Imperial College London

Imperial College London

Department of Computing

Structure-preserving automatic differentiation and pull-backs in a language for variational forms

Author:
Gavin Truter

Supervisor:
Prof. Paul Kelly

Submitted in partial fulfillment of the requirements for the MSc degree in MSc Computing Science of Imperial College London

8 September 2017

Abstract

Unified Form Language (UFL) is a domain-specific language for expressing and manipulating the variational forms that arise when using the finite element method to approximate solutions to partial differential equations. One of these manipulations is the pull-back from an arbitrary finite element to an associated reference finite element, during which various quantities related to the Jacobian of the mapping between the reference and physical cells emerge. In some cases, the Jacobian-related factors cancel, but UFL was previously unable to recognise such cancellation. In this project, enhancements were made to preserve certain mathematical operators during transformations of the variational forms, so that Jacobian cancellation could be recognised, and to implement this cancellation. The success of the cancellation algorithm is shown on a number of forms. Finite element solvers that use UFL may now be adapted to extract a performance benefit from this enhancement.

Acknowledgements

I would like to thank Prof. Paul Kelly and Dr. David Ham for their excellent supervision. It was a pleasure to work with them.

I was grateful to be funded on the MSc Computing Science by a Commonwealth scholarship.

I would also like to thank my family for their support.

Contents

Chapter 1

Introduction

This chapter describes the setting in which the work of this project occurred, explains its objectives and achievements, and demonstrates its results with a representative example.

The project consisted of enhancements to Unified Form Language, a domain-specific language for expressing the variational forms that arise when using the finite element method to approximate solutions to partial differential equations. These enhancements preserve certain mathematical structures during transformations of the variational forms, which allows, in some cases, simplifications that improve the performance of the subsequent numerical approximation.

Section 1.1 discusses the finite element method and variational forms, highlighting the concepts and terminology used later. Section 1.2 describes Unified Form Language, while Section 1.3 summarises the enhancements added by this project and demonstrates the effect of these enhancements by an example. Finally, Section 1.4 explains the structure of the rest of this report.

1.1 The finite element method

Partial differential equations (PDEs) arise in the study of many physical phenomena. These equations can seldom be solved analytically, and so one must resort to finding numerical approximations of their solutions.

The finite element method is a popular approach to finding such an approximation. Brenner and Scott [8] is a standard reference for the mathematics of the finite element method.

In the finite element method, the PDE is rewritten into its variational

form: a solution of the PDE is a function $u \in V$ such that

$$F(u; v) = 0 \qquad \forall v \in V', \tag{1.1}$$

where V and V' are spaces of functions, and $F : V \times V' \to \mathbb{R}$ is a form. By a *form*, we mean a function from one or more spaces of functions to the reals. We separate the arguments u and v to F by a semicolon, rather than a comma, because F is linear in v but is not necessarily linear in u. The function F is typically a sum of integrals over the problem domain and its boundary.

For example, the variational form corresponding to the Poisson problem

$$-\operatorname{div}\operatorname{grad} u = f \tag{1.2}$$

on some two-dimensional domain Ω, where f is a given scalar-valued function, is (see Section 1.1 of Logg, Mardal, and Wells [18])

$$F(u; v) = \int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x - \int_\Omega f v \,\mathrm{d}x. \tag{1.3}$$

If the PDE is linear then (1.1) takes the form

$$a(u, v) - L(v) = 0 \qquad \forall v \in V' \tag{1.4}$$

where $a : V \times V' \to \mathbb{R}$ is bilinear (linear in each of its arguments) and $L : V' \to \mathbb{R}$ is linear, as in (1.3). The finite element method proceeds by replacing V and V' by finite-dimensional subspaces $V_h \subseteq V$ and $V_h' \subseteq V'$, leaving us to find $u_h \in V_h$ such that $a(u_h, v_h) - L(u_h) = 0$ for all $v_h \in V_h'$. This reduces to large linear system of equations, which gives an approximation u_h to the solution u of the original problem.

The subspaces V_h and V_h' are not specified directly. Instead, the domain of interest is partitioned into a mesh of cells. Finite-dimensional spaces of functions are specified on each cell, and these, together with conditions for the functions on the cell boundaries (such as continuity), imply a finite-dimensional space of functions on the entire domain.

For example, a rectangle in two dimensions might be split into many triangles, and on each of the triangles one might specify the three-dimensional space of linear functions, plus the condition that the functions in neighbouring triangles must agree on the common boundary, making the global function continuous.

These subdomains, the functions specified on them, and the way in which those functions are parameterised together constitute a *finite element*.

For performance reasons, calculations are not performed on each element individually. Instead, we specify a *reference element*, and for each element we

find a mapping F that transforms the reference element into the element in which we are interested. For example, for each triangle in the mesh mentioned above, one can find an affine mapping F such that the triangle is a result of applying F to the reference triangle with vertices at the origin, $(0, 1)$ and $(1, 0)$. Having found such an F, one expresses all the relevant aspects of the physical element in terms of the reference element. The process of rewriting the variational form in terms of functions on the reference element is called *pull-back*.

During pull-back, the determinant of the Jacobian matrix of F emerges in the form, due to the change of integration variables. The Jacobian matrix itself and its inverse may also emerge, either as a result of way in which functions are transformed between the physical element and the reference element, or because of the translation of gradients on the physical element to gradients on the reference element. If the mapping F is not affine (so that the Jacobian matrix varies over each subdomain) and the function spaces are high-dimensional (so that the value of the Jacobian matrix must be found at a large number of points in each subdomain) then the calculation of these quantities may become a significant computational cost. However, in some cases there is cancellation between these quantities, so that the pulled-back form does not depend upon the Jacobian at all, or only depends upon it through the sign of its determinant.

1.2 Firedrake and UFL

Firedrake [20, 13] is a tool for numerically approximating the solution of a PDE using the finite element method. The PDE is specified in its variational form in near-mathematical terms in Python, and Firedrake automatically generates efficient, parallelised code to find an approximate solution. Firedrake is developed and used by the Computing, Mathematics, and Earth Science and Engineering departments at Imperial College, amongst others.

Variational forms in Firedrake are represented in Unified Form Language (UFL) [3, 1]. Other components of Firedrake include the Two Stage Form Compiler (TSFC) [15], PyOP2 [21, 19], and the FInite element Automatic Tabulator (FIAT) [16]. Firdrake also makes use of solvers from the Portable, Extensible Toolkit for Scientific Computation (PETSc) [6, 5, 4] via the Python interface petsc4py [12]. UFL and FIAT are actually part of the FEniCS (pronounced "phoenix") project [2, 18]. FEniCS solves many of the same problems as Firedrake, but makes less use of symbolic mathematics to do so. The Firedrake project maintains its own versions of UFL and FIAT.

This project focuses exclusively on UFL, which is a domain-specific lan-

guage embedded in Python for representing and manipulating variational forms, including performing pull-backs. Before the work of this project, UFL used to perform *algebra lowering* before pull-backs: variational forms were re-expressed in lower terms by having dot products rewritten as indexed sums, divergences and curls rewritten as the appropriate sums of elements of gradients, and so on. This type of lowering must occur at some point in UFL, as it is necessary for other tools in the Firedrake stack; by performing it early, the number of operations that the pull-back procedure had to support was reduced. However, once algebra lowering had been performed, the Jacobian cancellations mentioned above could not be recognised, and so users bore the cost of Jacobian calculations even when these were unnecessary.

A form in UFL may also be a functional derivative: the derivative of another form with respect to one of its function arguments. An important step in UFL is the *application of derivatives*: the reduction, for example, of the gradient (or functional derivative) of a sum to the sum of the gradients (or functional derivatives), so that derivative operators are applied directly to functions or are eliminated completely. UFL used to apply derivatives after algebra lowering but before pull-backs.

1.3 Project objectives and achievements

The purpose of this project was to modify UFL such that the lowering of certain operators would be delayed until after pull-backs had been performed, so that Jacobian cancellations could be performed.

This consisted of the identification of appropriate operators to preserve, and the following five modifications:

1. Shifting the appropriate part of the algebra lowering until after function pull-backs.

2. Adding support for the vector operators in the application of spatial derivatives.

3. Adding support for the vector operators in the application of functional derivatives.

4. Expressing the pull-backs of physical-space derivatives in terms of the vector operators, including the recognition of two special cases where non-trivial simplifications are known.

5. Adding the explicit cancellation of Jacobian-related factors.

Jacobian cancellation in UFL is an important step towards improving the performance of Firedrake on forms where this cancellation occurs, especially in the cases of non-affine mappings F, where the Jacobian is not constant on each cell, and high-order elements, where there are many evaluation points within each cell. However, the realisation of this performance benefit was not an objective of the project. Possible steps towards this performance benefit are described in Section 8.2.

The following example form demonstrates the success of the project. In Chapter 7 this example is discussed in more detail, and further results are shown.

Define

$$a(q, f) = \int_K q \cdot \operatorname{grad} f \, \mathrm{d}x \tag{1.5}$$

where K is a domain and q and f are functions. Let $\partial a_f(q, f; v)$ be the functional derivative of a with respect to f in the direction of another function v. Then $\partial a_f(q, f; v)$ is itself a variational form.

Now let $\hat{K}$ be the reference space, let F be the mapping between K and $\hat{K}$, and let $\hat{q}$ and $\hat{v}$ be functions on $\hat{K}$ such that $q = \mathcal{F}^{\mathrm{div}}(\hat{q})$ and $v = \mathcal{F}^{\mathrm{id}}(\hat{v})$ (where $\mathcal{F}^{\mathrm{div}}$ and $\mathcal{F}^{\mathrm{id}}$ are particular function mappings, described in Chapter 4). Then we know, mathematically, that $\partial a_f(q, f; v)$ can be expressed in terms of these reference-space quantities as

$$\partial a_f(q, f; v) = \pm \int_{\hat{K}} \hat{q} \cdot \widehat{\operatorname{grad}} \, \hat{v} \, \mathrm{d}\hat{x} \tag{1.6}$$

where $\pm$ denotes the sign of the determinant of the Jacobian matrix of F on $\hat{K}$, and $\widehat{\operatorname{grad}}$ is the gradient on the reference space. In particular, this expression does not depend upon the Jacobian matrix of F, its inverse or its determinant, except through the sign of the determinant.

We can express and process this form in UFL. The resulting UFL expression, before the modifications made in this project, is shown in Listing 1. Terms related to the Jacobian have been highlighted in red. This expression is not worth examining in detail; the important point is that the Jacobian matrix, its inverse and its determinant all appear, and are separated by so many operations that it appears difficult to cancel them.

