}(A_{\alpha ij} B_{ij\beta})\rangle_{\alpha\beta}$
Modulate	$A \odot B$	$\lambda(A,B)\langle A_\alpha B_\alpha\rangle_\alpha$

Figure 3.4: Operators on Tensors

Argument	Index(α,β)	expression
General	(α,β)	$\lambda(A,B)\langle \Sigma_k(A_{\alpha k} B_{k\beta})\rangle_{\alpha\beta}$
- vector ·vector	$(-,-)$	$\lambda(A,B)\langle \Sigma_k(A_k B_k)\rangle$
- matrix · vector	$(\alpha,-)$	$\lambda(A,B)\langle \Sigma_k(A_{\alpha k} B_k)\rangle_\alpha$
- vector · matrix	$(-,\beta)$	$\lambda(A,B)\langle \Sigma_k A_k B_{k\beta}\rangle_\beta$
- matrix ·matrix	(α,β)	$\lambda(A,B)\langle \Sigma_k(A_{\alpha k} B_{k\beta})\rangle_{\alpha\beta}$

Figure 3.5: Inner Product

Argument	Index(α,β)	expression
General	(α,β)	$\lambda(A,B)\langle\Sigma_{kl}(A_{\alpha kl}B_{kl\beta})\rangle_{\alpha\beta}$
- m : v	$(\alpha,-)$	$\lambda(A,B)\langle\Sigma_{jk}(A_{\alpha jk}B_{jk})\rangle_{\alpha}$
- v : m	$(-,\beta)$	$\lambda(A,B)\langle\Sigma_{jk}(A_{jk}B_{jk\beta})\rangle_{\beta}$
- m : m	(α,β)	$\lambda(A,B)\langle\Sigma_{kl}(A_{\alpha kl}B_{kl\beta})\rangle_{\alpha\beta}$

Figure 3.6: Double Dot Product

3.3.2 Basic Field Operators

This section focuses on the representation of simple field operators. Basic field evaluations use the convolution and probe operators. The representation of the convolution and probe operators in EIN expressions have been discussed in Section 3.2. The domain of the range of the image field is mapped to the image, and field subscript in the EIN expression. The convolution operation is written in the Diderot Syntax as

```
image(d)[σ] V;
field#k(d)[σ] F = V ⊛ h;
```

and with the generic EIN operator as

$$F = \lambda(v,h)\langle v_\alpha \circledast h\rangle_\alpha(V,h) \text{ where } \sigma=\alpha.$$

The probe operator is written in the Diderot syntax as

```
tensor [d] p;
tensor [σ] t= F(p);
```

and with the generic EIN operator as

$$t = \lambda(F,x)\langle F_\alpha(x)\rangle_\alpha(F,p) \text{ where } \sigma=\alpha.$$

Diderot lifts tensor operations on fields such as the negation, scale, addition, subtraction and division of fields. Operations on scalar fields, and of fields of arbitrary range are represented in EIN expressions in Figure 3.7. The scaling of a field can be expressed in the Diderot syntax as

```
tensor [] t;
field#k(d)[σ] F;
field#k(d)[σ] G=F*t;
```

The scaling operator is written with the generic EIN operator as

$$G = \lambda(s,F)\langle sF_\alpha\rangle_\alpha(t,F)$$

$$\text{where } \sigma=\alpha.$$

Ideally, any operator between tensors can be represented between fields. EIN expressions can represent addition, subtraction, product, and division between fields (Figure 3.8). Field operations have the same restrictions as their tensor operations counterparts. For instance, the two fields must have the same type in order to add or subtract them.

Description	Surface Language	EIN expression
Negative Field	-F	$\lambda F\langle -F_\alpha\rangle_\alpha$
Scale	s*F	$\lambda(s,F)\langle sF_\alpha\rangle_\alpha$
Division	$\frac{F}{s}$	$\lambda(s,F)\left\langle \frac{F_\alpha}{s}\right\rangle_\alpha$
Probe	F(x)	$\lambda(F,x)\langle F_\alpha(x)\rangle_\alpha$
Convolution	$v \circledast h$	$\lambda(e)\langle v_\alpha \circledast h\rangle_\alpha$
addtenField	s+f	$\lambda(s,f)\langle s+\varphi\rangle$
subTenField	s-f	$\lambda(s,f)\langle s-\varphi\rangle$
subFieldTen	f-s	$\lambda(s,f)\langle \varphi-s\rangle$

Figure 3.7: Operators on Fields

Surface Language	EIN expression	Description
F+G	$\lambda(F,G)\langle F_\alpha+G_\alpha\rangle_\alpha$	Addition
F-G	$\lambda(F,G)\langle F_\alpha-G_\alpha\rangle_\alpha$	Subtraction
(fg)	$\lambda(fg)\langle fg\rangle$	Product of Scalar Fields
f/g	$\lambda(fg)\langle f/g\rangle$	Division of Scalar Fields.
(fG)	$\lambda(fG)\langle fG_\alpha\rangle_\alpha$	Product of Fields.
$F\cdot G$	$\lambda(FG)\langle \Sigma_i F_{\alpha i}G_{i\beta}\rangle_{\alpha\beta}$	Inner Product of Fields
$F\times G$	$\lambda(FG)\langle \mathcal{E}_{ijk}F_jG_k\rangle_i$	Cross Product of Vector field

Figure 3.8: Tensor operators between fields

3.3.3 Field differentiation Operators

There are four different basic differentiation operators in EIN expressions: Curl $\nabla\times F$, Gradient $\nabla\varphi$, dot-times $\nabla\otimes F$, and Divergence $\nabla\cdot F$. Table 3.9 summarizes the basic types of differentiation EIN expressions support. Section 2.2 provided a mathematical basis for these different types field differentiation. The Curl has 2 different types of representations that will be reviewed in this section. The Gradient and dot-times are represented the same as in index notation from Section 2.4. The Divergence operator uses an explicit Σ to bind the variable index.

Description	Surface Language	EIN expression
Gradient	∇F	$\lambda F\left\langle \frac{\partial}{\partial x_i}\varphi\right\rangle_i$
Divergence	$\nabla\cdot F$	$\lambda F\left\langle \Sigma_i\frac{\partial}{\partial x_i}F_{\alpha i}\right\rangle_\alpha$
dot-times	$\nabla\otimes F$	$\lambda F\left\langle \frac{\partial}{\partial x_j}F_\alpha\right\rangle_{\alpha j}$
2d-Curl	$\frac{F_y}{dx}-\frac{F_x}{dy}$	$\lambda F\left\langle \frac{\partial}{\partial x_{\underline{0}}}F_{\underline{1}}-\frac{\partial}{\partial x_{\underline{1}}}F_{\underline{0}}\right\rangle$
3d-Curl	$\nabla\times F$	$\lambda F\left\langle \Sigma_{jk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}F_k\right\rangle_i$

Figure 3.9: Differentiation of Fields

Curl EIN Expressions provide a large advantage over direct-style notation when expressing the Curl operator. The 3-d Curl operator is unlike the other basic types of differentiation because it uses the $\mathcal{E}_{ijk}$ notation. The use of $\mathcal{E}_{ijk}$ allows us to do $\mathcal{E}_{ijk}$ base simplifications that are introduced in Section 4.3.2. The 3-d Curl is written in the Diderot syntax as

```
field#k(3)[3] F;
tensor [3] t = ∇ × F(p);
```

and in EIN notation as

$$t = \lambda F \left\langle \Sigma_{jk} \mathcal{E}_{ijk} \frac{\partial}{\partial x_j} F_k \right\rangle_i (F).$$

To compute the 2-d Curl we use the constant index to specify an axis and the result is a scalar.The 2-d Curl in Diderot syntax is

```
field#k(2)[2] F;
tensor [2] t = ∇ × F(p);
```

and in EIN notation

$$t = \lambda F \left\langle \frac{\partial}{\partial x_{\underline{0}}} F_{\underline{1}} - \frac{\partial}{\partial x_{\underline{1}}} F_{\underline{0}} \right\rangle (F).$$

dot-times The dot-times operator introduces a variable index to an expression. A single dot-times operator applied to a vector field evaluates the Jacobian. Two dot-time operators can be applied to a scalar field to represent the Hessian. Generally, multiple dot-times operators can be used to increase the level of differentiation to a field in respect to a variable index.

Divergence The Divergence operator uses an explicit Σ to represent the bound index on the field. The Divergence is written in the Diderot syntax as

```
field#k(d)[v] F;
field#k(d)[] G = ∇ · F
```

and in EIN notation

$$G = \lambda F \left\langle \Sigma_i \frac{\partial}{\partial x_i} F_{\alpha i} \right\rangle_\alpha (F)$$

EIN expressions can also represent more complicated types of differentiation. For instance, the Laplacian, Hessian, and Jacobian are combinations of the basic differentiation operators. Section 4.3 introduces the process of combining EIN operators to one another.

3.4 Discussion

EIN Expressions were designed to represent tensors and field computations in the Diderot compiler. This motivated the addition of operators that are not found in index notation, such as convolution, image indexing, and kernel differentiation. Therefore EIN expression supports a wider set of operations. This section will compare EIN operators to index notation as described by section 2.2. Other people have done work designing index notation, and that work is discussed further in the related works chapter.

3.4.1 Summation expression

In standard index notation an explicit summation symbol is rarely used. By using an explicit summation symbol, EIN expressions are able to support diverse operations, indicate range of a bound index, and vectorize code generation in a simpler way.

Implicit vs. explicit index

The index notation described in chapter 2.2 suggested that an implicit summation occurs when an index is repeated exactly twice. This strategy could not differentiate between the dot product and modulate, two different operations with a repeated index. The dot product is written in EIN expressions as

$$s = \langle \sum_i A_i B_i \rangle$$

and modulate as

$$C_i = \langle A_i B_i \rangle_i \ .$$

Indicating range

In standard index notation, the default range for a variable index starts at 0. The summation expression indicates the range of the variable index. The index can vary over a shape of tensor and the support of a kernel. The latter is particularly valuable to evaluating fields since it does not start at 0. For instance, kernel evaluation could be expressed as

$$s = \left| \ \left\langle \sum_{j=1-s}^{s} h(\widetilde{j}) \right\rangle \ \right|$$

Code generation

An explicit summation index can help split large EIN operators into vectorizable simple expressions. Without the summation expression, the compiler would need to scan every sub-expression in the EIN operator to find the summation indices and to indicate vectorization potential. Operations like the dot product provide opportunity for vectorization but could be embedded in complicated EIN expressions.

3.4.2 Subscripts

In index notation, only one free index is allowed in an expression. Since we support operations like modulate, we allow any number of variable index variables in an expression. The use of a constant index is not always discussed for index notation. The constant index allows us to express a specific axis or direction. Particularly, it allows us to compute the 2-d Curl and the differentiation in a kernel, which cannot be represented with variable indices.

3.4.3 Additional Notes

Consider the following expressions

1. $a_i = b_j$
2. a^i
3. $a_i b_j = b_j a_i$

The indices in EIN expressions are bound by outside the EIN operator or by the summation expression. (1) does not make sense because it is unclear where i and j are bound. An upper index has been used to differentiate between covariant and contra-variant tensors in other works. Our notation does not support upper index (2). Instead all tensors operations are represented with individual tensors and their multi-index. The product of tensors can be represented in any order (3).