The resulting UFL expression, after the modifications made in this project, is shown in Listing 2. While this expression is much simpler than the previous one, this brevity is not in itself helpful. The important improvement of the second expression over the first is the absence of any quantities related to the Jacobian matrix except for the sign of its determinant.

Here and elsewhere, we say "before the modifications made in this project" and "after the modifications made in this project", which is shorthand for

the more complicated reality: The enhancements described in this report were made in the `vector_operator_derivatives` branch to the Firedrake version of UFL, which has not yet been merged into master. By "before the modifications made in this project", we mean "on the master branch of Firedrake UFL", specifically at revision `8caab`; by "after the modifications made in this project", we mean "on the `vector_operator_derivatives` branch" at revision `e600a`.

```
Product(
  Product(
    QuadratureWeight(domain),
    Abs(JacobianDeterminant(domain))),
  IndexSum(
    Product(
      Indexed(
        ComponentTensor(
          IndexSum(
            Product(
              Indexed(
                JacobianInverse(domain),
                MultiIndex((Index(14), Index(13)))),
              Indexed(
                ReferenceGrad(
                  ReferenceValue(v)),
                MultiIndex((Index(14),)))),
            MultiIndex((Index(14),))),
          MultiIndex((Index(13),))),
        MultiIndex((Index(8),))),
      Indexed(
        ComponentTensor(
          IndexSum(
            Product(
              Indexed(
                ComponentTensor(
                  Product(
                    Indexed(
                      Jacobian(domain),
                      MultiIndex((Index(9), Index(10)))),
                    Division(
                      FloatValue(1.0),
                      JacobianDeterminant(domain))),
                  MultiIndex((Index(9), Index(10)))),
                MultiIndex((Index(11), Index(12)))),
              Indexed(
                ReferenceValue(q),
                MultiIndex((Index(12),)))),
            MultiIndex((Index(12),))),
          MultiIndex((Index(11),))),
        MultiIndex((Index(8),)))),
    MultiIndex((Index(8),))))
```

Listing 1: The processed evaluation form, prior to this project.

```
Product(
  Product(
    QuadratureWeight(domain),
    Conditional(
      LT(
        JacobianDeterminant(domain),
        Zero((), (), ())),
      FloatValue(-1.0),
      FloatValue(1.0))),
  IndexSum(
    Product(
      Indexed(
        ReferenceGrad(
          ReferenceValue(v)),
        MultiIndex((Index(10),))),
      Indexed(
        ReferenceValue(q),
        MultiIndex((Index(10),)))),
    MultiIndex((Index(10),))))
```

Listing 2: The processed evaluation form, subsequent to this project.

1.4 Report structure

The remainder of the report is as follows.

Chapter 2 discusses the tensors, products and spatial derivatives used by UFL, and the rules for simplifying the spatial derivatives of these products.

Chapter 3 defines functional derivatives, proves and lists some of their properties, and shows how they are applied to certain variational forms.

Chapter 4 describes pull-backs, including the pull-backs of physical-space derivatives. The covariant and contravariant Piola mappings are defined, and situations in which Jacobian cancellation occurs are pointed out.

Chapter 5 introduces Unified Form Language and shows the processing of the same example form presented in this chapter, using UFL as it stood before the work of this project. The aspects of the pre-existing implementation of UFL that are relevant for this project are also described.

Finally, in Chapter 6, the contributions made in this project are described. This chapter applies the mathematics of Chapters 2, 3 and 4 to the pre-existing implementation of Chapter 5, resulting in modifications that allow implicit and explicit Jacobian cancellation.

The effectiveness of these modifications is demonstrated on example forms in Chapter 7.

Chapter 8 provides suggestions for further development and concludes.

Chapter 2

Tensor-valued functions and their spatial derivatives

Unified Form Language supports general tensor expressions with arbitrarily many dimensions, rather than just vector- or matrix-valued expressions. This complicates the definitions of the dot and inner products, and the gradient, divergence and curl operators. It also complicates the rules for the gradients, divergences and curls of dot and inner products. This chapter discusses the generalised definitions and rules in mathematical terms; their implementation in UFL is discussed in Chapter 5. We refer to the gradient, divergence and curl as *spatial derivatives* to distinguish them from the functional derivatives discussed in Chapter 3.

Here and throughout this report we deal only with real-valued PDEs.

2.1 Tensors

A vector is a collection of scalars indexed by a single index:

$$v = (v_i)_{i=1,\ldots,n}. \tag{2.1}$$

A matrix is a collection of scalars indexed by two indices:

$$A = (A_{ij})_{i=1,\ldots,n,\ j=1,\ldots,m}. \tag{2.2}$$

A *tensor*, for our purposes, is just the natural extension of these two concepts to an arbitrary number of indices:

$$x = (x_{i_1,\ldots,i_p})_{i_1=1,\ldots,n_1,\ \ldots,\ i_p=1,\ldots,n_p}. \tag{2.3}$$

Vectors and matrices are thus special cases of tensors. Spaces of tensors are denoted in the natural way, as $\mathbb{R}^{n_1 \times \ldots \times n_p}$.

2.2 Products

2.2.1 Inner product

Given a real vector space V, a function $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ is called an inner product on V if it is

- bilinear (linear in each argument),
- symmetric: $\langle x, y \rangle = \langle y, x \rangle$ for all $x, y \in V$, and
- positive definite: $\langle x, x \rangle \geq 0$ for all $x \in V$, and $\langle x, x \rangle = 0$ iff $x = 0$.

See Coleman [11, Section 6.1].

Inner products on function spaces are extremely important for the finite element method, but for our purposes the only relevant inner product is the most prosaic one: on the vector space $\mathbb{R}^{n_1 \times \ldots \times n_p}$ of rank-p tensors, we define

$$\langle x, y \rangle = \sum_{i_1=1}^{n_1} \cdots \sum_{i_p=1}^{n_p} x_{i_1,\ldots,i_p} \; y_{i_1,\ldots,i_p}. \tag{2.4}$$

That is, the inner product of two tensors of the same size is the sum of the component-wise products of their elements.

2.2.2 Dot product

The dot product of two elements x and y of $\mathbb{R}^n$ is given by

$$x \cdot y = \sum_{i=1}^{n} x_i y_i. \tag{2.5}$$

This is a special case of the inner product of the previous section.

UFL also uses the dot product in a more general sense, which includes both the usual dot product just mentioned and matrix multiplication. Given tensors x and y of sizes $n_1 \times \ldots \times n_p$ and $m_1 \times \ldots \times m_q$ respectively, with $n_p = m_1$, their dot product is defined to be the tensor of size $n_1 \times \ldots \times n_{p-1} \times m_2 \times \ldots \times m_q$ where the element in position $(i_1, \ldots i_{p-1}, j_2, \ldots j_q)$ is given by

$$\sum_{k=1}^{n_p} x_{i_1,\ldots i_{p-1},k} \; y_{k,j_2,\ldots,j_q}. \tag{2.6}$$

That is, the two "inner" dimensions must be equal, and they are removed, with summation over the products of their elements.

Here and in the sequel, the dot product (in this broader sense) of x and y is denoted $\operatorname{dot}(x, y)$ to distinguish it from the standard vector dot product. If x and y are (column) vectors and A is a matrix, then we have

$$\operatorname{dot}(x, y) = x \cdot y \qquad \text{(vector dot product)} \tag{2.7}$$

$$\operatorname{dot}(A, x) = Ax \qquad \text{(matrix pre-multiplication)} \tag{2.8}$$

$$\operatorname{dot}(x, A) = (x^T A)^T = A^T x \qquad \text{(matrix post-multiplication).} \tag{2.9}$$

2.3 Spatial derivatives

All of our functions are elements of Sobolev spaces, and so the derivatives that we take are meant as weak derivatives – see Section 1.2 of Brenner and Scott [8] or the brief summary thereof in Section 4.1.

2.3.1 Gradient

Traditionally, if u is a function from an open set $\Omega \subseteq \mathbb{R}^n$ to $\mathbb{R}$ whose partial derivatives exist everywhere, then the gradient of u is the function from Ω to $\mathbb{R}^n$ (i.e. the vector-valued function) whose entries are given by

$$(\operatorname{grad} u)_i = \frac{\partial u}{\partial x_i}. \tag{2.10}$$

This is generalised to tensors as follows: If u is a sufficiently smooth function from $\Omega \subseteq \mathbb{R}^n$ to the space $\mathbb{R}^{n_1 \times \ldots \times n_p}$ of rank-p tensors, then $\operatorname{grad} u$ is the function from Ω to the space $\mathbb{R}^{n_1 \times \ldots \times n_p \times n}$ of rank-$(p+1)$ tensors whose entries are given by

$$(\operatorname{grad} u)_{i_1,\ldots,i_p,i} = \frac{\partial u_{i_1,\ldots i_p}}{\partial x_i}. \tag{2.11}$$

Note that the gradient operator *appends* an axis to the tensor. UFL also provides a "nabla-gradient" operator which prepends an axis instead.

If $p = 1$, then $\operatorname{grad} u$, in this sense, is the *Jacobian matrix* of u. We will never use this term in this sense; in this report, "the Jacobian matrix" refers exclusively to the Jacobian matrix of the function mapping between the reference and physical finite element domains.

2.3.2 Divergence

The divergence operator is generalised similarly: if u is a sufficiently smooth function from Ω to the space $\mathbb{R}^{n_1 \times \ldots \times n_p \times n}$ of rank-$(p+1)$ tensors, then $\operatorname{div} u$

is the function from Ω to the space $\mathbb{R}^{n_1 \times \ldots \times n_p}$ of rank-p tensors whose entries are given by

$$(\operatorname{div} u)_{i_1,\ldots,i_p} = \sum_{i=1}^{n} \frac{\partial u_{i_1,\ldots,i_p,i}}{\partial x_i}. \tag{2.12}$$

Note that the length of the final axis of u must match the number of spatial dimensions. The divergence operator removes an axis from the back of the tensor; UFL also provides a "nabla-divergence" operator that removes an axis from the front, with the obvious restriction on the dimension of that axis.

2.3.3 Curl

Traditionally, if u is a map from an open subset Ω of $\mathbb{R}^3$ to $\mathbb{R}^3$ whose partial derivatives exist everywhere, then the curl of u is the map $\operatorname{curl} u : \Omega \to \mathbb{R}^3$ defined by

$$\operatorname{curl} u(x) = \begin{pmatrix} \frac{\partial u_3}{\partial x_2}(x) - \frac{\partial u_2}{\partial x_3}(x) \\ \frac{\partial u_1}{\partial x_3}(x) - \frac{\partial u_3}{\partial x_1}(x) \\ \frac{\partial u_2}{\partial x_1}(x) - \frac{\partial u_1}{\partial x_2}(x) \end{pmatrix}. \tag{2.13}$$

UFL does not generalize the curl operator to arbitrary tensors, but it does define curl in two other cases. In particular, for $u : \Omega \subseteq \mathbb{R}^2 \to \mathbb{R}$, we have

$$\operatorname{curl} u(x) = \begin{pmatrix} \frac{\partial u}{\partial x_2}(x) \\ -\frac{\partial u}{\partial x_1}(x) \end{pmatrix}, \tag{2.14}$$

and for $u : \Omega \subseteq \mathbb{R}^2 \to \mathbb{R}^2$, we have

$$\operatorname{curl} u(x) = \frac{\partial u_2}{\partial x_1}(x) - \frac{\partial u_1}{\partial x_2}(x). \tag{2.15}$$

2.4 Spatial derivatives of products

2.4.1 Spatial derivatives of a dot product

Gradient

We show the case of two-dimensional tensors, from which the general case can be inferred.

Let f and g be smooth maps from $\mathbb{R}^n$ to $\mathbb{R}^{p \times q}$ and $\mathbb{R}^{q \times r}$ respectively. Define $h : \mathbb{R}^n \to \mathbb{R}^{p \times r}$ by

$$h(x) = \operatorname{dot}(f(x), g(x)), \tag{2.16}$$

that is, the entries of h are given by

$$(h(x))_{ik} = \sum_{j=1}^{q} (f(x))_{ij} (g(x))_{jk}. \tag{2.17}$$

Then $\operatorname{grad} h$ is the map from $\mathbb{R}^n$ to $\mathbb{R}^{p\times r\times n}$ with entries given by

$$\begin{aligned}
(\operatorname{grad} h)_{ik\ell} &= \frac{\partial h_{ik}}{\partial x_\ell} && (2.18)\\
&= \sum_{j=1}^{q} \frac{\partial}{\partial x_\ell} \left[f_{ij} g_{jk} \right] && (2.19)\\
&= \sum_{j=1}^{q} \left\{ \frac{\partial f_{ij}}{\partial x_\ell} g_{jk} + f_{ij} \frac{\partial g_{jk}}{\partial x_\ell} \right\} && (2.20)\\
&= \sum_{j=1}^{q} (\operatorname{grad} f)_{ij\ell} g_{jk} + \sum_{j=1}^{q} f_{ij} (\operatorname{grad} g)_{jk\ell} && (2.21)\\
&= \sum_{j=1}^{q} (\operatorname{grad} f)_{ij\ell} g_{jk} + (\operatorname{dot}(f, \operatorname{grad} g))_{ik\ell}\,. && (2.22)
\end{aligned}$$

While the second term can be reduced to an element of a dot product, the second term cannot (at least without introducing a nabla-gradient). Only in the standard case that f and g are vector-valued do we have

$$\begin{aligned}
(\operatorname{grad} h)_\ell &= \sum_{j=1}^{q} (\operatorname{grad} f)_{j\ell} g_j + (\operatorname{dot}(f, \operatorname{grad} g))_\ell && (2.23)\\
&= (\operatorname{dot}(g, \operatorname{grad} f))_\ell + (\operatorname{dot}(f, \operatorname{grad} g))_\ell && (2.24)
\end{aligned}$$

and so

$$\operatorname{grad} h = \operatorname{grad} \operatorname{dot}(f, g) = \operatorname{dot}(g, \operatorname{grad} f) + \operatorname{dot}(f, \operatorname{grad} g). \tag{2.25}$$

It is easy to confuse the order of the arguments to the dot product in the first term.

Divergence

To examine the divergence, it is helpful to consider three-dimensional tensors. Let f and g be smooth maps from an open subset Ω of $\mathbb{R}^n$ to $\mathbb{R}^{p\times q\times r}$ and $\mathbb{R}^{r\times s\times n}$ respectively. Note that the final dimension of g matches the number of spatial dimensions. Define $h : \Omega \to \mathbb{R}^{p\times q\times s\times n}$ by

$$h(x) = \operatorname{dot}(f(x), g(x)), \tag{2.26}$$

so that the entries of h are given by

$$(h(x))_{ij\ell m} = \sum_{k=1}^{r} (f(x))_{ijk} (g(x))_{k\ell m}. \tag{2.27}$$

Then $\operatorname{div} h$ is the map from $\mathbb{R}^n$ to $\mathbb{R}^{p \times q \times s}$ with entries given by

$$\begin{aligned}
(\operatorname{div} h)_{ij\ell} &= \sum_{m=1}^{n} \frac{\partial h_{ij\ell m}}{\partial x_m} && (2.28)\\
&= \sum_{m=1}^{n} (\operatorname{grad} h)_{ij\ell mm} && (2.29)\\
&= \sum_{m=1}^{n} \left\{ \sum_{k=1}^{r} (\operatorname{grad} f)_{ijkm} g_{k\ell m} + \sum_{k=1}^{r} f_{ijk} (\operatorname{grad} g)_{k\ell mm} \right\} && (2.30)\\
&= \sum_{m=1}^{n} \sum_{k=1}^{r} (\operatorname{grad} f)_{ijkm} g_{k\ell m} + \sum_{k=1}^{r} f_{ijk} \sum_{m=1}^{n} (\operatorname{grad} g)_{k\ell mm} && (2.31)\\
&= \sum_{m=1}^{n} \sum_{k=1}^{r} (\operatorname{grad} f)_{ijkm} g_{k\ell m} + \sum_{k=1}^{r} f_{ijk} (\operatorname{div} g)_{kl} && (2.32)\\
&= \sum_{m=1}^{n} \sum_{k=1}^{r} (\operatorname{grad} f)_{ijkm} g_{k\ell m} + (\operatorname{dot}(f, \operatorname{div} g))_{ijl}\,. && (2.33)
\end{aligned}$$

As with the gradient, one term simplifies pleasantly but the other does not.

Here, the case in which both terms simplify is not the one in which f and g are both vector-valued, but the one in which f is vector-valued and g is matrix-valued, so that h is vector-valued and $\operatorname{div} h$ is scalar-valued; then the indices i, j and ℓ in (2.33) disappear, and we have

$$\begin{aligned}
\operatorname{div} h &= \sum_{m=1}^{n} \sum_{k=1}^{r} (\operatorname{grad} f)_{km} g_{km} + (\operatorname{dot}(f, \operatorname{div} g)) && (2.34)\\
&= \langle \operatorname{grad} f(x), g(x) \rangle + \operatorname{dot}(f, \operatorname{div} g). && (2.35)
\end{aligned}$$

A special case, not covered by the calculations above, is that where g is one-dimensional. In this case, we have

$$\begin{aligned}
(\operatorname{div} h)_i &= \sum_{j=1}^{q} (\operatorname{grad} h)_{ijj} && (2.36)\\
&= \sum_{j=1}^{q} \left\{ \sum_{k=1}^{r} (\operatorname{grad} f)_{ijkj} g_k + (\operatorname{dot}(f, \operatorname{grad} g))_{ijj} \right\} && (2.37)\\
&= \sum_{k=1}^{r} \left(\sum_{j=1}^{q} (\operatorname{grad} f)_{ijkj} \right) g_k + \sum_{j=1}^{q} (\operatorname{dot}(f, \operatorname{grad} g))_{ijj}\,, && (2.38)
\end{aligned}$$

which cannot be helpfully reduced.

Curl

There is no helpful simplification for the curl of a dot product.

2.4.2 Spatial derivatives of an inner product

Gradient

Again, we infer the general case from that of two-dimensional tensors.

Let f and g be smooth maps from an open subset Ω of $\mathbb{R}^n$ to $\mathbb{R}^{p\times q}$. Define $h : \Omega \to \mathbb{R}$ by

$$h(x) = \langle f(x), g(x)\rangle = \sum_{i=1}^{p}\sum_{j=1}^{q}(f(x))_{ij}(g(x))_{ij}. \tag{2.39}$$

Then $\operatorname{grad} h$ is the map from Ω to $\mathbb{R}^n$ with entries given by

$$(\operatorname{grad} h)_k = \sum_{i=1}^{p}\sum_{j=1}^{q}\frac{\partial}{\partial x_k}\left[f_{ij}g_{ij}\right] \tag{2.40}$$

$$= \sum_{i=1}^{p}\sum_{j=1}^{q}\left\{\frac{\partial f_{ij}}{\partial x_k}g_{ij} + f_{ij}\frac{\partial g_{ij}}{\partial x_k}\right\} \tag{2.41}$$

$$= \sum_{i=1}^{p}\sum_{j=1}^{q}\left\{(\operatorname{grad} f)_{ijk}g_{ij} + f_{ij}(\operatorname{grad} g)_{ijk}\right\}. \tag{2.42}$$

This cannot be reduced except when f and g are vector-valued, in which case the inner product is exactly the dot product and we have

$$\operatorname{grad} h = \operatorname{grad}\langle f, g\rangle = \operatorname{dot}(g, \operatorname{grad} f) + \operatorname{dot}(f, \operatorname{grad} g). \tag{2.43}$$

Divergence

An inner product is always a scalar, so it makes no sense to take its divergence.

Curl

Since the inner product is a scalar, its curl is only defined if $n = 2$. If there were a helpful simplification of the curl of an inner product, it would be in terms of the curls of the arguments to the inner products; but for these curls to be defined, the arguments must be functions from $\mathbb{R}^3$ to $\mathbb{R}^3$ (a contradiction), from $\mathbb{R}^2$ to $\mathbb{R}$ (in which case the inner product is just a standard product, for which there is no rule) or from $\mathbb{R}^2$ to $\mathbb{R}^2$. Even in this last case, there does not appear to be a helpful simplification.

Chapter 3

Functional derivatives

Unified Form Language (UFL) expresses variational forms. A variational form is a map from a sequence of function spaces to the reals. In the context of partial differential equations, these function spaces may include the spaces of test and trial functions, and the spaces of coefficients for the underlying problem, such as boundary conditions. In UFL, one is able to calculate the derivative of such a form. Since these derivatives are with respect to functions, we call them *functional derivatives*.

Part of this project was an extension to the functional derivative capabilities of UFL: the ability to calculate the functional derivatives of expressions containing dot and inner products, and the gradient, divergence and curl operators. This allows the preservation of these operators up to the point of pull-backs.

In this chapter, we provide a precise mathematical definition for functional derivatives, briefly show their usefulness, and prove or list the properties that are relevant for the vector operators. These properties are also used to calculate the functional derivatives in the forms that will be used, in Chapter 7, to evaluate the work of this project. The implementation of this extension to UFL will be discussed in Section 6.3.1.