Chapter 4

Implementation

In the lowering and optimization phase of the compiler, operations between tensors and fields are expressed with an EIN operator. Initially, these operations are expressed with a basic set of generic EIN expressions that are specialized. Then, as the optimization and transitions are made, EIN operators are changed to lower-level operations. Section 4.1 gives an overview of the SSA Infrastructure. Section 4.2 introduces how EIN operators are represented in the compiler in high-IL and mid-IL. Section 4.3 describes their simplifications and transitions. Writing mathematical operators in this concise way allows for index-base optimizations and rewriting in the high-IL optimization phase. During the transition between high-IL and mid-IL, field expressions are expanded as described in Section 4.4. Then, complicated EIN operators are split into simple EIN operators. Section 4.5 described the simple-EIN process. The code generation chapter discusses the transition from mid-IL to low-IL. During that transition, EIN operators are transformed into scalar and vector operations.

4.1 SSA Infrastructure

The optimization and lowering phase of the compiler occurs in three intermediate representations that are instances of Single State Assignment form [9]. Figure 4.1 presents the infrastructure of SSA. High-IL, mid-IL, and low-IL have the same control flow graph but vary on their types and operators. Section 4.2 will focus on the representation of EIN operators in high-IL and mid-IL.

$$
\begin{array}{rcll}
\varsigma & ::= & \textbf{variable} & \\
x & ::= & \textbf{State} & \textbf{Read strand states} \\
 & | & \textbf{Var}(\varsigma) & \\
 & | & \textbf{Lit} & \textbf{Literal} \\
 & | & \textbf{OP}(\bar{\varsigma}) & \textbf{Operator} \\
 & | & \textbf{Apply}(\bar{\varsigma}) & \textbf{basis function application} \\
 & | & \textbf{Cons}(\tau, \bar{\varsigma}) & \textbf{cons } \varsigma \textbf{ variables of type } \tau \\
 & | & \langle e \rangle_{\alpha}(\bar{\varsigma}) & \textbf{EIN operator}
\end{array}
$$

Figure 4.1: SSA

4.2 EIN Operators

The compiler supports all the operations on tensors mentioned in the design chapter, including addition, subtraction, negation, division, and multiplication of tensors as shown in Figures 3.2, 3.3 and 3.4. Diderot's implementation does not support all the field operations in the design chapter, and in particular, does not yet support division and products between fields. My implementation supports the negation, scaling, addition, subtraction and division by a scalar, probe, convolution, differentiation of fields, as well as basic addition and subtraction between fields as shown in Figures 3.7, 3.8, and 3.9.

EIN Expressions are originally mapped to a basic set of high-IL EIN operators. Then, the operators are translated to lower level operators. The derivations of fields are carried out in a series of steps. A field is represented abstractly with F_α until it is substituted with the $v \circledast h$ expression that represents the surface language definition of the field. Then, the probe of $v \circledast h$ gets expanded to index-images $V_\alpha[\bar{e}]$ and kernels $h^{\bar{\psi}}[e]$. Finally, the $V_\alpha[\bar{e}]$ and $h^{\bar{\psi}}[e]$ expressions are compiled away into tensor operations in low-IL. Presented here is the grammar for EIN operators in high-IL and mid-IL.

high-IL

During the high-IL optimization phase, the F_α is replaced with $v_\alpha \circledast h^\beta$. Substitutions are discussed in Section 4.3.1. The $\frac{\partial}{\partial\alpha}$ expression in the $\diamond$ constructor will then be pushed to the kernel in the convolution expression, $\frac{\partial}{\partial\beta} \diamond (v_\alpha \circledast h) \Longrightarrow v_\alpha \circledast h^\beta$. (Term rewrites are discussed in Section 4.3.2) The F_α,$\diamond$ and $\frac{\partial}{\partial\alpha}$ operators will disappear after the high-IL optimization phase. Figure 4.2 shows the EIN expression grammar in high-IL.

e	$::=$	s	**real**
	$\mid$	T_α	**Tensor**
	$\mid$	$\mathcal{E}_{iii}$	**Epsilon**
	$\mid$	$\delta_{\mu,\mu}$	**knocker-deltas**
	$\mid$	$\Sigma_{\bar{\nu}} e$	**Summation**
	$\mid$	$-e$	**Negation**
	$\mid$	$e + e$	**Addition**
	$\mid$	$e - e$	**Subtraction**
	$\mid$	$e * e$	**Multiplication**
	$\mid$	$\frac{e}{e}$	**Division**
	$\mid$	F_α	**Field**
	$\mid$	$\frac{\partial}{\partial\alpha}$	**Derivative**
	$\mid$	$e \diamond e$	**Apply**
	$\mid$	$v_\alpha \circledast h^\alpha$	**Convolution**
	$\mid$	$e(e)$	**Probe**

Figure 4.2: high-IL EIN Expression Grammar

mid-IL

The probe $(v_\nu \circledast h^\mu)(x)$ gets expanded to image and kernel expressions : $V_\alpha[\bar{e}]$, $h^{\bar{\psi}}[e]$. Section 4.4 describes the process of expanding fields. During the transition from high-to-mid-IL, EIN operations are also split into simple expressions as described in Section 4.5. Figure 4.3 shows the EIN expression representation in mid-IL.

$$
\begin{array}{rccc}
\psi & ::= & (\underline{c}, i) & \textbf{Delta} \\
b & ::= & s & \textbf{real} \\
 & | & T_\alpha & \textbf{Tensor} \\
 & | & \mathcal{E}_{iii} & \textbf{Epsilon} \\
 & | & \delta_{\mu,\mu} & \textbf{kronecker-deltas} \\
 & | & \widetilde{i} & \textbf{Value-of-index} \\
 & | & V_\alpha[\bar{e}] & \textbf{Index-Images} \\
 & | & h^{\bar{\psi}}[e] & \textbf{kernel} \\
 & & & \\
e & ::= & -b & \textbf{Negation} \\
 & | & b - b & \textbf{Subtraction} \\
 & | & \frac{b}{b} & \textbf{Division} \\
 & | & b + b & \textbf{Addition} \\
 & | & bb & \textbf{Multiplication} \\
 & | & \Sigma_{\bar{\nu}} b & \textbf{Summation}
\end{array}
$$

Figure 4.3: mid-IL EIN expressions Grammar

4.3 Simplifications and Transitions

The goal of the normalization stage is to make the compiler smarter about tensor calculus. With concise expressions, the simplification and transformation process is more efficient. The normalization stage in high-IL occurs in two phases, substitution and rewrites.

4.3.1 Substitutions

This section introduces a systematic way of substituting EIN operators into one another. These techniques allows the support of a larger set of operators that are mathematically sound. In the following examples 'outer indices' are the variable indices that are bound to the outer shape of the EIN operator.

Substitution Technique

An operation normally represented by multiple direct-style operators in the compiler can be combined and represented by one EIN operator. The outer product and addition operation is written in the Diderot syntax as

```
tensor a [σ];tensor b [σ];tensor s [ς]
tensor c [σ]= a+b;
tensor t [ςσ]= s ⊗ c;
```

and expressed in high-IL as two EIN operators

$$t_1 = \lambda(A,B)\langle A_\eta + B_\eta\rangle_\eta(a,b)$$
$$t_2 = \lambda(S,T)\langle S_\alpha T_\beta\rangle_{\alpha\beta}(s,t_1)$$
$$\text{where } \eta = \sigma, \alpha = \varsigma, \text{ and } \beta = \sigma$$

The high-IL optimization phase will combine these operators into one EIN operator. In any substitution where EIN operator t_1 is an argument to tensor T_β in t_2, T_β and t_1 must have the same shape. The outer indices for t_1 will map to the indices in T_β. In this case, t_1 has outer indices η and so $\eta \Longrightarrow \beta$. The result is a single EIN operator to represent these two operations.

$$t_2' \quad = \lambda(S,A,B) \quad \langle S_\alpha(A_\beta + B_\beta)\rangle_{\alpha\beta} \quad (s,a,b)$$
$$\text{where } \eta \Longrightarrow \beta$$

Field substitutions are treated the same way as tensor substitutions. A common substitution is a field by its image and kernel definition. The probing of a field is written in the Diderot syntax as

```
field#k(d)[σ]  F = V ⊛ h;
tensor [σ] t= F(p);
```

The convolution is expressed with EIN operator t_1 and probe with EIN operator t_2. The surface language gets mapped to EIN notation as

$$t_1 = \lambda(v,h)\langle v_\beta \circledast h\rangle_\beta(V,h)$$
$$t_2 = \lambda(F,x)\langle F_\alpha @ x\rangle_\alpha(t_1,p)$$
$$\text{where } \sigma = \beta = \alpha.$$

Then represented with one EIN operator as

$$t_2' = \lambda(v,h,x)\langle(v_\alpha \circledast h)@x\rangle_\alpha(V,h,p)$$
$$\text{where } \beta \Longrightarrow \alpha$$

Surface Language

Combinations of EIN operators are used to represent more complicated expressions in the surface language. For example, the $\nabla\times$ of the Gradient is the Hessian of a scalar field. The operators are written in the Diderot syntax as

```
field#k(d)[]φ;
field#k(d)[d,d] F= ∇ ⊗ ∇φ
```

This operation is represented by two different basic EIN operators. The Gradient is represented with EIN operator t_1 and the dot-times operator as EIN operator t_2.

$$t_1 \quad = \lambda(\varphi) \quad \left\langle \frac{\partial}{\partial x_k}\varphi \right\rangle_k \quad (f)$$
$$t_2 \quad = \lambda(F) \quad \left\langle \frac{\partial}{\partial x_j} \cdot F_i \right\rangle_{ij} \quad (t_1)$$

During the substitution phase, the operations will be combined to one operator. Outer index k in t_1 is mapped to index i in F_1. The new expression will be the Hessian of the scalar field.

$$t_2' \quad = \lambda(\varphi) \quad \left\langle \frac{\partial}{\partial x_j x_i}\varphi \right\rangle_{ij} \quad (f)$$
$$\text{where } k \Longrightarrow i$$

Index-base substitutions

The simple act of substitution can also allow us to take advantage of index-base optimizations. The trace of the outer product is written in the Diderot syntax as

```
tensor[d] a;tensor[d] b;
real s = $trace(a \otimes b)$
```

In high-IL, EIN expressions present the outer product and trace as

$$\begin{array}{llll} t_1 & = \lambda(A,B) & \langle A_i B_j \rangle_{ij} & (a,b) \\ t_2 & = \lambda(T) & \langle \Sigma_i T_{ii} \rangle & (t_1) \end{array}$$

When these operations are applied, the result will be the dot product. The outer indices for t_1 ij are mapped to indices for T_{ii}.

$$\begin{array}{llll} t_2' & = \lambda(A,B) & \langle \Sigma_i A_i B_i \rangle & (a,b) \\ & \text{where } i \Longrightarrow i & j \Longrightarrow i & \end{array}$$

Figure 4.4 shows examples from index-based substitution.