3.1 The Fréchet derivative

The derivative of a form with respect to a function is easily defined: if L is a form, then its derivative at the function u in the direction of the function v is the real number

$$\lim_{\epsilon \to 0} \frac{L(u + \epsilon v) - L(u)}{\epsilon}. \tag{3.1}$$

if this limit exists. However, in order to differentiate complicated forms in an automated fashion, we need a more sophisticated definition than this. For

example, to differentiate

$$L(u) = \operatorname{grad} u \cdot \operatorname{grad} f, \tag{3.2}$$

we must be able to differentiate the dot product and the gradient operator, neither of which are forms. Crucially, we must also be able to combine the derivatives of these operators with a chain rule, which we cannot do given the simple definition (3.1).

Accordingly, we consider the functions, forms and operators with which we work as points in normed vector spaces and as functions between normed vector spaces. With this point of view, we use the derivative defined for maps between normed vector spaces for which a chain rule is valid: the Fréchet derivative.

3.1.1 Definition

We follow Ciarlet [9] closely in the definitions of this section.

Let X and Y be normed vector spaces over $\mathbb{R}$.

A mapping from X to Y is said to be a *linear operator* if

$$A(x+y) = A(x) + A(y) \tag{3.3}$$

for all $x, y \in X$ and

$$A(\alpha x) = \alpha A(x) \tag{3.4}$$

for all $x \in X$ and $\alpha \in \mathbb{R}$. In this case, it is common to write Ax for $A(x)$, and AB for $A \circ B$.

The set of continuous linear mappings from X to Y is denoted by $\mathcal{L}(X;Y)$. This set, together with an appropriately defined norm, is itself a normed vector space.

(Note that functions between general normed vector spaces, unlike functions from $\mathbb{R}$ to $\mathbb{R}$, may be linear without being continuous, if the domain X is infinite-dimensional. See Theorem 2.9-3 and the subsequent discussion in Ciarlet [9].)

Now let Ω be an open subset of X. Let f be a mapping from Ω to Y, and let $a \in \Omega$. We say that f is *differentiable at* a if there exists $f'(a) \in \mathcal{L}(X;Y)$ such that

$$f(a+h) = f(a) + f'(a)(h) + \|h\|\delta(h) \qquad \forall (a+h) \in \Omega \tag{3.5}$$

with $\lim_{h\to 0} \delta(h) = 0$. This element $f'(a)$ is unique, and is called the *Fréchet derivative* of f at a. If f is differentiable at all points of Ω, then it is said to be *differentiable*.

It is easily shown that if $f'(a)$ is defined then

$$\lim_{\epsilon \to 0} \frac{f(a + \epsilon h) - f(a)}{\epsilon} = f'(a)(h), \tag{3.6}$$

which shows that the Fréchet derivative satisfies our initial definition (3.1). However, a function can fail to be Fréchet differentiable even if this limit exists for all $a \in \Omega$ and $h \in X$.

If X is a product space $X_1 \times \ldots \times X_n$, so that f is a function of n arguments, then we can define the partial derivative of f at a with respect to the jth argument a_j, which we denote by $\partial_j f(a)$. See Ciarlet [9] for more details.

We have used here the notation from Ciarlet [9]: $f'(a)$ for the derivative of f at a and $\partial_j f(a)$ for its partial derivative with respect to the jth argument. However, many texts (such as Schwedes et al. [24]) use different notation: in terms of Ciarlet's notation, we have

$$\mathrm{d}f(u; w) := f'(u)(w) \tag{3.7}$$

and, for a function f of two variables u and v,

$$\partial f_u(u, v; w) := \partial_1 f(u, v)(w) \tag{3.8}$$

$$\partial f_v(u, v; w) := \partial_2 f(u, v)(w). \tag{3.9}$$

Note that here again we have used the convention that arguments in which a function is known to be linear follow the others and are separated from them by a semicolon.

3.1.2 Comparison with the conventional derivative

Let us compare the Fréchet derivative with the usual derivative of a function from $\mathbb{R}$ to $\mathbb{R}$; that is, the case of $X = Y = \mathbb{R}$. First, we note that the Fréchet derivative $f'(a)$ is not an element of Y, as in the calculus of the reals; it is instead an element of $\mathcal{L}(\mathbb{R}; \mathbb{R})$: a continuous, linear function from X to Y. Now $\mathcal{L}(\mathbb{R}; \mathbb{R})$ is not the space of straight lines in the plane, but the subset of these that pass through the origin, since, for $A \in \mathcal{L}(\mathbb{R}; \mathbb{R})$ and $x \in \mathbb{R}$ we have

$$A(x) = A(x1) = xA(1). \tag{3.10}$$

So A is the straight line through the origin with slope $A(1)$, and we can identify $A \in \mathcal{L}(\mathbb{R}; \mathbb{R})$ with $A(1) \in \mathbb{R}$. Now, if $f : \mathbb{R} \to \mathbb{R}$ is Fréchet differentiable at a then the usual derivative of f at a is

$$\lim_{h \to 0} \frac{f(a + h) - f(a)}{h} = \lim_{h \to 0} \frac{[f(a) + f'(a)(h) + |h|\delta(h)] - f(a)}{h} \tag{3.11}$$

$$= \lim_{h \to 0} \frac{hf'(a)(1) + |h|\delta(h)}{h} \tag{3.12}$$

$$= \lim_{h \to 0} \left[f'(a)(1) + \frac{|h|}{h}\delta(h) \right] \tag{3.13}$$

$$= f'(a)(1). \tag{3.14}$$

So, when $X = Y = \mathbb{R}$ and the Fréchet derivative exists, then the conventional derivative also exists. Using Taylor's theorem, we can easily show the converse as well: that if the conventional derivative exists, then so does the Fréchet derivative. In both cases, the Fréchet derivative is the linear function passing through the origin whose slope is equal to the conventional derivative.

3.2 Solving nonlinear PDEs with functional derivatives

In this section, we demonstrate one use of the functional derivative. For more detail, see Schwedes et al. [24].

Let us consider again the general variational problem: find $u \in V$ such that

$$F(u; v) = 0 \qquad \forall v \in V' \tag{3.15}$$

where V and V' are some function spaces and $F : V \times V' \to \mathbb{R}$. If the problem is nonlinear in u, then it cannot be solved directly by the finite element method. However, suppose that we have an initial approximation to the solution u. Then we can write, roughly, the truncated Taylor series

$$F(u + w; v) \approx F(u; v) + \partial F_u(u; v, w) \tag{3.16}$$

in the direction w. Specifically, we may be able to find w such that $F(u + w; v) \approx 0 \ \forall v \in V'$ by setting

$$\partial F_u(u; v, w) = -F(u; v) \qquad \forall v \in V'. \tag{3.17}$$

If we fix u at our initial approximation, then this is a *linear* variational problem in w with test functions v, which can be solved directly by the finite element method. Updating $u \leftarrow u + w$ and repeating the process, we can produce a (hopefully improving) succession of estimates to the solution of the original nonlinear problem by solving this sequence of linear problems.

3.3 Linear maps

Let $f \in \mathcal{L}(X;Y)$. Then for all $a \in X$, $h \in X$,

$$f(a+h) = f(a) + f(h) + \|h\|\delta(h) \tag{3.18}$$

with δ identically zero. Hence $f'(a) = f$ for all $a \in X$; in other notation, $\mathrm{d}f(a;h) = f(h)$ for all $a \in X$, $h \in X$.

Given a particular space of tensors, say $\mathbb{R}^m$ for concreteness, the gradient operator is a map $u \mapsto \operatorname{grad} u$ whose domain is some subset of $\mathbb{R}^m$-valued functions on an open subset Ω of $\mathbb{R}^n$ and whose codomain is a subset of $\mathbb{R}^{m\times n}$-valued functions on Ω. Considered as such, it is clearly linear. The domain is assumed to be regular enough that the gradient operator is also continuous. (Note that it is important to distinguish between the continuity of $\operatorname{grad} u$, which would be guaranteed if u was restricted to be continuously differentiable, and the continuity of the gradient operator itself.) Thus the Fréchet derivative of the gradient operator is

$$(\operatorname{grad})'(u)(h) = \operatorname{grad} h \tag{3.19}$$

or, in alternative notation, $\mathrm{d}\operatorname{grad}(u;h) = \operatorname{grad} h$.

Similarly, the divergence and curl operators are linear, continuous maps between spaces of functions, and so we have

$$(\operatorname{div})'(u)(h) = \operatorname{div} h \tag{3.20}$$

and

$$(\operatorname{curl})'(u)(h) = \operatorname{curl} h, \tag{3.21}$$

or, in the alternative notation, $\mathrm{d}\operatorname{div}(u;h) = \operatorname{div} h$ and $\mathrm{d}\operatorname{curl}(u;h) = \operatorname{curl} h$.

3.4 Bilinear maps

Let X, Y and Z be normed vector spaces. A map $B : X \times Y \to Z$ is said to be bilinear if it is linear in each argument.

Ciarlet [9] shows in Theorm 2.11-1 that a bilinear map is continuous if and only if the following quantity is finite:

$$\|B\| := \sup_{\substack{x\in X\setminus\{0\}\\ y\in Y\setminus\{0\}}} \frac{\|B(x,y)\|}{\|x\|\|y\|}. \tag{3.22}$$

Now, suppose that B is continuous. Then we can find the Fréchet derivative of B with respect to the pair (x,y), following Ciarlet [9]: we have

$$B(x+h,y+k) = B(x,y) + B(h,y) + B(x,k) + B(h,k) \tag{3.23}$$

$$= B(x,y) + B(h,y) + B(x,k) + \|(h,k)\|\delta((h,k)) \tag{3.24}$$

where

$$\begin{aligned}
\delta((h,k)) &= \frac{B(h,k)}{\|(h,k)\|} && (3.25)\\
&\leq \frac{\|B\|\|h\|\|k\|}{\max(\|h\|,\|k\|)} && (3.26)\\
&\leq \|B\|\max(\|h\|,\|k\|) && (3.27)\\
&= \|B\|\|(h,k)\| && (3.28)\\
&\to 0 \text{ as } (h,k) \to 0. && (3.29)
\end{aligned}$$

Also the mapping

$$(h,k) \mapsto B(h,y) + B(y,k) \tag{3.30}$$

is linear and continuous. Hence

$$B'(x,y)(h,k) = B(h,y) + B(x,k). \tag{3.31}$$

Any inner product is bilinear by definition and can be shown to be continuous (Theorem 4.1-1 in Ciarlet [9]). Hence its Fréchet derivative is given, in somewhat laboured notation, by

$$\langle\cdot,\cdot\rangle'(x,y)(h,k) = \langle h,y\rangle + \langle x,k\rangle . \tag{3.32}$$

The dot product is also bilinear and continuous, and so its Fréchet derivative is given by

$$\mathrm{dot}'(x,y)(h,k) = \mathrm{dot}\,(h,y) + \mathrm{dot}\,(x,k). \tag{3.33}$$

3.5 Chain rule

We require a chain rule so that we can calculate the Fréchet derivatives of complicated expressions. The following is Theorem 7.1-3 in Ciarlet [9], where a proof can be found.

Theorem 1 *Let X, Y and Z be normed vector spaces, and let U and V be open subsets of X and Y respectively. Let $f : U \to Y$ be differentiable at $a \in U$ and such that $f(U) \subseteq V$, and let $g : V \to Z$ be differentiable at $f(a)$. Then $g \circ f : U \to Z$ is differentiable at a, and*

$$(g \circ f)'(a) = g'(f(a))f'(a). \tag{3.34}$$

Note that (3.34) uses the shorthands mentioned above; more explicitly, the result is that

$$(g \circ f)'(a)(h) = g'\Big(f(a)\Big)\Big(f'(a)(h)\Big) \tag{3.35}$$

or, in the notation of Schwedes et al. [24],

$$\mathrm{d}(g \circ f)(a; h) = \mathrm{d}g\Big(f(a); \mathrm{d}f(a; h)\Big). \tag{3.36}$$

There are alternative definitions for derivatives of functions on normed vector spaces, such as the Gâteaux derivative. The Fréchet derivative is preferred precisely because it allows this chain rule.

3.6 Application to evaluation forms

The evaluation of this project in Chapter 7 uses three example forms, two of which involve functional derivatives. This section shows how the functional derivatives are applied in these forms.

3.6.1 First evaluation form

The first evaluation form, which was already presented in Section 1.3, is $\partial a_f(q, f; v)$ where

$$a(q, f) = \int_K q \cdot \operatorname{grad} f \,\mathrm{d}x, \tag{3.37}$$

K is some domain, and q and f are functions, q vector-valued and f scalar-valued. (More precisely, q and f are elements of particular Sobolev spaces, as described in Section 4.1.)

With a slight abuse of notation (shifting the subscript f), we have

$$\begin{aligned} \partial a_f(q, f; v) &= \partial_f \left[a(q, f)\right](v) && (3.38)\\ &= \partial_f \left[\int_K q \cdot \operatorname{grad} f \,\mathrm{d}x\right](v) && (3.39)\\ &= \int_K \partial_f \left[q \cdot \operatorname{grad} f\right](v) \,\mathrm{d}x && (3.40) \end{aligned}$$

since integration over a domain is linear and continuous, and using the chain rule. We proceed to apply the rule for vector dot products (which is a special case of that for the inner product (3.32)), and the rule for gradients (3.19):

$$\partial a_f(q, f; v) = \int_K \Big([\partial_f q](v) \cdot \operatorname{grad} f + q \cdot \partial_f[\operatorname{grad} f](v)\Big) \,\mathrm{d}x \tag{3.41}$$

$$= \int_K \Big([\partial_f q](v) \cdot \operatorname{grad} f + q \cdot \operatorname{grad} \left(\partial_f [f](v) \right) \Big) \, \mathrm{d}x. \tag{3.42}$$

Now, the Fréchet derivative of the constant mapping $G(f) = q$ is the zero mapping, for which $G'(f)(v)$ is the zero function from $\mathbb{R}^n$ to $\mathbb{R}$, since then

$$G(f + v) = q = G(f) + G'(f)(v) + \|v\| \delta(v) \tag{3.43}$$

with $\delta(v)$ identically zero. The Fréchet derivative of the identity mapping $H(f) = f$ is again the identity mapping, $H'(f)(v) = v$, since this is linear and continuous and

$$H(f + v) = f + v = H(f) + H'(f)(v) + \|v\| \delta(v) \tag{3.44}$$

with $\delta(v)$ again identically zero. Hence we have

$$\partial a_f(q, f; v) = \int_K \left(0 \cdot \operatorname{grad} f + q \cdot \operatorname{grad} v \right) \, \mathrm{d}x \tag{3.45}$$

$$= \int_K q \cdot \operatorname{grad} v \, \mathrm{d}x. \tag{3.46}$$

3.6.2 Second evaluation form

The second evaluation form is $\partial a_f(u, f; v)$ where

$$a(u, f) = \int_K u \operatorname{div} f \, \mathrm{d}x, \tag{3.47}$$

K is again a domain, and u and f are functions in some Sobolev spaces, u scalar-valued and f vector-valued.

Then, applying the appropriate rules, we have

$$\partial a_f(u, f; v) = \partial_f \left[\int_K u \operatorname{div} f \, \mathrm{d}x \right] (v) \tag{3.48}$$

$$= \int_K \partial_f \left[u \operatorname{div} f \right] (v) \, \mathrm{d}x \tag{3.49}$$

$$= \int_K \Big(\partial_f [u](v) \operatorname{div} f + u \partial_f \left[\operatorname{div} f \right] (v) \Big) \, \mathrm{d}x \tag{3.50}$$

$$= \int_K \Big(0 \operatorname{div} f + u \operatorname{div} \left(\partial_f [f](v) \right) \Big) \, \mathrm{d}x \tag{3.51}$$

$$= \int_K u \operatorname{div} v \, \mathrm{d}x. \tag{3.52}$$

In this case we have used a rule for the products of scalar-valued functions, which clearly follows the pattern of the rule for inner products.

In both this and the previous form, the net effect of the functional derivative has been to replace f with v in the original form. This is, of course, not the case in general.

Chapter 4

Pullbacks

This chapter briefly introduces Sobolev spaces and finite elements before moving on to discuss the reference-space to physical-space function mappings that give rise to the Jacobian-related quantities that we aim to cancel.

4.1 Sobolev spaces

Forms are maps from one or more function spaces to the reals. The function spaces that arise naturally in solving partial differential equations are called *Sobolev spaces*. Chapter 1 of Brenner and Scott [8] describe Sobolev spaces in detail; this section provides a very brief overview, following that text closely.

We work on a domain $\Omega \subseteq \mathbb{R}^n$. A multi-index $\alpha = (\alpha_1, \ldots, \alpha_n)$ is a tuple of indices. For sufficiently smooth functions $\phi : \Omega \to \mathbb{R}$, we define

$$D^\alpha \phi = \frac{\partial^{\alpha_1}}{\partial x_1^{\alpha_1}} \cdots \frac{\partial^{\alpha_n}}{\partial x_n^{\alpha_n}} \phi. \tag{4.1}$$

One can use an integration-by-parts formula to define a similar *weak derivative* $D_w^\alpha \phi$ on the relatively wide class of locally integrable functions. This weak derivative coincides with the classical derivative whenever the latter exists, but is also defined for functions that are not differentiable in the classical sense.

Let $p \in [0, \infty)$ and let k be a non-negative integer. Let A be the set of multi-indices α where $\sum_{i=1}^n \alpha_i \leq k$. Also, let f be a locally integrable function for which $D_w^\alpha f$ exists for all $\alpha \in A$. Then *Sobolev norm* of f is

$$\|f\|_{W_p^k(\Omega)} = \left(\sum_{\alpha \in A} \|D_w^\alpha f\|_{L^p(\Omega)}^p \right)^{\frac{1}{p}} \tag{4.2}$$

where $\|\cdot\|_{L^p(\Omega)}$ is the norm on $L^p(\Omega)$. (There is also a special case for $p = \infty$.) The Sobolev space $W_p^k(\Omega)$ is the set of functions for which this norm is defined and finite; it is a Banach space.

For given k, the space $W_2^k(\Omega)$, with $p = 2$, is also denoted $H^k(\Omega)$. With an appropriate definition for the inner product, $H^k(\Omega)$ is a Hilbert space. Very loosely, $H^k(\Omega)$ is the natural space of functions on which we can take up to k partial derivatives.

Equivalently, $H^k(\Omega)$ is the space of functions f in $L^2(\Omega)$ for which $D_w^\alpha f$ is in $L^2(\Omega)$ for all multi-indices α of order k or less. This is the definition given by Rognes, Kirby, and Logg [23], who define the spaces $H(\mathrm{div}; \Omega)$ and $H(\mathrm{curl}; \Omega)$ similarly: $H(\mathrm{div}; \Omega)$ is the space of functions in $L^2(\Omega, \mathbb{R}^n)$ whose divergences (in the sense of weak derivatives) are in $L^2(\Omega)$. For $n = 3$, $H(\mathrm{curl}; \Omega)$ is the space of functions in $L^2(\Omega, \mathbb{R}^3)$ whose curls are in the same space. Thus, loosely, $H(\mathrm{div}; \Omega)$ and $H(\mathrm{curl}; \Omega)$ are the natural spaces of functions on which we can take divergences and curls, respectively.

Rognes, Kirby, and Logg [23] give examples of problems in which $H(\mathrm{div})$ and $H(\mathrm{curl})$ naturally occur.

4.2 Finite elements

We have not yet stated a precise definition for a finite element. The following definition is taken from Brenner and Scott [8], who follow Ciarlet [10].

Definition 1 *Let*

1. $K \subseteq \mathbb{R}^n$ *be a bounded, closed set with a nonempty interior and a piecewise-smooth boundary,*
2. $\mathcal{P}$ *be a finite-dimensional space of functions from* K *to* $\mathbb{R}^m$*, and*
3. $\mathcal{N}$ *be a basis for* $\mathcal{P}' = \mathcal{L}(\mathcal{P}, \mathbb{R})$*, the space of continuous linear maps from* $\mathcal{P}$ *to* $\mathbb{R}$*.*

Then $(K, \mathcal{P}, \mathcal{N})$ *is called a* finite element*.*

The set K is called the *element domain*. The functions in $\mathcal{P}$ are called *shape functions*. The finitely many elements of $\mathcal{N}$ are called the *nodal variables*.

A simple example is the linear Lagrange triangle. Here K is a triangle with vertices z_1, z_2 and z_3 and $\mathcal{P}$ is the set of linear functions on K. Since $\mathcal{P}$ is three-dimensional, there must be three elements in $\mathcal{N}$; they are N_1, N_2 and N_3, each defined by

$$N_i(v) = v(z_i). \tag{4.3}$$

That is, the three nodal variables map any shape function to its values at the vertices of the triangle.

More complicated nodal variables include partial derivative values at certain points, and integrals of the function over the interior of the domain with some weight function. Kirby et al. [17] (Chapter 3 of [18]) provide extensive coverage of finite elements.

The *nodal basis* for a finite element is a basis $(\phi_j)_{j=1,\dots,k}$ for $\mathcal{P}$ such that $N_i(\phi_j) = \delta_{ij}$ where δ is the Kronecker delta. In other words, the nodal basis is a set of functions such that every element of $\mathcal{P}$ is a linear combination of these functions, and such that each function "triggers" exactly one of the nodal variables.