Surface Language	Description	EIN Op
$\nabla \cdot \nabla\varphi \Longrightarrow \nabla^2\varphi$	Grad(Divergence) $\Longrightarrow$ Laplacian	$\lambda f \left\langle \Sigma_i \frac{\partial}{\partial x_i} \frac{\partial}{\partial x_i} \varphi \right\rangle$
$\nabla \otimes (\nabla\varphi) \Longrightarrow \nabla \otimes \nabla\varphi$	dot-times (Gradient) $\Longrightarrow$ Hessian	$\lambda\varphi \left\langle \frac{\partial}{\partial x_i} \frac{\partial}{\partial x_j} \varphi \right\rangle_{ij}$
Trace$(\nabla \otimes \nabla\varphi) \Longrightarrow \nabla^2\varphi$	Trace(Hessian)$\Longrightarrow$ Laplacian	$\lambda F \left\langle \Sigma_i \frac{\partial}{\partial x_i x_i} \varphi \right\rangle$
Trace $(a \otimes b) \Longrightarrow a \cdot b$	Trace(outer) $\Longrightarrow$ dot product	$\lambda(A,B)\langle \Sigma_i A_i B_i \rangle$

Figure 4.4: Substitution

4.3.2 Rewrite Rules

Diderot's direct-style compiler supports various rewrites for probing and differentiation of fields. EIN notation introduces additional index-based rewrites supported by tensor calculus. As expressions become more complicated, it can be difficult to find instances of these rules that are embedded. The bulk of the rules are discussed and represented in EIN expressions in the Appendix. This section offers a few domain-specific and index-based rewrites.

Probe

The probe expression is distributed over the addition and subtraction operator (A9).

$$(F+G)(x) \Longrightarrow F(x) + G(x)$$

Other helpful rewrites allow us to make computations on the tensor result of a probe rather than the entire field.

$$(-F)(x) \Longrightarrow -(F(x))$$

$$(sF)(x) \Longrightarrow s(F(x))$$

Differentiation

Constants are moved to the outside of the differentiation expression (C1).

$$\nabla(s\varphi) \Longrightarrow s\nabla\varphi$$

The differentiation operator is distributed to each addition and subtraction subexpression with just one rule (A10).

$$\nabla(\varphi_1 + \varphi_2) \Longrightarrow \nabla\varphi_1 + \nabla\varphi_2$$

$$\nabla \cdot (\varphi_1 + \varphi_2) \Longrightarrow (\nabla \cdot \varphi_1) + (\nabla \cdot \varphi_2)$$

$$\nabla \times (F + G) \Longrightarrow (\nabla \times F) + (\nabla \times G)$$

The differentiation index is pushed down to the kernels in a convolution expression (C6).

$$\nabla(v \circledast h) \Longrightarrow v \circledast \nabla h$$

The direct-style version of the compiler supports this rewrite as

$$\nabla(v \circledast h^i) \Longrightarrow v \circledast h^{i+1}$$

This rule limits the type of differentiation that can be supported, so instead EIN notation has the following rewrite

$$\nabla_j(v \circledast h^i) \Longrightarrow v \circledast h^{ij}$$

Index-based rewrites

Index notation supports rewrites that were not available in the direct-style notation of the compiler. There are several index-based identities that were mentioned in Section 2.2. Two epsilons in an expression with a shared index can be rewritten to deltas (B1-B3) [8].

$$\mathcal{E}_{ijk}\mathcal{E}_{ilm} \Longrightarrow \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$$

A δ_{ij} expression can be applied to tensors, fields, convolution, and partial expressions, replacing an instance of j with i (B4- B7).

$$\delta_{ij}T_j \Longrightarrow T_i$$

$$\delta_{ij}F_j \Longrightarrow F_i$$

$$\nabla_j \diamond (\delta_{ij}e) \Longrightarrow \nabla_i \diamond (e)$$

Additionally, when the differentiation indices have two of the same epsilon indices, then the result is 0 (B8,B9).

$$\nabla_{ij} \diamond (\mathcal{E}_{ijk}e)) \Longrightarrow 0.0$$

Surface Language	Transition
$\nabla \times \nabla\varphi$	$\Longrightarrow 0$
$\nabla \cdot (\nabla \times F)$	$\Longrightarrow 0$
$\nabla \times (\nabla \times V)$	$\Longrightarrow \nabla(\nabla \cdot V) - \nabla^2 V$
$trace(V(x) \otimes G(x))$	$\Longrightarrow V(x) \cdot G(x)$

Figure 4.5: Fields Identities, φ is a scalar field, V and G are vector fields, and F is a generic field

4.3.3 Identities

Tensor-calculus based rewrites and the algebraic identities found from the systematic technique of substitution are advantages of this type representation in the compiler. Figure 4.5 and Figure 4.6, show examples of the tensor and field identities. The Appendix shows their transformations in EIN notation.

This section illustrates the transformation of a tensor and field identify. The tensor identity $(a \times b) \times c \Longrightarrow b(a \cdot c) - a(b \cdot c)$ is found by index-based optimizations. The cross product operations are written in the Diderot syntax as

```
tensor[3] a;tensor[3] b;tensor[3] c;
tensor[3] d = (a × b)
tensor[3] e = (d × c)
```

The cross product operations are expressed in high-IL with two EIN operators as

$$d = \lambda(A, B)\langle \Sigma_{jk}\mathcal{E}_{ijk}A_j B_k\rangle_i(a, b)$$

$$e = \lambda(A, B)\langle \Sigma_{jk}\mathcal{E}_{ijk}A_j B_k\rangle_i(d, c)$$

The EIN operators are combined to a single EIN operator as

$$e' = \lambda(u, v, t)\langle \Sigma_{jk}\mathcal{E}_{ijk}(\Sigma_{lm}\mathcal{E}_{jlm}u_l v_m)t_k\rangle_i(a, b, c).$$

Then, through a series of rewrites, the expression goes through the following transformations.

$$\begin{array}{cc}
\Longrightarrow & \lambda(u, v, t)\langle \Sigma_{jklm}\mathcal{E}_{ijk}\mathcal{E}_{jlm}u_l v_m t_k\rangle_i \\
\Longrightarrow & \lambda(u, v, t)\langle \Sigma_{klm}\delta_{kl}\delta_{im}u_l v_m t_k - \Sigma_{klm}\delta_{il}\delta_{km}u_l v_m t_k\rangle_i \\
\Longrightarrow & \lambda(u, v, t)\langle \Sigma_k(u_k v_i t_k) - \Sigma_k(u_i v_k t_k)\rangle_i \\
\Longrightarrow & \lambda(u, v, t)\langle v_i\Sigma_k(u_k t_k) - u_i\Sigma_k(v_k t_k)\rangle_i
\end{array}$$

This resulting EIN expression matches the right hand side of the algebraic identity $b(a\cdot c) - a(b\cdot c)$.

Presented next is the field identity $\nabla \times \nabla\varphi \Longrightarrow 0$. The Curl and Gradient are represented with two EIN operators as

$$c = \lambda F\left\langle \Sigma_{jk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}F_k\right\rangle_i$$

$$g = \lambda\varphi\left\langle \frac{\partial}{\partial x_i}\varphi\right\rangle_i(c).$$

Then, the EIN operators are combined to a single EIN operator in high-IL as

$$g = \lambda\varphi\left\langle \Sigma_{jk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}(\frac{\partial}{\partial x_k}\varphi)\right\rangle_i$$

Surface Language	Transition
$trace(T+S)$	$\Longrightarrow trace(T)+trace(S)$
$(S^T)^T$	$\Longrightarrow S$
$(a\times b)\times c$	$\Longrightarrow b(a\cdot c)-a(b\cdot c)$
$a\times(b\times c)$	$\Longrightarrow b(a\cdot c)-c(a\cdot b)$
$(a\times b)\times(c\times d)$	$\Longrightarrow (a\cdot(c\times d))b-(b\cdot(c\times d))a$
$(a\times b)\cdot(c\times d)$	$\Longrightarrow (a\cdot c)(b\cdot d)-(a\cdot d)(b\cdot c)$

Figure 4.6: Tensors

The computation is rewritten as

$$\Longrightarrow \lambda\varphi\left\langle \Sigma_{jk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_{jk}}\varphi\right\rangle_i$$

$$\Longrightarrow \lambda\varphi\langle 0)\rangle_i$$

4.4 Expanding a probe of a field

During the transition from high-IL to mid-IL, the probed fields are expanded. The new expression directly expresses the computations being done on separable image and kernels expressions. Section 2.4 introduces the mathematics behind the reconstruction of fields. Index notation could represent operations on fields in Section 2.2 but does not provide the notation needed to show the reconstruction of fields. EIN expressions $V_\alpha(\bar{e})$,$\widetilde{i}$,and $h^{\bar{\psi}}(e)$ were introduced for this purpose in Section 3.2. This section will represent the reconstruction of fields using EIN notation.

4.4.1 Transformation between basis

The following section will refer to image-space tensors **n** and **f**, and transformation matrix **P**. Those tensors are created using a mix of direct-style and EIN operators. Mid-IL operators “inverse” and “floor” are expressed in the compiler using direct-style notation. Those operators will be expressed as

$$e^{-1} \text{ and } \lfloor e \rfloor \text{ respectfully.}$$

In the surface language, a field can be probed at a position **p**. Position **p** is in world-space and is transformed to image-space position **x** using transformation matrix **M** and translation matrix **T**. We define the commonly used inverse of matrix **M** as **R**.

$$\mathbf{R}=\mathbf{M}^{-1}$$

$$\mathbf{x}=\lambda(A,B,C)\langle A_{ij}B_j+C_i\rangle_i(\mathbf{R},\mathbf{p},\mathbf{T})$$

The tensors **n** and **f** are created using **x** as

$$\mathbf{n}=\lfloor x\rfloor$$

$$\mathbf{f}=\mathbf{x}-\mathbf{n}$$

We can represent transformation **P** where **P**=$[\mathbf{M}^{-T}]_{WI}$ as the transpose of R with EIN operator

$$P=\lambda(R)\langle R_{ji}\rangle_{ij}(\mathbf{R})$$

4.4.2 Reconstruction of Pure Field in EIN Notation

Generally, the expansion of the probe of a d-dimensional field is in Figure 4.7. The probing of a 2-by-2 field is represented in the Diderot syntax as

```
field#k(2)[2,2] $F = V \circledast h$;
tensor [2,2] t= $F(p)$;
```

The surface language gets mapped to two EIN operators in high-IL and then combined into a single EIN operator as

$$t = \lambda(v,h,x)\langle v_\alpha \circledast h(x)\rangle_\alpha(V,h,p)$$

$$\text{where } \alpha = [k,l] \text{ and } k,l \text{ both range from 0 to 1}$$

The probe of a field is expanded to image and kernel subexpressions. An image $V([e])$ is evaluated at an integer neighborhood of positions. A d-dimensional field will create d kernel expressions $h(e)$. Mathematically, this operation on a 2-by-2 field is expressed as

$$\begin{bmatrix} \sum_{ij:1-s}^{s} & V_{00}([\mathbf{n}+i,\mathbf{n}+j]) & h(\mathbf{f}-i)h(\mathbf{f}-j) & V_{01}([\mathbf{n}+i,\mathbf{n}+j]) & h(\mathbf{f}-i)h(\mathbf{f}-j) \\ \sum_{ij:1-s}^{s} & V_{10}([\mathbf{n}+i,\mathbf{n}+j]) & h(\mathbf{f}-i)h(\mathbf{f}-j) & V_{11}([\mathbf{n}+i,\mathbf{n}+j]) & h(\mathbf{f}-i)h\mathbf{f}-j) \end{bmatrix}$$

The operation is written in EIN notation as

$$\left\langle \textstyle\sum_{ij:1-s}^{s}((v_\alpha[\mathbf{n}_{\underline{0}}+\widetilde{i},\mathbf{n}_{\underline{1}}+\widetilde{j}]) * (h[\mathbf{f}_{\underline{0}}-\widetilde{i}]) * (h[\mathbf{f}_{\underline{1}}-\widetilde{j}]))\right\rangle_\alpha$$

$$\text{where } \alpha = [k,l] \text{ and } k,l \text{ both range from 0 to 1}$$

The specific axis for the fractional $\mathbf{f}$ and integer $\mathbf{n}$ position are represented with constant index, $\underline{0}$ and $\underline{1}$. Variable indices are lifted to integer values as $\widetilde{i}$ and $\widetilde{j}$.