4.3 Reference elements

Instead of constructing finite elements on arbitrary domains directly, we construct such elements on reference domains, such as the triangle with vertices $(0,0)$, $(1,0)$ and $(0,1)$, or the tetrahedron with vertices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$. We then define elements on arbitrary domains in terms of these *reference elements*.

Let the reference element be $(\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{N}})$, and let the domain of interest be K. (We distinguish notationally between physical-space and reference-space quantities by adding "hats" to the latter.) We suppose that there is a smooth bijection F from $\hat{K}$ to K whose Jacobian DF is invertible on $\hat{K}$. This quantity, DF, is the Jacobian in whose cancellation we are interested. In many cases, F can be chosen to be affine, so that DF is constant; Jacobian cancellation is most important in cases where F is not affine.

We must also choose a mapping $\mathcal{F}$ from functions in $\hat{\mathcal{P}}$ (which are on $\hat{K}$) to functions on K that will form $\mathcal{P}$. The functions in $\hat{\mathcal{P}}$ usually form a subspace of a particular Sobolev space on $\hat{K}$, say $H(\mathrm{div}; \hat{K})$ for concreteness. If the mapping $\mathcal{F}$ is an isomorphism between $H(\mathrm{div}; \hat{K})$ and $H(\mathrm{div}; K)$, then $\mathcal{P}$ will form a subspace of $H(\mathrm{div}; K)$, and these two subspaces will have the same dimension.

In most cases, the obvious *identity* mapping

$$\mathcal{F}^{\mathrm{id}}(v) = v \circ F^{-1} \tag{4.4}$$

suffices: it is an isomorphism between $H^m(\hat{K})$ and $H^m(K)$ for all m. However, it is *not* an isomorphism between $H(\mathrm{div}; \hat{K})$ and $H(\mathrm{div}; K)$ or between $H(\mathrm{curl}; \hat{K})$ and $H(\mathrm{curl}; K)$, so when dealing with these function spaces, it is beneficial to work instead with mappings that are: the contravariant Piola

mapping $\mathcal{F}^{\text{div}}$ and the covariant Piola mapping $\mathcal{F}^{\text{curl}}$. These three mappings ($\mathcal{F}^{\text{id}}$, $\mathcal{F}^{\text{div}}$ and $\mathcal{F}^{\text{curl}}$) are discussed in the next three sections, following Rognes, Kirby, and Logg [23] and Boffi, Brezzi, and Fortin [7].

4.4 Identity mapping

The identity mapping between functions on the reference domain $\hat{K}$ and functions on the physical domain K is

$$\mathcal{F}^{\text{id}}(\hat{v}) = \hat{v} \circ F^{-1}. \tag{4.5}$$

This means that if $v = \mathcal{F}^{\text{id}}(\hat{v})$ and $x = F(\hat{x})$, then

$$v(x) = \hat{v}(\hat{x}). \tag{4.6}$$

The map $\mathcal{F}^{\text{id}}$ is an isomorphism from $H^m(\hat{K})$ to $H^m(K)$.

When considering gradients, it may be helpful to distinguish notationally between the gradients of physical-space functions and the gradients of reference-space functions, even though this is strictly unnecessary. We do so by denoting the latter by $\widehat{\text{grad}}$. The following result (equation 2.1.60 in Boffi, Brezzi, and Fortin [7]) relates $\operatorname{grad} v$ and $\widehat{\operatorname{grad}}\,\hat{v}$:

$$\operatorname{grad} v(x) = (DF(\hat{x}))^{-T}\,\widehat{\operatorname{grad}}\,\hat{v}(\hat{x}). \tag{4.7}$$

This is easily shown for scalar- or vector-valued v, as follows. Note that here we mix notation, using $D[v]$ to represent $(\operatorname{grad} v)^T$, the usual Jacobian matrix of v; elsewhere we reserve D for "the" Jacobian DF. We have

$$\begin{aligned} D[v](x) &= D[\hat{v} \circ F^{-1}](x) && (4.8)\\ &= D[\hat{v}](F^{-1}(x))D[F^{-1}](x) && (4.9)\\ &= D[\hat{v}](\hat{x})D[F^{-1}](F(\hat{x})) && (4.10)\\ &= D[\hat{v}](\hat{x})[DF(\hat{x})]^{-1} && (4.11) \end{aligned}$$

using the chain rule and the fact that F is invertible everywhere. Taking transposes, we obtain (4.7).

This is easily extended to tensor-valued v using the generalised dot product:

$$\operatorname{grad} v(x) = \operatorname{dot}\left(\widehat{\operatorname{grad}}\,\hat{v}(\hat{x}), (DF(\hat{x}))^{-1}\right). \tag{4.12}$$

4.5 Contravariant Piola mapping

The contravariant Piola mapping is

$$\mathcal{F}^{\mathrm{div}}(\hat{v}) = \frac{1}{\det DF}\, DF\, \hat{v} \circ F^{-1} \tag{4.13}$$

where $\det A$ means the determinant of the matrix A and $\hat{v}$ is assumed to be take values in $\mathbb{R}^n$. This means that if $v = \mathcal{F}^{\mathrm{div}}(\hat{v})$ and $x = F(\hat{x})$, then

$$v(x) = \frac{1}{\det DF(\hat{x})}\, DF(\hat{x})\, \hat{v}(\hat{x}). \tag{4.14}$$

The map $\mathcal{F}^{\mathrm{div}}$ is an isomorphism from $H(\mathrm{div}; \hat{K})$ to $H(\mathrm{div}; K)$, which explains the notation $\mathcal{F}^{\mathrm{div}}$.

Note that we define $\mathcal{F}^{\mathrm{div}}$ with the determinant of the Jacobian and not its absolute value, unlike much of the literature, including Boffi, Brezzi, and Fortin [7]. This follows the convention in UFL, which is explained by Rognes, Kirby, and Logg [23]. When we quote results from Boffi, Brezzi, and Fortin [7], the results are often adjusted accordingly by the sign of the determinant of the Jacobian.

The contravariant Piola mapping is usually used for functions in $H(\mathrm{div})$, and the natural derivative operator on such functions is the divergence. Again, we distinguish between the reference-space and physical-space divergences; the former will be written $\widehat{\mathrm{div}}$. The appropriate result is equation 2.1.71 in Boffi, Brezzi, and Fortin [7], adjusted for the definition of $\mathcal{F}^{\mathrm{div}}$:

$$\mathrm{div}\, v = \frac{1}{\det DF}\, \widehat{\mathrm{div}}\, \hat{v}. \tag{4.15}$$

This is easily shown for affine F. Letting tr denote the trace operator, and J the constant value of DF, we have

$$\begin{aligned}
\mathrm{div}\, v(x) &= \mathrm{tr}\left(D[v](x)\right) && (4.16)\\
&= \mathrm{tr}\left(D\left[\frac{1}{\det DF}\, DF\, \hat{v} \circ F^{-1}\right](x)\right) && (4.17)\\
&= \frac{1}{\det J}\, \mathrm{tr}\left(J\, D\left[\hat{v} \circ F^{-1}\right](x)\right) && (4.18)\\
&= \frac{1}{\det J}\, \mathrm{tr}\left(J\, D[\hat{v}](\hat{x})\, J^{-1}\right) && (4.19)\\
&= \frac{1}{\det J}\, \mathrm{tr}\left(D[\hat{v}](\hat{x})\right) && (4.20)\\
&= \frac{1}{\det J}\, \widehat{\mathrm{div}}\, \hat{v}(\hat{x}). && (4.21)
\end{aligned}$$

In (4.19) we have reused (4.8)-(4.11), and in (4.20) we have used the fact that $\mathrm{tr}(ABA^{-1}) = \mathrm{tr}(B)$ for any square matrices A and B, A invertible.

For non-constant DF, the proof is much more subtle; Boffi, Brezzi, and Fortin [7] defer it to Raviart and Thomas [22].

The mapping $\mathcal{F}^{\mathrm{div}}$ is easily extended for tensor-valued functions as

$$\mathcal{F}^{\mathrm{div}}(\hat{v}) = \frac{1}{\det DF} \operatorname{dot}(DF, \hat{v}) \circ F^{-1}. \tag{4.22}$$

The result (4.15) holds unchanged for such functions. For example, let v be a two-dimensional tensor with the second dimension matching the number of spatial dimensions. Then the entries of $\operatorname{div} v$ are given by

$$(\operatorname{div} v)_i = \operatorname{div} v_{i*} \tag{4.23}$$

$$= \frac{1}{\det DF} \widehat{\operatorname{div}}\, \hat{v}_{i*} \tag{4.24}$$

$$= \left(\frac{1}{\det DF} \widehat{\operatorname{div}}\, \hat{v} \right)_i, \tag{4.25}$$

where we have used v_{i*} to denote the vector of elements $(v_{ij})_{j=1,\ldots,n}$.

4.6 Covariant Piola mapping

The covariant Piola mapping is

$$\mathcal{F}^{\mathrm{curl}}(\hat{v}) = DF^{-T}\, \hat{v} \circ F^{-1} \tag{4.26}$$

where $n = 3$ and $\hat{v}$ is assumed to take values in $\mathbb{R}^3$.

Again, this means that if $v = \mathcal{F}^{\mathrm{curl}}(\hat{v})$ and $x = F(\hat{x})$, then

$$v(x) = DF^{-T}(\hat{x})\, \hat{v}(\hat{x}). \tag{4.27}$$

The map $\mathcal{F}^{\mathrm{curl}}$ is an isomorphism from $H(\mathrm{curl}; \hat{K})$ to $H(\mathrm{curl}; K)$.

The derivative operator naturally associated with this mapping is the curl. Once again, we distinguish between the curls of physical-space and reference-space functions, denoting the latter by $\widehat{\mathrm{curl}}$. The appropriate result is equation 2.1.92 in Boffi, Brezzi, and Fortin [7], who defer proof to Girault and Raviart [14]:

$$\operatorname{curl} v = \frac{1}{|\det DF|} DF\, \widehat{\operatorname{curl}}\, \hat{v}. \tag{4.28}$$

We do not have to worry about the general tensor case, as in UFL one may only take the curl of a vector. However, one can take the curls of functions from $\mathbb{R}^2$ to $\mathbb{R}$, and from $\mathbb{R}^2$ to $\mathbb{R}^2$. In the former case, the mapping $\mathcal{F}^{\mathrm{curl}}$ does not make sense, as v and $\hat{v}$ would take values in different spaces. In the latter case, the mapping does make sense, and it is possible that (4.28) might be extended for this situation, but this extension was not attempted.