1-d	$\left\langle \Sigma_i((V_\alpha[\mathbf{n}_{\underline{0}}+\widetilde{i}]) * (h[\mathbf{f}_{\underline{0}}-\widetilde{i}]))\right\rangle_\alpha$
2-d	$\left\langle \Sigma_{ij}((V_\alpha[\mathbf{n}_{\underline{0}}+\widetilde{i},\mathbf{n}_{\underline{1}}+\widetilde{j}]) * (h[\mathbf{f}_{\underline{0}}-\widetilde{i}]) * (h[\mathbf{f}_{\underline{1}}-\widetilde{j}]))\right\rangle_\alpha$
3-d	$\left\langle \Sigma_{ijk}((V_\alpha[\mathbf{n}_{\underline{0}}+\widetilde{i},\mathbf{n}_{\underline{1}}+\widetilde{j},\mathbf{n}_{\underline{2}}+\widetilde{k}]) * (h[\mathbf{f}_{\underline{0}}-\widetilde{i}]) * (h[\mathbf{f}_{\underline{1}}-\widetilde{j}]) * (h[\mathbf{f}_{\underline{2}}-\widetilde{k}]))\right\rangle_\alpha$

Figure 4.7: Probe of Field

4.4.3 Reconstruction of the Differentiation of Fields in EIN Notation

Generally, the derivative of a field can be expressed as $\frac{\partial}{\partial x_\beta} \diamond F_\alpha$ in high-IL and $v_\alpha \circledast h^\beta$ in mid-IL. A derived field is reconstructed similar to a pure field. The difference in the reconstruction is in the kernel subexpressions $h^{\overline{\psi}}[x]$. The index for the differentiation is captured in $\overline{\psi}$. Each index in the multi-index β creates a pair, and ψ is a list of these pairs. Highlighted here are examples of the Gradient and Hessian.

First Derivative

Generally, the first derivative of a field can be expressed as $v_\alpha \circledast h^i$ in mid-IL, and the result of the probe operation is a tensor of shape α,i. The Gradient of a 2-d scalar field is written in the Diderot syntax as

```
field#k(2)[] φ = V ⊛ h;
field#k(2)[2] F = ∇φ;
tensor [2] t= F(p);
```

In high-IL, this would be expressed as

$$t = \lambda(v,h,p)\big\langle v \circledast h^{i}(p)\big\rangle_{i}(V,h,p).$$

and in mid-IL as

$$\Big\langle \Sigma_{jk}((V[\mathbf{n}_{\underline{0}} + \widetilde{j}, \mathbf{n}_{\underline{1}} + \widetilde{k}]) * (h^{\delta_{0i}}[\mathbf{f}_{\underline{0}} - \widetilde{j}]) * (h^{\delta_{1i}}[\mathbf{f}_{\underline{1}} - \widetilde{k}]))\Big\rangle_{i}$$

Two kernels $h^{\bar{\psi}}$ are created for this 2-d field where $\bar{\psi}$=$[(\underline{c}, i)]$ and $\underline{c}$ is a constant to represent an axis of the image. In the code generation phase each pair (c, i) is evaluated as $\delta_{c,i}$. The result is in index space, so we use transformation matrix $\mathbf{P}$ to transform the tensor to world-space.

$$\lambda(A,B)\langle \Sigma_{j} A_{ij} B_{j}\rangle_{i}(P,t)$$

Similarly, two kernels are created when we take the Jacobian for a 2-d vector fields. The Jacobian is written in Diderot syntax as

```
field#k(2)[2] F = V ⊛ h;
tensor [2,2] t= ∇ ⊗ F(p);
```

In high-IL this would be expressed as

$$\lambda(v,h,p)\big\langle v_{i} \circledast h^{j}(p)\big\rangle_{ij}$$

and in mid-IL as

$$\Big\langle \Sigma_{jk}((v_{i}[\mathbf{n}_{\underline{0}} + \widetilde{k}, \mathbf{n}_{\underline{1}} + \widetilde{l}]) * (h^{\delta_{0j}}[\mathbf{f}_{\underline{0}} - \widetilde{k}]) * (h^{\delta_{1j}}[\mathbf{f}_{\underline{1}} - \widetilde{l}]))\Big\rangle_{ij}$$

General examples of expanding the first derivative of a d-dimension field are in Figure 4.8.

1-d	$\Big\langle \Sigma_{i}((V_{\alpha}[\mathbf{n}_{\underline{0}} + \widetilde{i}]) * (h^{\delta_{0,\mu}}[\mathbf{f}_{\underline{0}} - \widetilde{i}]))\Big\rangle_{\alpha\mu}$
2-d	$\Big\langle \Sigma_{ij}((V_{\alpha}[\mathbf{n}_{\underline{0}} + \widetilde{i}, \mathbf{n}_{\underline{1}} + \widetilde{j}]) * (h^{\delta_{0,\mu}}[\mathbf{f}_{\underline{0}} - \widetilde{i}]) * (h^{\delta_{1,\mu}}[\mathbf{f}_{\underline{1}} - \widetilde{j}]))\Big\rangle_{\alpha\mu}$
3-d	$\Big\langle \Sigma_{ijk}((V_{\alpha}[\mathbf{n}_{\underline{0}} + \widetilde{i}, \mathbf{n}_{\underline{1}} + \widetilde{j}, \mathbf{n}_{\underline{2}} + \widetilde{k}]) * (h^{\delta_{0,\mu}}[\mathbf{f}_{\underline{0}} - \widetilde{i}]) * (h^{\delta_{1,\mu}}[\mathbf{f}_{\underline{1}} - \widetilde{j}]) * (h^{\delta_{2,\mu}}[\mathbf{f}_{\underline{2}} - \widetilde{k}]))\Big\rangle_{\alpha\mu}$

Figure 4.8: Differentiation of a Field

Second Derivatives

The second derivative of a field can be expressed as $\frac{\partial}{\partial x_{ij}} \diamond F_{\alpha}$ in high-IL and $v_{\alpha} \circledast h^{ij}$ in mid-IL. The Hessian of a 2-d scalar field in Diderot syntax as

```
field#k(2)[] φ = V ⊛ h;
field#k(2)[2,2] F = ∇ ⊗ ∇φ;
tensor [2,2] t= F(p);
```

In high-IL this would be expressed as

$$t = \lambda(v, h, p)\big\langle v \circledast h^{ij}(p)\big\rangle_{ij}(V, h, p).$$

Two kernels $h^{\overline{\psi}}$ are created for this 2-d field, $\overline{\psi}$=[$(\underline{c}, i)$,$(\underline{c}, j)$]. During the code generation phase each pair in ψ is evaluated as a δ and added together. The probe of a field is expressed in mid-IL as

$$t = \Big\langle \Sigma_{kl}((v[\mathbf{n}_{\underline{0}} + \widetilde{k}, \mathbf{n}_{\underline{1}} + \widetilde{l}]) * (h^{\delta_{0i}+\delta_{0j}}[\mathbf{f}_{\underline{0}} - \widetilde{k}]) * (h^{\delta_{1i}+\delta_{1j}}[\mathbf{f}_{\underline{1}} - \widetilde{l}]))\Big\rangle_{ij}$$

The result is in index space, so we use transformation matrix $\mathbf{P}$ to transform the tensor to world-space.

$$\lambda(A, B)\langle \Sigma_{kl} A_{ik} B_{kl} A_{jl}\rangle_{ij}(P, t)$$

Examples of expanding the second derivative of a d-dimensional field is in Figure 4.9.

1-d	$\Big\langle \Sigma_i((V_\alpha[\mathbf{n}_{\underline{0}} + \widetilde{i}]) * (h^{\delta_{0,\mu}+\delta_{0,\nu}}[\mathbf{f}_{\underline{0}} - \widetilde{i}]))\Big\rangle_{\alpha\mu\nu}$
2-d	$\Big\langle \Sigma_{ij}((V_\alpha[\mathbf{n}_{\underline{0}} + \widetilde{i}, \mathbf{n}_{\underline{1}} + \widetilde{j}]) * (h^{\delta_{0,\mu}+\delta_{0,\nu}}[\mathbf{f}_{\underline{0}} - \widetilde{i}]) * (h^{\delta_{1,\mu}+\delta_{1,\nu}}[\mathbf{f}_{\underline{1}} - \widetilde{j}]))\Big\rangle_{\alpha\mu\nu}$
3-d	$\Big\langle \Sigma_{ijk}((V_\alpha[\mathbf{n}_{\underline{0}} + \widetilde{i}, \mathbf{n}_{\underline{1}} + \widetilde{j}, \mathbf{n}_{\underline{2}} + \widetilde{k}]) * (h^{\delta_{0,\mu}+\delta_{0,\nu}}[\mathbf{f}_{\underline{0}} - \widetilde{i}]) * (h^{\delta_{1,\mu}+\delta_{1,\nu}}[\mathbf{f}_{\underline{1}} - \widetilde{j}]) * (h^{\delta_{2,\mu}+\delta_{2,\nu}}[\mathbf{f}_{\underline{2}} - \widetilde{k}]))\Big\rangle_{\alpha\mu\nu}$

Figure 4.9: Second differentiation of a Field

4.5 Simple-Ein

During the transition from high-IL to mid-IL, complicated EIN expressions are split into simpler ones in order to better identify methods for code generation and common subexpressions. Combining EIN operators in the optimization phase can lead to large and complicated EIN operators. A general code generator would need to expand every operation to work on scalars, which can be unwieldy and could miss the opportunity for vectorization. Instead, every EIN operator is split into a set of simple EIN operators. Each EIN expression then only has one operation working on constants, tensors, deltas, epsilons, images and kernels. Figure 4.3 shows the EIN expression representation in mid-IL. Here is an example of splitting an EIN operator

$$t = \lambda(A, B, C)\left\langle A_i + (\sum_j B_{ij} C_j)\right\rangle_i (a, b, c)$$

In this case, the EIN operator t is split into two EIN operators t_1 and t_2, each with a single operation. These mid-IL EIN operators are trivially rewritten as low-IL vector product and vector addition operators as

$$t_1 = \lambda(A, B)\left\langle \sum_j A_{ij} B_j\right\rangle_i (b, c)$$
$$t_2 = \lambda(A, B)\langle A_i + B_i\rangle_i (A, t_1)$$

Deciding how to multiply tensors can be much more complicated. The TCE project uses various techniques on multiplying tensors which include reshaping tensors and library calls in [13]. In the future, we hope to explore other techniques. For now, we split when there is an embedded operator or a summation. For instance,

$$t = \lambda(A, B, C)\left\langle A_i * (\sum_k B_{jk} C_k) \right\rangle_{ij} (a, b, c)$$

is split into two EIN operators as

$$t_1 = \lambda(A, B)\left\langle \sum_j A_{ij} B_j \right\rangle_i (b, c)$$

$$t_2 = \lambda(A, B)\langle A_i B_j \rangle_{ij} (a, g_1).$$

Splitting EIN expressions into simple expressions does not reverse the optimizations done earlier.

Chapter 5

Code Generation

Code generation is the last phase in the Diderot compiler. The direct-style compiler aims to generate code for a target platform, but there are some limitations. EIN notation is designed to extend the number of supported operations, but finding vectorization potential on a large complicated expressions can pose some challenges. The implementation chapter of this paper discussed techniques used to transform one EIN operator to a set of simple ones that can be better vectorized. EIN operators are transformed to low-IL operators, which are designed to take advantage of the target hardware. Section 5.1 will discuss the current state of the Diderot compiler. Section 5.2 will introduce the notation used by low-IL operators. Section 5.3 and Section 5.4 will demonstrate transforming tensor and field operations, respectfully.

5.1 Diderot Compiler

The Diderot compiler aims to take advantage of target hardware, but it has several limitations. The surface language expresses operations on tensors, but only tensor operations of a subset of lengths are supported in the code generation phase of the compiler. The code generator is not general enough. Any target hardware can have limited vector ability and the vector code would get implemented as scalar computations. Currently, we do not take advantage of the GPU's fine grain vector operations for small scale tensor operations. For these operators, the scheduling overhead to put the computation on separable threads could be more costly than the advantages.

The notation presented in this chapter was designed to take advantage of the target hardware before knowing what the hardware supports. Mid-IL EIN operators represent operations on tensors of arbitrary length. The notation in low-IL assumes that infinitely wide vector operators and vector reduction can be supported. We then intend to map the vector operators presented in this section to finite size operations as supported by the target hardware. If the platform does not support vector operations, then we use scalar operations. This intermediate step allows us to break down EIN operators into basic scalar and vector operations while keeping the ideal generated code in mind.

The design of EIN operators can pose some challenges for code generation. EIN operators created after the high-IL stage can be complicated and large. Currently, each EIN operator is simplified as it transitions to a mid-IL operator in order to better identify vectorization potential. There are various techniques that we can explore such as tensor reshaping and design choices. On the other hand, if each vector operation created in low-IL gets implemented as a scalar operation on the target-hardware, then some of these intermediate steps may not be worthwhile.

5.2 Notation

The figure below shows a representation of the low-IL operators. Low-IL vector operators include vector addition, vector subtraction, vector product, scalar/vector product, and summation. Low-IL scalar operators include scalar addition, subtraction, and product. Image probing is expressed with two low-IL operators Ld, and ς. All low-IL operators work on variable ids.

V	$::=$	**Image Field**	
$\boldsymbol{\mu}$	$::=$	**Constant index**	
ς	$::=$	$Addr(V, x)$	**image address of V[x]**
$Basic$	$::=$	$v_{\boldsymbol{\mu}}$	**Projection**
	$\mid$	$s_{\boldsymbol{\mu}}$	**Scalar**
	$\mid$	c	**const**
l	$::=$	$Basic$	
	$\mid$	$v_{\boldsymbol{\mu}} + v_{\boldsymbol{\mu}}$	**vector addition**
	$\mid$	$v_{\boldsymbol{\mu}} - v_{\boldsymbol{\mu}}$	**vector subtraction**
	$\mid$	$v_{\boldsymbol{\mu}} * v_{\boldsymbol{\mu}}$	**vector product**
	$\mid$	$\boldsymbol{\Sigma} v_{\boldsymbol{\mu}}$	**Summation or vector reduction**
	$\mid$	$s_{\boldsymbol{\mu}} + s_{\boldsymbol{\mu}}$	**scalar addition**
	$\mid$	$s_{\boldsymbol{\mu}} - s_{\boldsymbol{\mu}}$	**scalar subtraction**
	$\mid$	$s_{\boldsymbol{\mu}} s_{\boldsymbol{\mu}}$	**scalar product**
	$\mid$	$s_{\boldsymbol{\mu}} * v_{\boldsymbol{\mu}}$	**scalar vector product**
	$\mid$	$l :: l$	**cons to create a higher order tensor**
	$\mid$	ς	
	$\mid$	$Ld(V, \varsigma, d)$	**Load V starting at ς to create shape tensor[d]**

The compiler transforms EIN operators into a basic set of low-IL operators. The first step is to identify potential for vectorization. If vectorization can be used then a specific set of vector operators are used to generate code for the operation, otherwise, only scalar operations are used. The second step is to iterate over the shape of the tensor result, and bound variable indices. The third step is to evaluate each expression in the body of the EIN operator. Tensors can be evaluated as a basic expression $s_{\boldsymbol{\mu}}$ or $v_{\boldsymbol{\mu}}$ depending on whether vectorization is being used. Section 6.3 discusses Kernels and Images, which are more complicated. The expressions $\mathcal{E}_{iii}$ and $\delta_{\mu,\mu}$ are evaluated to an integer. $\widetilde{i}$ is lifted to the bound value of variable index i. If there is a $\sum$ then we unroll loops and bound variable indices. EIN expressions $+, -, *, /$ are mapped to their low-IL counterparts $+, -, *, /$.

This transition uses basic algebraic rewriting. Evaluating $\widetilde{i}$, $\mathcal{E}_{iii}$ and $\delta_{\mu,\mu}$ to integers can provide a new opportunity to simplify subexpressions. Field probing typically requires high-arithmetic intensity, so it is optimal to have a design that can detect common subexpressions. We lift the image address out of the load operator and image-space position from the kernel evaluation.

5.3 Tensors

The dot product and outer product on tensor A, B and C can be written in Diderot syntax as

```
1 tensor [3,2] A; vec2 B; tensor [2,3] C;
2 tensor [3] D = A · B;
3 tensor [3] E = $B \cdot A^T$;
4 tensor [3,3] F = A · C;
5 tensor [3,2,2] G = A ⊗ B;
```

The inner product operation on line 2 is written in mid-IL as

$$\langle \Sigma_j A_{ij} B_j \rangle_i \textbf{ where } i\epsilon 3 \textbf{ and } j\epsilon 2$$

low-IL transforms the EIN operator into vector product, vector reduction, and cons operations as

$$\begin{array}{ccc} a = u_{\mathbf{0}} * v & b = u_{\mathbf{1}} * v & c = u_{\mathbf{2}} * v \\ a' = \mathbf{\Sigma} a & b' = \mathbf{\Sigma} b & c' = \mathbf{\Sigma} c \\ & D = a' :: b' :: c' & \end{array}$$

$$\textbf{where } u = A \textbf{ and } v = B$$

This operation can also be written in Diderot syntax on line 3 and written in mid-IL as

$$\langle \Sigma_j B_j A_{ji} \rangle_i \textbf{ where } i\epsilon 3 \textbf{ and } j\epsilon 2$$

This computation should produce the same set of low-IL operators, but currently it does not. This example highlights a missed opportunity in the current implementation of EIN operators. The expression here needs to be organized in order to detect the potential for vectorization. As is, this expression will use low-IL scalar operators and transform the EIN operator as

$$\begin{array}{ccc} a_0 = s_{00} r_0 & a_1 = s_{10} r_1 & x_0 = a_0 + a_1 \\ b_0 = s_{01} r_0 & b_1 = s_{11} r_1 & x_1 = b_0 + b_1 \\ c_0 = s_{02} r_0 & c_1 = s_{12} r_1 & x_2 = c_0 + c_1 \\ & D = x_0 :: x_1 :: x_2 & \end{array}$$

$$\textbf{where } s = A \textbf{ and } r = B$$

The inner product operation between matrices can be written in Diderot syntax on line 4 and written in mid-IL as

$$\langle \Sigma_k B_{ik} A_{kj} \rangle_{ij} \textbf{ where } i,j\epsilon 3 \textbf{ and } k\epsilon 2$$

Similarly, this computation is an example of a missed opportunity in the current implementation of EIN operators. As is, this expression will use low-IL scalar operators not presented here. The outer product of tensor A and B is written Diderot syntax in Line 5 and in mid-IL as

$$\langle A_{ij} B_k \rangle_{ijk}$$

In low-IL this expression uses the scalar vector product operator as

$$\begin{array}{ccc} a_0 = s_{00} * v & a_1 = s_{01} * v & x_0 = a_0 :: a_1 \\ b_0 = s_{10} * v & b_1 = s_{11} * v & x_1 = b_0 :: b_1 \\ c_0 = s_{20} * v & c_1 = s_{21} * v & x_2 = c_0 :: c_1 \\ & E = x_0 :: x_1 :: x_2 & \end{array}$$

$$\textbf{where } s = A \textbf{ and } v = B$$

Cons operator used in x_0, x_1, x_2 creates an i-by-j matrix, and the cons operator used in E creates an i-by-j-by-k tensor.

5.4 Fields

At this point, probed fields are expressed in terms of Images, kernels, image-space positions, and a summation symbol. Low-IL will unroll the summation loops, probe image fields, and evaluate kernels to produce a block of simple low-IL operators. The Gradient of a 2-d scalar field is written in mid-IL as

$$\left\langle \Sigma_{jk}((V[n_{\underline{0}} + \widetilde{j}, n_{\underline{1}} + \widetilde{k}])(h^{\delta_{0i}}[f_{\underline{0}} - \widetilde{j}])(h^{\delta_{1i}}[f_{\underline{1}} - \widetilde{k}])) \right\rangle_i$$

The kernel has just one variable index i and it ranges from 0 to 1. In low-IL we iterate over the outer index, bind every instance of i, and create the structure of the tensor as

$$\Longrightarrow \begin{bmatrix} \Sigma_{jk}((V[n_{\underline{0}} + \widetilde{j}, n_{\underline{1}} + \widetilde{k}])(h^{\delta_{00}}[f_{\underline{0}} - \widetilde{j}])(h^{\delta_{10}}[f_{\underline{1}} - \widetilde{k}])) \\ \Sigma_{jk}((V[n_{\underline{0}} + \widetilde{j}, n_{\underline{1}} + \widetilde{k}])(h^{\delta_{01}}[f_{\underline{0}} - \widetilde{j}])(h^{\delta_{11}}[f_{\underline{1}} - \widetilde{k}])) \end{bmatrix}$$

Evaluating the upper subscript to each kernel produces an expression similar to the mathematical counterpart.

$$\Longrightarrow \begin{bmatrix} \Sigma_{jk}((V[n_{\underline{0}} + \widetilde{j}, n_{\underline{1}} + \widetilde{k}])(h^{1}[f_{\underline{0}} - \widetilde{j}])(h^{0}[f_{\underline{1}} - \widetilde{k}])) \\ \Sigma_{jk}((V[n_{\underline{0}} + \widetilde{j}, n_{\underline{1}} + \widetilde{k}])(h^{0}[f_{\underline{0}} - \widetilde{j}])(h^{1}[f_{\underline{1}} - \widetilde{k}])) \end{bmatrix}$$

The image and kernel values are computed independently. For the following example let

$$lb = 1 - s, s' = 2 * s \textbf{ and } y = n_{\underline{1}} + lb$$

Images We can define the neighborhood of positions as

$$Pos = \begin{bmatrix} (n_{\underline{0}} + lb, y) & ... & (n_{\underline{0}} + lb, y + s) \\ .. & .. & .. \\ (n_{\underline{0}} + s, y) & .. & (n_{\underline{0}} + s, y + s) \end{bmatrix}$$