4.7 Jacobian cancellation

If $v = \mathcal{F}^{\mathrm{div}}(\hat{v})$, then $\operatorname{div} v$ is given by

$$\operatorname{div} v = \frac{1}{\det DF} \widehat{\operatorname{div}}\, \hat{v}, \qquad \text{(repeat of 4.15)}$$

which does not involve the Jacobian matrix or its inverse. This occurs because the inverse Jacobian matrix that occurs in the transformation $\mathcal{F}^{\mathrm{div}}$ is cancelled, roughly speaking, by a Jacobian matrix that emerges from the change of variables F.

Similar Jacobian cancellations can occur elsewhere. Boffi, Brezzi, and Fortin [7] give results such as the following: for $v = \mathcal{F}^{\mathrm{id}}(\hat{v})$ and $q = \mathcal{F}^{\mathrm{div}}(\hat{q})$,

$$\int_K q \cdot \operatorname{grad} v \,\mathrm{d}x = \pm \int_{\hat{K}} \hat{q} \cdot \widehat{\operatorname{grad}}\, \hat{v} \,\mathrm{d}\hat{x} \tag{4.29}$$

(2.1.72 in [7]) and, with $u = \mathcal{F}^{\mathrm{curl}}(\hat{u})$,

$$\int_K u \cdot q \,\mathrm{d}x = \pm \int_{\hat{K}} \hat{u} \cdot \hat{q} \,\mathrm{d}\hat{x}. \tag{4.30}$$

Here we have written $\pm$ for the sign of the determinant of the Jacobian, which must be added to the results from [7] because of our definition of $\mathcal{F}^{\mathrm{div}}$, without the absolute value. Note that in these two equations, the determinant of the Jacobian has been cancelled too, except for its sign.

These identities can be derived from the rules provided above as follows: for (4.29), we have

$$\begin{aligned}
q(x) \cdot \operatorname{grad} v(x) &= q(x)^T \operatorname{grad} v(x) && (4.31)\\
&= \left[\frac{1}{\det DF(\hat{x})} DF(\hat{x})\hat{q}(\hat{x})\right]^T \left[(DF(\hat{x}))^{-T} \widehat{\operatorname{grad}}\, \hat{v}(\hat{x})\right] && (4.32)\\
&= \frac{1}{\det DF(\hat{x})} (\hat{q}(\hat{x}))^T (DF(\hat{x}))^T (DF(\hat{x}))^{-T} \widehat{\operatorname{grad}}\, \hat{v}(\hat{x}) && (4.33)\\
&= \frac{1}{\det DF(\hat{x})} (\hat{q}(\hat{x}))^T \widehat{\operatorname{grad}}\, \hat{v}(\hat{x}) && (4.34)\\
&= \frac{1}{\det DF(\hat{x})} (\hat{q}(\hat{x})) \cdot \widehat{\operatorname{grad}}\, \hat{v}(\hat{x}) && (4.35)
\end{aligned}$$

and so

$$\begin{aligned}
\int_K q \cdot \operatorname{grad} v \,\mathrm{d}x &= \int_{\hat{K}} q(x) \cdot \operatorname{grad} v(x) |\det DF(\hat{x})| \,\mathrm{d}\hat{x} && (4.36)\\
&= \int_{\hat{K}} \frac{1}{\det DF(\hat{x})} (\hat{q}(\hat{x})) \cdot \widehat{\operatorname{grad}}\, \hat{v}(\hat{x}) |\det DF(\hat{x})| \,\mathrm{d}\hat{x} && (4.37)
\end{aligned}$$

$$= \pm \int_{\hat{K}} (\hat{q}(\hat{x})) \cdot \widehat{\operatorname{grad}}\, \hat{v}(\hat{x})\, \mathrm{d}\hat{x}. \tag{4.38}$$

In the last step, the sign of the determinant of the Jacobian is brought outside of the integration, as it is constant on $\hat{K}$.

Similarly, for (4.30), we have

$$u(x) \cdot q(x) = \left[DF^{-T}(\hat{x})\hat{v}(\hat{x})\right]^T \left[\frac{1}{\det DF(\hat{x})} DF(\hat{x})\hat{q}(\hat{x})\right] \tag{4.39}$$

$$= \frac{1}{\det DF(\hat{x})} (\hat{v}(\hat{x}))^T (DF(\hat{x}))^{-1} DF(\hat{x})\hat{q}(\hat{x}) \tag{4.40}$$

$$= \frac{1}{\det DF(\hat{x})} \hat{v}(\hat{x}) \cdot \hat{q}(\hat{x}) \tag{4.41}$$

and so

$$\int_K u(x) \cdot q(x)\, \mathrm{d}x = \int_{\hat{K}} u(x) \cdot q(x) |\det DF(\hat{x})|\, \mathrm{d}\hat{x} \tag{4.42}$$

$$= \pm \int_{\hat{K}} \hat{v}(\hat{x}) \cdot \hat{q}(\hat{x})\, \mathrm{d}\hat{x}. \tag{4.43}$$

These two cases illustrate that not all Jacobian cancellation occurs directly, as in the divergence of an $\mathcal{F}^{\mathrm{div}}$-mapped function (4.15). The simplifications in these short proofs are not obvious to a computer algebra system, and if they are desired, they must be sought out. This sort of Jacobian cancellation was implemented as part of this project, as described in Section 6.5.

Chapter 5

Unified Form Language

This chapter discusses the representation of variational forms in Unified Form Language (UFL), and the algorithms by which spatial derivatives, functional derivatives and pull-backs are applied to these forms. This lays the groundwork for Chapter 6, which discusses the modifications and additions made to these algorithms to allow Jacobian cancellation. Everything presented in this chapter thus relates to UFL before these modifications and additions, or to UFL in general, both before and after this work.

5.1 Specifying forms

The objects of interest in UFL are *variational forms* as described in Section 2 of Alnæs et al. [3]. A variational form is a mapping from a product of function spaces to the reals.

These function spaces are divided into *argument spaces* and *coefficient spaces*. A form is linear in its *arguments*, but potentially nonlinear in its *coefficients*. Note the potential confusion in that we separate the arguments to a form (using the word "argument" in its usual mathematical meaning) into arguments (now using the term in the specific sense just mentioned) and coefficients. Coefficients represent functions in terms of which the PDE is parameterised, such as the heat source over the domain. There are usually just two arguments, the test and trial functions.

A form in UFL is a sum of integrals, each of which is over one of three possible domains: the interiors of the cells of the finite element mesh, the exterior facets of the mesh, or the interior facets of the mesh. For simplicity, we focus on integrals over the interiors of the cells.

Each integrand must be a scalar-valued *expression*. We are interested in the functional derivatives and pull-backs of forms, but, since forms are sums

of integrals, and differentiation and pull-backs pass through both summation (by linearity) and integration (by assumptions of smoothness), we often consider expressions rather than forms.

To define expressions and forms in UFL, one typically starts by importing all the relevant names:

```
from ufl import *
```

In order to form useful expressions in UFL, we need first to construct finite elements. For simplicity, we can do this on a single cell. First, we specify the cell itself:

```
cell = triangle
```

Alternatively, we might specify a tetrahedron, for a three-dimensional domain.

In either case, finite elements can now be specified, for example as follows:

```
cg_element = FiniteElement('CG', cell, degree=2)
rt_element = FiniteElement('RT', cell, degree=1)
```

This forms two finite elements on the cell. In the first, `CG` stands for Continuous Galerkin: the finite element function space consists of scalar-valued second degree polynomials on the cell (and that, if we had more than one cell, would be continous over cell boundaries). This space uses the identity mapping. In the second, `RT` stands for Raviart-Thomas: this $H(\mathrm{div})$-conforming element provides vector-valued functions on the cell, and uses the contravariant Piola mapping.

We create a pair of coefficients and an arguments like this:

```
q = Coefficient(rt_element)
f = Coefficient(cg_element)
v = Argument(cg_element, 0)
```

Coefficients are automatically numbered, while arguments must be numbered manually. Arguments 0 and 1 can also be created by `TrialFunction` and `TestFunction`, but we prefer to number arguments manually, since they may not have interpretations as test and trial functions.

We can now create forms in an obvious way: the form

$$a(q, f) = \int_K q \cdot \mathrm{grad}\, f \,\mathrm{d}x \tag{5.1}$$

is constructed by

```
a = dot(q, grad(f)) * dx
```

The multiplication by `dx` means an integral over the interiors of the cells. This is the form for which final results were displayed in Section 1.3, and which will be discussed further in Section 7.1.

UFL forms and expressions are expression trees, and can be converted to a graphical form. In particular, UFL includes utilities to convert them to the popular DOT graph description language, from which the `dot` program can convert them to an image format. The expression tree for `a` is shown in Figure 5.1. We use UFL's "compact" representation, and replace UFL's representations of the coefficients and arguments with our variable names `q`, `f` and `v`, of which UFL is unaware.

Functional derivatives can now be formed using `derivative`: the functional derivative of `a` with respect to `f` in the direction `v`,

$$\partial a_f(q, f; v) \tag{5.2}$$

is specified in UFL by

```
a_prime = derivative(a, f, v)
```

We take derivatives with respect to coefficients because forms and expressions may be nonlinear in them; the directions are arguments because the derivative is linear in the direction. The expression tree for `a_prime` is shown in Figure 5.2.

Functional derivatives are represented by the `CoefficientDerivative` class in UFL. The first operand is the expression being differentiated. The second and third operands are the coefficient with respect to which the derivative is being taken, and the direction of the derivative; each object can in fact represent several differentiations with respect to different coefficients, so these operands are lists. The fourth operand represents any explicitly specified relationships between the differentiation coefficient and any other coefficients in the form.

Note that the functional derivative has not yet been simplified: it is akin to the expression $\frac{\mathrm{d}}{\mathrm{d}x}x^2$. The process akin to converting this to $2x$ is called *applying* the functional derivative, and is discussed in the next section.

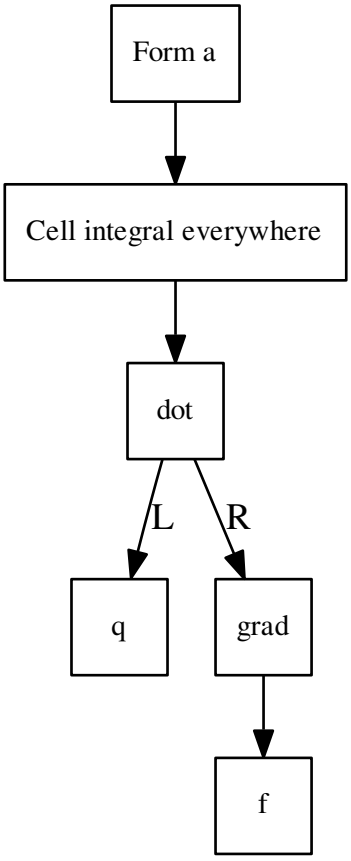

Figure 5.1: The expression tree for $a(q, f)$.