The image is probed at these positions by loading a vector of positions over the fastest-axis. We get the address of a single position x and then load a vector of length s' from that position.

$$\varsigma = Addr(V, x)$$

$$L = Ld(V, \varsigma, s')$$

Specifically, we get the address for each position in the first column of Pos by using scalar operations

$$x = \begin{bmatrix} n_{\underline{0}} + lb, y \\ n_{\underline{0}} + lb + 1, y \\ ...., y \\ n_{\underline{0}} + s, y \end{bmatrix}$$

and use ld to load over the fast axis.

$$L_i = Ld(V, Addr(V, x_i), s')$$

Kernels The kernels are evaluated independently over the range of its support. For simplicity, we can represent this process as

$$P(h^k, x) \Longrightarrow t$$

where h is a kernel, k is the level of differentiation, x is a position vector with length s', and t is a vector with length s'. We can define polynomial a as the segments to the polynomial kernel h, and so P evaluates

$$a_0 + a_1 x + a_2 x^2 + ... a_x x^n$$

This is implemented as

$$a_0 + x(a_1 + x(a_2 + ... x a_n)$$

to reduce the number of expensive product operations. The 2-d field in this example has two kernel computations that range over the support of the kernel.

$$hXPOS = \begin{bmatrix} f_{\underline{0}} - lb \\ .. \\ f_{\underline{0}} - s \end{bmatrix}$$

$$hYPOS = \begin{bmatrix} f_{\underline{1}} - lb \\ .. \\ f_{\underline{1}} - s \end{bmatrix}$$

h_x and h_y then are the polynomial evaluation of the kernels, explicitly represented as

$$\begin{array}{ll} h_{x0} = P(h^1, hXPOS) & h_{y0} = P(h^0, hYPOS) \\ h_{x1} = P(h^0, hXPOS) & h_{y1} = P(h^1, hYPOS) \end{array}$$

Finally, the Gradient of the 2-d field has the final vector product operations

$$\begin{bmatrix} \boldsymbol{\Sigma} \begin{bmatrix} \boldsymbol{\Sigma} L_0 {*} h_{y0} \\ \boldsymbol{\Sigma} L_1 {*} h_{y0} \\ .. \\ \boldsymbol{\Sigma} L_{s'} {*} h_{y0} \end{bmatrix} {*} h_{x0} \\ \boldsymbol{\Sigma} \begin{bmatrix} \boldsymbol{\Sigma} L_0 {*} h_{y1} \\ \boldsymbol{\Sigma} L_1 {*} h_{y1} \\ .. \\ \boldsymbol{\Sigma} L_{s'} {*} h_{y1} \end{bmatrix} {*} h_{x1} \end{bmatrix}$$

Chapter 6

Related Work

There are various domain-specific languages that can provide a link between the mathematical algorithms and programming. These languages allow the end-user to write code that looks like mathematics and lets the system focus on generating high-performance code. In particular, the Spiral DSL and Tensor Contraction Engine projects feature relevant designs of tensor operators and techniques used for producing high quality code. Lastly, we will discuss how index notation has been expanded and implemented in a variety of ways.

Spiral Spiral, a DSL created for digital signal processing, is designed to support the mathematical knowledge of various algorithms such as fast Fourier transform, circular convolution, and viterbi decoding. Its implementation uses a signal processing language (SPL) featuring a small set of constructs that includes symbols, matrix operators, krnocker product, butterfly matrix, and stride permutation matrix. Spiral automatically generates code for algorithms in their target domain for specific hardware. Operator Language (OL) was created to help generate high quality code in the Spiral compiler [12]. They extend SPL to OL using a few kernels. OL is translated into a target program, usually C code and library calls that help with their domain.

TCE TCE is tensor contraction engine created by Hatono, Albert et. al to represent quantum chemistry [1]. Their work provides an algorithm to use common subexpression eliminate (CSE) in their compiler. Tensor contraction representation can cause an explosion in the search space. Diderot supports CSE on direct-style notation and on EIN operators in a literal way. The TCE project approaches problems in effective code generation [17]. They have an algorithm that chooses between libraries, including GEMM, BLAST [20]. They discuss their techniques to cost-effectively multiply tensors and find the best sequence of two-tensor contractions [13]. The different order of operations can result in very different cost-effective operation costs. The work demonstrates that the best course for code generation can depend on the index ranges of tensors. They provide examples of expressions being split apart and tensors being reshaped before making calls to libraries [2]. The approach in this paper is not as extensive, as it splits EIN operators based on their outer operation, but only handles a basic set of ways to cost-effectively do these computations.

Notation Support Ahlander et al. created support for programming directly with index notation [3]. The work develops grammar and semantics for index notation as a DSL and implemented as a C++ library. Algorithms written in index notation can be converted to code made for scientific computing. By using index notation in software, it closes a notational gap between mathematical understanding and programming. This work supports index notation in the compiler and not the surface language and so Diderot keeps it programmability while still gaining the advantages that index notation can provide.Some of these advantages include term rewriting and the ability to find tensor identities. Franchetti does manipulation with well known mathematical identities in order to produce vector code in [11]. This research does term rewriting based on mathematical identities and does vectorization as a separate step. Rocklin, M. built a system to generate linear algebra code in [19]. His work did similar term rewriting to reflect algebraic identities.

Index Notation Einstein index notation, sometimes called the summation convention, can be used to represent physical quantities, stress tensors, velocity vectors, and various algorithms in scientific computing. Various implementations of index notation require the designers to study the ambiguities and limitations, and to develop grammar and semantics. This paper introduces an index-inspired design, but there are some notable differences.

Index notation can support a diverse set of operations that Diderot currently does not support. There are notable efforts to represent index notation on paper for various operations, functions, transforms including Taylor series, angular velocity, physics concepts [3], rotation [4], and spherical coordinates [5]. EIN notation expands the set of tensor and field operations that are applicable to Diderot but provides the framework for much more.

A major part of the ambiguity in index notation is related to the implicit summation. An implicit summation restricts the types of operations that can be supported. Therefore, various things are done to suppress summation [3]. These include using notation to differentiate between types of indices, using a no sum-operator [4], and describing summation to be between one upper and one lower index [8, 21] or between indices that repeat exactly twice [23, 10]. EIN notation uses an explicit summation symbol leading to more book-keeping but allowing us to express explicit boundaries for diverse operations. A specific axis of a tensor is only supported by some. Sokolnikoff, I. uses parenthesis [23]. EIN uses an underbar $\underline{c}$.

Upper and lower indices have been used on tensors to distinguish between covariant and contravariant and to define summation. Bolton, E. highlighted that the notation can be avoided for orthogonal coordinates [5]. EIN notation chooses to not use upper and lower indices. Instead, the compiler does the transformation between world-space and image-space internally.

Chapter 7

Future Work

While the design of EIN notation has added many desired operators to Diderot, we have a plans to further extend the operators. The internal representation and optimizations have been implemented up to low-IL and so code generation is the next step.

7.1 Implementation More Tensor Calculus

We intend to extend the implementation to a larger set of field operators, such as the inner product $F \cdot G$ and cross product $F \times G$. We can compute tensor operators typically used on values and lift them to be used on fields. We described the design of these additional field operations in the Design chapter. Normalization techniques used on tensors can be lifted to fields. A summary of the new algebraic identities can be found in the table below.

Surface Language	Transition
$\nabla(fg)$	$f(\nabla g) + g(\nabla f)$
$\nabla \cdot (\nabla f \times \nabla g)$	0
$\nabla \cdot (f\nabla g - g\nabla f)$	$f\nabla^2 g - g\nabla^2 f$
$\nabla(A \cdot B)$	$(A \cdot \nabla)B + (B \cdot \nabla)A + A \times (\nabla \times B) + B \times (\nabla \times A)$

The transitions are expressed in the figure above as they would be in the surface language. In some cases the computation will be translated to a less expressive EIN expression that is tough to express in the surface language. For instance, $\nabla(A \cdot B)$ is expressed as $\langle \Sigma_{ij} A_j \nabla_i B_j + \Sigma_{ij} B_j \nabla_i A_j \rangle_i$ instead of the more complicated transition presented on the right hand side.

7.2 Optimization Techniques

The Diderot compiler uses various standard optimization techniques. Common subexpression elimination (CSE) has been implemented to work on EIN operators quite literally. Consider expressions $c + b + a$ and $a + c + b$ - they would not be detected as equal. Simple EIN operators provide a simple solution in this case because we could just order the tensors a + b + c. Ordering every EIN operator, however, may not be cost effective.

7.3 Code generation

We intend to complete the implementation of the code generation phase. The current low-IL representation of EIN expressions assumes infinitely-wide vector operators in order to take advantage of the variety of plausible hardware targets. The next step is to map all low-IL vector operations to either finite vector operations or scalar operations, depending on the target hardware.

The direct-style notation used in the Diderot compiler made it easy to vectorize operations that are otherwise not obvious in EIN notation. For instance, the inner product between matrices generates vector operators in the direct-style notation compiler but into scalar operations in the EIN notation version of the compiler. We can explore a representation of EIN notation that can indicate potential for vectorization that may not be straightforward otherwise.

Chapter 8

Conclusion

Diderot is a domain-specific language for visualization algorithms. Diderot provides the programmer with a very-high-level programming model based on the concepts and notations of tensor calculus, which allows the algorithms to be expressed in their natural mathematical notation. We have presented the design of EIN notation to represent these tensor and field operations in the Diderot compiler. We have described the implementation and techniques used to make optimizations.

Bibliography

[1] et. al A. Hartono. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry*, 113(45):12715–12723, 2009.

[2] Wenjing Ma Sriram Krishnamoorthy Oreste Villa Karol Kowalski Gagan Agrawal. Optimizing tensor contraction expressions for hybrid cpu-gpu execution. *Cluster Computing Special Issue*, 2011.

[3] K. Ahlander. Einstein summation for multi-dimensional arrays. *An International Journal computers and mathematics with applications*, pages 1007–1017, December 2002.

[4] A. Barr. The einstein summation notation, introduction to cartesian tensors and extensions to the notation. Draft paper; available at url-http://zeus.phys.uconn.edu/ mcintyre/workfiles/Papers/Einstien-Summation-Notation.pdf.

[5] E. Bolton. A simple notation for differential vector expressions in orthogonal curvilinear coordinates. *Geo-physical Journal International*, 115(3):654–666, December 1999.

[6] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques (SIGGRAPH '93)*, pages 263–270, New York, NY, August 1993. ACM.

[7] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *Proceedings of the 2012 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 111–120, New York, NY, June 2012. ACM.

[8] Tai L. Chow. *Mathematical Methods for Physicists : A Concise Introduction.* CAMBRIDGE UNIVERSITY PRESS, Cambridge, 2000.

[9] 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, Oct 1991.

[10] Kees Dullemond and Kasper Peeters. *Introduction to Tensor Calculus.* Kees Dullemond and Kasper Peeters, 1991.

[11] F. Franchetti, Y. Voronenko, and M. Pu schel. A rewriting system for the vectorization of signal transforms. *In: High Performance Computing for Computational Science (VECPAR)*, 4395:363–377, 2006.

[12] Franz Franchetti, Frdric de Mesmay, Daniel McFarlin, and Markus Pschel. Operator language: A program generation framework for fast kernels. *IFIP*, 2009.

[13] Albert Hartono, Alexander Sibiryakov, Marcel Nooijen, Gerald Baumgartner, and David E. B. Automated operation minimization of tensor contraction expressions in electronic structure calculations. *Proc. ICCS 2005 5th International Conference on Computational Science*, pages 155–164, 2005.

[14] Miloš Hašan, John Wolfgang, George Chen, and Hanspeter Pfister. Shadie: A domain-specific language for volume visualization. Draft paper; available at url-http://miloshasan.net/Shadie/shadie.pdf, 2010.

[15] Gerhard A. Holzapfel. *Nonlinear Solid Mechanics.* John Wiley and Sons, West Sussex, England, 2000.

[16] Gordon Kindlmann. Foundations of scientific visualization. Draft paper; available from Dr.Kindlmann, January 2013.

[17] Qingda Lua, Xiaoyang Gaoa, Sriram Krishnamoorthya, Gerald Baumgartnerb, J. Ramanujamc, and P. Sadayappana. Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions. *Journal of Parallel and Distributed Computing*, 72:338352, March 2012.

[18] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, February 2005.

[19] Matthew Rocklin. Mathematically informed linear algebra codes through term rewriting. PhD Thesis, August 2013.

[20] S.Blackford, J.Demmel, J. Dongarra, I.Du, S.Hammarling, G.Henry, M. Heroux, L. Kaufman, A. Lumsdaine, and A. Petitet. An updated set of basic linear algebra subprograms blast. Draft paper; available at url=http://dl.acm.org/citation.cfm?id=567807, 2001.

[21] James. Simmonds. *A Brief on Tensor Analysis.* Springer-Verlag, New York, 1982.

[22] D.B. Skillicorn, Jonathan M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[23] Sokolinkoff. *Tensor Analysis.* John Wiley and Sons, New York, 1960.

[24] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML '11)*, June 2011.

Appendix A

Term Rewriting

A.1 Rewrite Rules

As expressions get more complicated it can be difficult to find instances of rules that are embedded. This section reviews the rewrite rules that are applied in order to make domain-specific optimizations, tensor calculus, and other simplifications. The rewrite rules are written using the index notation described in the implementation section. We will add three new expressions χ, ρ ,and θ to simplify how the rules are expressed.

$$\begin{array}{rccc}
\chi & ::= & s & \textit{Constant} \\
 & | & \widetilde{i} & \textit{Index} \\
 & | & T_{[]} & \textit{real} \\
\rho & ::= & \delta_{ij} & \textit{Deltas} \\
 & | & \mathcal{E} & \textit{Epsilon} \\
\theta & ::= & + & \textit{addition} \\
 & | & - & \textit{subtraction}
\end{array}$$

The rules are divided and discussed here by their goals in mind. Algebraic rules uses algebraic principals like distributivity. Index optimization rules are those that are possible by comparing indices. Various rewrites are created so differentiation and probe operations work primarily on the fields. Additionally, some rules aim to flatten or simplify expressions. Finally, the generic rules are implemented for each expression when no other rule applies.

Algebraic Rules Several algebraic rules are used to make a cost-effective expression, Rules (A.1- A.6). Rules (A.7-A.10) are used to show the distributive property of EIN operators, such as $(a + b)c \Longrightarrow ac + bc$ (A.7). Summations are distributed so that the compiler can apply index-based optimization independently (A.8) The Apply operator is distributed to each addition and subtraction subexpression. So expressions like $\nabla(\varphi_1 + \varphi_2) \Longrightarrow \nabla\varphi_1 + \nabla\varphi_2$ and $\nabla \cdot (\varphi_1 + \varphi_2) \Longrightarrow (\nabla \cdot \varphi_1) + (\nabla \cdot \varphi_2)$ can be supported with just one single rule (A.10).

A.1 $(a - b) - e_2 \Longrightarrow a - (b + e_2)$

A.2 $e_1 - (c - d) \Longrightarrow (e_1 - c) + d$

A.3 $(a - b) - (c - d) \Longrightarrow (a + d) - (b + c)$

A.4 $\frac{\frac{a}{b}}{c} \Longrightarrow \frac{a}{bc}$

A.5 $\frac{\frac{a}{b}}{c} \Longrightarrow \frac{ac}{b}$

A.6 $\frac{\frac{a}{b}}{\frac{c}{d}} \Longrightarrow \frac{ad}{bc}$

A.7 $(e_1 \theta e_2)es \Longrightarrow (e_1 es)\theta(e_2 es)$

A.8 $\Sigma(e_1 \theta e_2) \Longrightarrow (\Sigma e_1)\theta(\Sigma e_2)$

A.9 $(F\theta G)@, x \Longrightarrow (F@x)\theta(G@x)$

A.10 $\nabla \diamond (e_1 \theta e_2) \Longrightarrow (\nabla \diamond e_1)\theta(\nabla \diamond e_2)$

Index Optimization ρ expressions are anticipated with summation symbol that bind their indices. Summation indices can only occur twice in an EIN expressions. The extra summation constructor adds extra book-keeping and causes us to look for the identity in many different forms. Epsilons are used to represent a relationship between the indices in the cross product of tensors or in the Curl of a field. Two Epsilons with a shared index can be rewritten to deltas, $\mathcal{E}_{ijk}\mathcal{E}_{ilm} \Longrightarrow \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}$ (B.1 -B.3). Naturally, the next step would be to distribute the deltas to the remainder of the expression. A δ_{ij} expression can be applied to tensors, fields, convolution, and partial expressions, replacing a instance of j with i (B.4-B.7). When two of the indices in $\mathcal{E}_{ijk}$ are repeated in the tensor or differentiation operator than the result is zero (Rule B.8-Rule B.9)

When an index is removed from a tensor because of a ρ, rewrite, then it is also removed from the summation index. Rule (B.10) is used for clean up. A χ has no variable index and moved outside (B.11).

B1 $$\frac{\Sigma(c_1 c_2, \mathcal{E}_{i--}\mathcal{E}_{i--} e_1 e_2 es) \Longrightarrow e'}{\Sigma(c_1, \mathcal{E}_{i--} * e_1) * \Sigma(c_2, \mathcal{E}_{i--} * e_2) * es \Longrightarrow e'}$$

B2 $$\frac{\mathcal{E}_{ijk}\mathcal{E}_{ilm} \Longrightarrow \delta_{jl}\delta_{km} - \delta_{jm}\delta_{kl}}{\Sigma_{i,c}(\mathcal{E}_{ijk}\mathcal{E}_{ilm}p) \Longrightarrow \Sigma_c(\delta_{jl}\delta_{km}p) - (\Sigma_c \delta_{jm}\delta_{kl}p)}$$

B3 $$\frac{\Sigma_{c_1 c_2}(\mathcal{E}_{i--} * \mathcal{E}_{i--} * p_1 * p_2) \Longrightarrow e'}{\Sigma_{c_1}(\mathcal{E}_{i--} * \Sigma_{c_2}(\mathcal{E}_{i--} * p_1) * p_2) \Longrightarrow e'}$$

B4 $\Sigma_{jc}, \delta_{ij} T_j \Longrightarrow \Sigma_c T_i$

B5 $\Sigma_{j,c}\delta_{ij} F_j \Longrightarrow \Sigma_c F_i$

B6 $\Sigma_{jc}(\nabla_j \diamond (\delta_{ij} e)) \Longrightarrow \Sigma_c \nabla_i \diamond (e)$

B7 $\Sigma_{jc}(\delta_{ij} v \circledast h^{\delta_{cj}}) \Longrightarrow \Sigma_c(v \circledast h^{\delta_{ci}})$

B8 $\mathcal{E}_{ijk} T_{jk} \Longrightarrow 0$

B9 $\Sigma_c(\nabla_{ij} \diamond (\mathcal{E}_{ijk} e)) \Longrightarrow 0$

B10 $\Sigma([], e) \Longrightarrow e$

B11 $\Sigma \chi e \Longrightarrow \chi(\Sigma e)$

Push operators to Fields The Apply expression is used to express the differentiation of a field. This extra constructor allows us to keep track of the field differentiation without allowing partial expressions to get lost in embedded ein expressions. The disadvantage of the constructor is extra book-keeping. The goal of Rules (C1-C6) is to push the apply operator down to the fields in the subexpression. The apply operator is pushed down until there are no more operators and the subexpression is just a constant, ρ, tensor or field expression(C1-C3). For instance (C1) supports expression $\nabla(s\varphi) \Longrightarrow s\nabla\varphi$. Although, the Diderot language has yet to support the product of fields, we have the chain rule (C4). The differentiation is pushed to the numerator of the division operator (C5). When Diderot supports the division of scalar fields this will be replaced with the quotient rule. The differentiation of a field, is expressed in the kernel of the convolution expression (C6). The differentiation index is pushed down to the kernels in a convolution expression , i.e. $\nabla(v \circledast h) \Longrightarrow v \circledast \nabla h$. At the end of the normalization phase the $\diamond$ expression is gone.

The goal of the probe rewrites is to push the probe down to field expressions. Non-Field operators are pushed outside the Probe (C7-C11). This helpful rewrite allows us to make computations on the tensor result rather than the entire field. $(-F)(x) \Longrightarrow -(F(x))$.

C1 $\nabla \diamond (\chi * e) \Longrightarrow \chi(\nabla \diamond e)$

C2 $\nabla \diamond (\rho * e) \Longrightarrow \rho(\nabla \diamond e)$

C3 $\nabla \diamond \Sigma(c, e) \Longrightarrow \Sigma(\nabla \diamond e)$

C4 $\nabla \diamond (e_1 * e) \Longrightarrow (e_1(\nabla \diamond e)) + (e(\nabla \diamond e_1))$

C5 $\nabla \diamond (\frac{e_1}{e_2}) \Longrightarrow \frac{(\nabla \diamond e_1)}{e_2}$

C6 $\nabla_p \diamond (v \circledast h^d) \Longrightarrow (v \circledast h^{d,p})$

C7 $(-e)@x \Longrightarrow -(e@x)$

C8 $(\chi e)@x \Longrightarrow \chi(e@x)$

C9 $(T_\alpha e)@x \Longrightarrow T_\alpha(e@x)$

C10 $(F/s)@, x \Longrightarrow (F@x)/s$

C11 $\Sigma(c, e)@x \Longrightarrow \Sigma(c, e@x)$

Concise Rules These rules just combine operators and flatten EIN expressions.

D1 $\frac{e_1}{e_2} es \Longrightarrow \frac{e_1 es}{e_2}$

D2 $e_1 * (e_2 * es) \Longrightarrow (e_1 * e_2 * es)$

D3 $\nabla_{[]} \diamond e \Longrightarrow e$

D4 $\nabla_i \diamond (\nabla_j \diamond e_1) \Longrightarrow (\nabla_{ij} \diamond e_1)$

D5 $\nabla \diamond \chi \Longrightarrow 0$

D6 $\Sigma_\nu(\Sigma_\mu(e)) \Longrightarrow \Sigma_{\nu\mu} e$

D7 $0e \Longrightarrow 0$

D8 $0 + e \Longrightarrow e$

Generic Rules Generally these rules apply when no other rules apply.

Negation

$$\frac{e \Longrightarrow e'}{-e \Longrightarrow -e'}$$

Addition, Subtraction

$$\frac{e_1 \Longrightarrow e_1' \quad \& \quad e_2 \Longrightarrow e_2'}{e_1 \theta e_2 \Longrightarrow e_1' \theta e_2'}$$

Product

$$\frac{e_1 \Longrightarrow e_1' \quad \& \quad es \Longrightarrow es'}{e_1 * es \Longrightarrow e_1' * es'}$$

Division

$$\frac{e_1 \Longrightarrow e_1' \quad \& \quad e_2 \Longrightarrow e_2'}{\frac{e_1}{e_2} \Longrightarrow \frac{e_1'}{e_2'}}$$

Summation

$$\frac{e \Longrightarrow e'}{\Sigma e \Longrightarrow \Sigma e'}$$

Probe

$$\frac{e \Longrightarrow e' \quad \& \quad x \Longrightarrow x'}{e@x \Longrightarrow e'@x'}$$

Apply

$$\frac{e_1 \Longrightarrow e_1' \quad \& \quad e_2 \Longrightarrow e_2'}{e_1 \diamond e_2 \Longrightarrow e_1' \diamond e_2'}$$

A.2 Proof

This section provides several proofs for the previously introduced index-based rules. $\mathcal{E}_{ijk}\mathcal{E}_{iqr} \Longrightarrow \delta_{jq}\delta_{kr} - \delta_{jr}\delta_{kq}$ (B.2). Consider the product of two $\mathcal{E}$ expressions as

$$\mathcal{E}_{ijk}\mathcal{E}_{pqr} \Longrightarrow \begin{vmatrix} \delta_{ip} & \delta_{iq} & \delta_{ir} \\ \delta_{jp} & \delta_{jq} & \delta_{jr} \\ \delta_{kp} & \delta_{kq} & \delta_{kr} \end{vmatrix}$$

$$\Longrightarrow \delta_{ip}(\delta_{jq}\delta_{kr} - \delta_{jr}\delta_{kq}) \quad +\delta_{iq}(\delta_{jr}\delta_{kp} - \delta_{jp}\delta_{kr}) \quad +\delta_{ir}(\delta_{jp}\delta_{kq} - \delta_{jq}\delta_{kp})$$

Then when the expressions have a common index such as $p = i$ we have

$$\implies \delta_{ii}\delta_{jq}\delta_{kr} - \delta_{ii}\delta_{jr}\delta_{kq} \quad \delta_{iq}\delta_{jr}\delta_{ki} - \delta_{iq}\delta_{ji}\delta_{kr} \quad \delta_{ir}\delta_{ji}\delta_{kq} - \delta_{ir}\delta_{jq}\delta_{ki}$$

and using $\delta_{xz}\delta_{zy} = \delta_{xy}$ and $\delta_{ii} = 3$ we have

$$\implies 3\delta_{jq}\delta_{kr} - 3\delta_{jr}\delta_{kq} \quad \delta_{kq}\delta_{jr} - \delta_{jq}\delta_{jr} \quad \delta_{jr}\delta_{kq} - \delta_{kr}\delta_{jq}$$

$$\implies \delta_{jq}\delta_{kr} - \delta_{jr}\delta_{kq}$$

Next, lets consider the rule (B.8). $\mathcal{E}_{ijk}\frac{\partial}{\partial x_i, x_j} \implies 0$
$\mathcal{E}_{ijk}\frac{\partial}{\partial x_i, x_j}$
$\implies \mathcal{E}_{ijk}\frac{\partial}{\partial x_j, x_i}$ swap order
$\implies -\mathcal{E}_{jik}\frac{\partial}{\partial x_j, x_i}$ anti-cyclic
$\implies -\mathcal{E}_{ijk}\frac{\partial}{\partial x_i, x_j}$ i →j j →i
$\implies 0$

A.3 Identities

This section walks through examples of tensor calculus identities found by using these rewrites. Consider the expressions,

$$e = a \times (b \times c)$$

The cross product is transformed to two operators as

$$d = \lambda(A, B)\langle \Sigma_{yz}\mathcal{E}_{xyz}A_yB_z\rangle_x(b, c)$$

$$e = \lambda(A, B)\langle \Sigma_{jk}\mathcal{E}_{ijk}A_jB_k\rangle_i(a, d)$$

In high-IL, the EIN operators would be represented with a single EIN operator. The outer index for EIN operator d x will be mapped to the variable indices in B_k. In this case $x \implies k$. The other variable indices in EIN operator d would be shifted $y \implies l$ and $z \implies m$.

$$d' = \lambda(A, B)\langle \Sigma_{lm}\mathcal{E}_{klm}A_lB_m\rangle_k(b, c)$$

The resulting operator is written as

$$e' = \lambda(A, B, C)\langle \Sigma_{jk}\mathcal{E}_{ijk}A_j(\Sigma_{lm}\mathcal{E}_{klm}B_lC_m)\rangle_i(a, b, c)$$

The expression is rewritten as

$$e' = \lambda(A, B, C)\langle \Sigma_{jklm}\mathcal{E}_{ijk}\mathcal{E}_{klm}A_jB_lC_m\rangle_i(a, b, c)$$

$$\implies \lambda(a, b, c)\langle \Sigma_{jlm}(\delta_{il}\delta_{jm}a_jb_lc_m) - \Sigma_{jlm}(\delta_{im}\ \delta_{jk}a_jb_lc_m)\rangle_i[B3, B2]$$

$$e \implies e'$$

$$\implies \lambda(a, b, c)\langle \Sigma_j(a_jb_ic_j) - \Sigma_j(a_jb_jc_i)\rangle_i[B4]$$

The tensor rewrites in the following table go through a similar process of rewriting.

Surface Language	Ein Ops
$(a \times b) \times c \Longrightarrow b(a \cdot c) - a(b \cdot c)$	$\lambda(u,v,t)\langle \Sigma_{jk}\mathcal{E}_{ijk}(\Sigma_{lm}\mathcal{E}_{jlm}u_l v_m)t_k\rangle_i$
	$\Longrightarrow \lambda(u,v,t)\langle \Sigma_k(u_k v_i t_k) - (u_i v_k t_k)\rangle_i$
$a \times (b \times c) \Longrightarrow b(a \cdot c) - c(a \cdot b)$	$\lambda(a,b,c)\langle \Sigma_{jk}\mathcal{E}_{ijk}a_j(\Sigma_{lm}\mathcal{E}_{klm}b_l c_m)\rangle_i$
	$\Longrightarrow \lambda(a,b,c)\langle \Sigma_{jlm}(\delta_{il}\delta_{jm} - \delta im\delta jk)a_j b_l c_m\rangle_i$
	$\Longrightarrow \lambda(a,b,c)\langle \Sigma_j(a_j b_i c_j) - (a_j b_j c_i)\rangle_i$
$(a \times b) \times (c \times d)$	$\Sigma_{jk}\mathcal{E}_{ijk}((\Sigma_{lm}\mathcal{E}_{jlm}a_l b_m)(\Sigma_{no}\mathcal{E}_{kno}c_n d_o))$
	$\Longrightarrow \Sigma_{klmno}\mathcal{E}_{kno}(\delta_{kl}\delta_{im} - \delta_{km}\delta_{il})(a_l b_m c_n d_o)$
	$\Longrightarrow (\Sigma_{kno}\mathcal{E}_{kno}a_k b_i c_n d_o) - (\Sigma_{kno}\mathcal{E}_{kno}a_i b_k c_n d_o)$
$(a \times b) \cdot (c \times d) \Longrightarrow (a \cdot c)(b \cdot d) - (a \cdot d)(b \cdot c)$	$\Sigma_i((\Sigma_{jk}\mathcal{E}_{ijk}a_j b_k)(\Sigma_{lm}\mathcal{E}_{ilm}c_l d_m))$
	$\Longrightarrow \Sigma_{jklm}(\delta_{jl}\delta_{km}a_j b_k c_l d_m) - (\delta_{jm}\delta_{kl}a_j b_k c_l d_m)$
	$\Longrightarrow \Sigma_{jk}(a_j b_k c_j d_k) - (a_j b_k c_k d_j)$

Another example is field identity $\nabla \cdot (\nabla \times F) \Longrightarrow 0$. This is expressed with two EIN operators as

$$c = \lambda f \left\langle \Sigma_{jk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}f_k \right\rangle_i (F)$$

$$t = \lambda f \left\langle \Sigma_i \frac{\partial}{\partial x_i} f_i \right\rangle (c)$$

Then in high-IL as one EIN operator as

$$t = \lambda f \left\langle \Sigma_i \frac{\partial}{\partial x_i}(\Sigma_{jk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}f_k) \right\rangle (F)$$

In the optimization stage this expression does through the following rewrites

$$\Longrightarrow \lambda f \left\langle \Sigma_i \Sigma_{jk}\frac{\partial}{\partial x_i}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}f_k \right\rangle (F)[C3]$$

$$\Longrightarrow \lambda f \left\langle \Sigma_{ijk}\frac{\partial}{\partial x_i}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}f_k \right\rangle (F)[D6]$$

$$\Longrightarrow \lambda f \left\langle \Sigma_{ijk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_i}\frac{\partial}{\partial x_j}f_k \right\rangle (F)[C2]$$

$$\Longrightarrow \lambda f \left\langle \Sigma_{ijk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_{ij}}f_k \right\rangle (F)[D4]$$

$$\Longrightarrow \lambda f \langle 0 \rangle (F)[B9]$$

Other field rewrites are expressed in the following table.

Surface Language	Ein Ops
$\nabla(s\varphi) \Longrightarrow s\nabla\varphi$	$\lambda(s\varphi)\left\langle \frac{\partial}{\partial x_i}s\varphi \right\rangle_i \Longrightarrow \lambda(s\varphi)\left\langle s\frac{\partial}{\partial x_i}\varphi \right\rangle_i$.
$\nabla \times \nabla\varphi \Longrightarrow 0$	$\lambda\varphi\left\langle \Sigma_{jk}\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}(\frac{\partial}{\partial x_k}\varphi) \right\rangle_i \Longrightarrow 0$
$\nabla \cdot (\nabla \times F) \Longrightarrow 0$	$\lambda f\left\langle \Sigma_i\frac{\partial}{\partial x_i}\Sigma_{jk}(\mathcal{E}_{ijk}\frac{\partial}{\partial x_j}f_k) \right\rangle (F) \Longrightarrow 0$
$\nabla(\varphi_1 + \varphi_2) \Longrightarrow \nabla\varphi_1 + \nabla\varphi_2$	$\lambda(\varphi_1\varphi_2)\left\langle \frac{\partial}{\partial x_i}(\varphi_1 + \varphi_2) \right\rangle_i$
	$\Longrightarrow \lambda(\varphi_1\varphi_2)\left\langle (\frac{\partial}{\partial x_i}\varphi_1) + (\frac{\partial}{\partial x_i}\varphi_2) \right\rangle_i$
$\nabla \times (F + G) \Longrightarrow (\nabla \times F) + (\nabla \times G)$	$\left\langle \frac{\partial}{\partial x_{\underline{0}}}(F_{\underline{1}} + G_{\underline{1}}) - \frac{\partial}{\partial x_{\underline{1}}}(F_{\underline{0}} + G_{\underline{0}}) \right\rangle$
	$\Longrightarrow \lambda(F,G)\left\langle \frac{\partial}{\partial x_{\underline{0}}}F_{\underline{1}} - \frac{\partial}{\partial x_{\underline{1}}}F_{\underline{0}} + \frac{\partial}{\partial x_{\underline{0}}}G_{\underline{1}} - \frac{\partial}{\partial x_{\underline{1}}}G_{\underline{0}} \right\rangle$