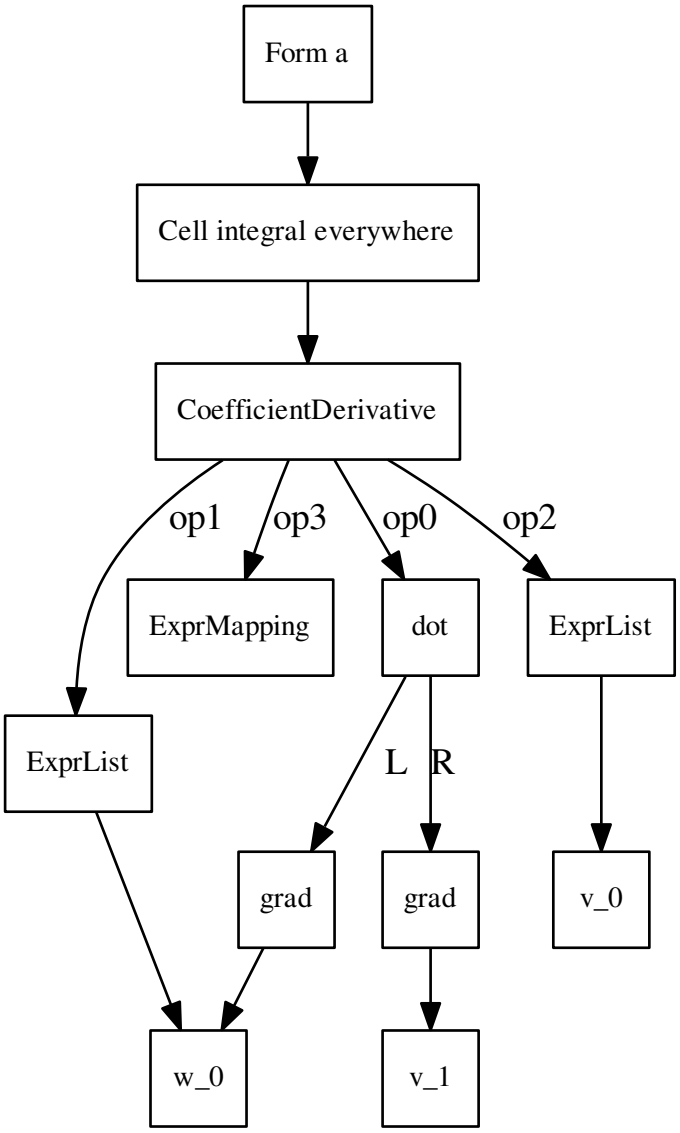

Figure 5.2: The expression tree for $\partial a_f(q, f; v)$.

5.2 Computing form data

Once a form has been specified in UFL, it must have various tranformations applied in order to prepare it for processing by a form compiler. The function `compute_form_data` performs this series of transformations. In this section, we perform the relevant transformations individually to examine their effects. We use UFL without any of the modifications added in this project.

5.2.1 Algebra lowering

The first step of `compute_form_data` is to perform *algebra lowering*: to rewrite the form such that it does not contain nodes such as dot products, inner products, cross products, divergences, curls, matrix transposes, matrix inverses, matrix determinants and cofactor matrices.

The expression tree for our example form, following algebra lowering, is too large to display in its entirety, but the lowered form of the original integrand, $q \cdot \operatorname{grad} f$, is displayed in Figure 5.3.

Starting at the bottom of the tree, we have q and $\operatorname{grad} f$. The next nodes up, indicated by `[]`, are called `Indexed` by UFL: they "split up" the vector quantities q and $\operatorname{grad} f$ into their components q_{i_8} and $(\operatorname{grad} f)_{i_8}$. The next node forms the products of these components, and the final node sums these products, thus representing

$$\sum_{i_8} q_{i_8} (\operatorname{grad} f)_{i_8}, \tag{5.3}$$

a lowered version of $q \cdot \operatorname{grad} f$.

5.2.2 Applying derivatives

Applying derivatives means creating an equivalent expression in which derivatives (functional or spatial) have been pushed towards the leaves of the expression tree as far as possible. In particular, after the application of derivatives, functional derivatives should be eliminated and gradients should operate only on coefficients, arguments or other gradients.

The result of applying the derivatives in our example form is shown in Figure 5.4.

If we had attempted to apply derivatives before performing algebra lowering, we would have received an error: "Missing differentiation handler for type Dot. Have you added a new type?" Derivative application was only supported for types that algebra lowering allowed to remain.

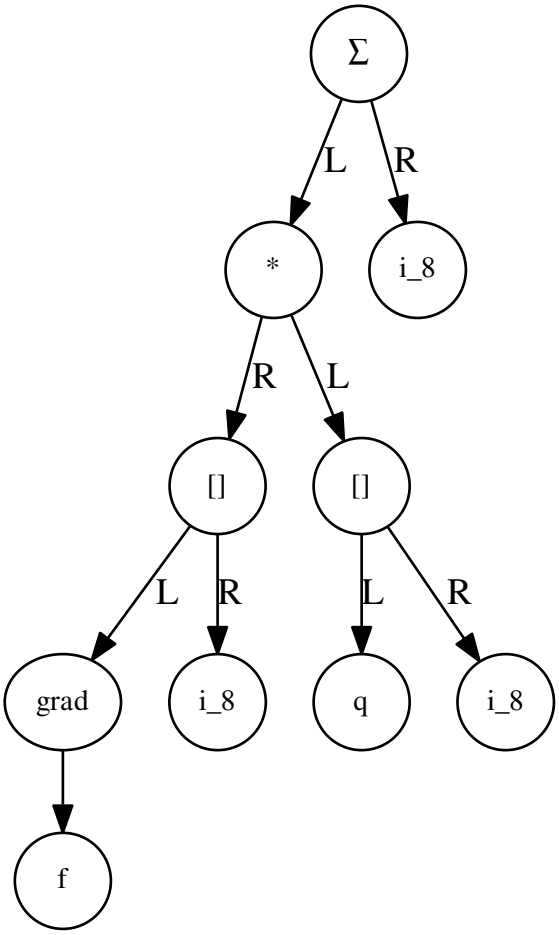

Figure 5.3: The expression tree for the integrand $q \cdot \operatorname{grad} f$ after algebra lowering.

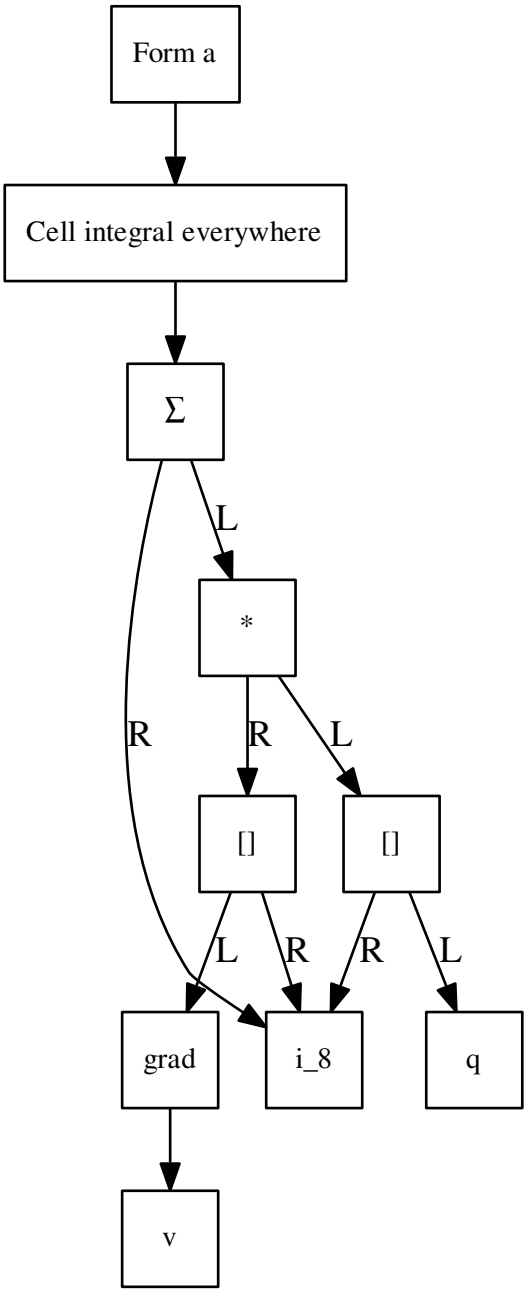

Figure 5.4: The expression tree for $\partial a_f(q, f; v)$ after the application of derivatives.

5.2.3 Applying pull-backs

The next step of `compute_form_data` is to pull back functions.

In mathematical terms, we have a mapping $\mathcal{F} \in \{\mathcal{F}^{\mathrm{id}}, \mathcal{F}^{\mathrm{div}}, \mathcal{F}^{\mathrm{curl}}\}$ between functions on the reference space $\hat{v}$ and functions on the physical space $v = \mathcal{F}(\hat{v})$. In UFL terms, the reference-space function associated with a physical-space coefficient or argument is represented by a `ReferenceValue` wrapping the coefficient or argument.

In our example form, the node for `q` is replaced by the set of nodes shown in Figure 5.5.

Near the bottom of the tree, the Jacobian `J` and its determinant `detJ` can be seen, each represented by its own type. The Jacobian is split into its components, each of which is multiplied by one over the determinant; the components are then rejoined in the `ComponentTensor` node indicated by `][` and split up again. On the right, we can see the reference value of `q` (that is, $\hat{q}$), which is also split into its components. The combination of the product, single-indexed sum and `ComponentTensor` at the top of the tree constitutes a matrix multiplication expressed component-wise. This large tree thus accurately represents

$$q = \frac{1}{\det DF}\, DF\, \hat{q}. \tag{5.4}$$

Since `v` uses the identity mapping, it is converted directly to its reference value.

5.2.4 A second application of derivatives

The application of pull-backs has left us with the physical-space gradient of the reference value of `v`. This is not a meaningful mathematical concept; instead, it represents a halfway stage in the conversion

$$\operatorname{grad} v \to \operatorname{dot}\left(\widehat{\operatorname{grad}}\, \hat{v}, (DF)^{-1}\right).$$

The remainder of this conversion is achieved by a second application of derivatives. While this call also pushes derivative operators towards the leaves of the expression tree, its essential effect is to convert physical-space gradients into reference-space gradients.

The transformed version of the subtree for `Grad(v)` is displayed in Figure 5.6.

The lower right subtree is the reference gradient of the reference value of `v`; that is, $\widehat{\operatorname{grad}}\, \hat{v}$. On the left, the inverse Jacobian `K` is also evident. These two trees are combined by the component-wise representation of a matrix multiplication.

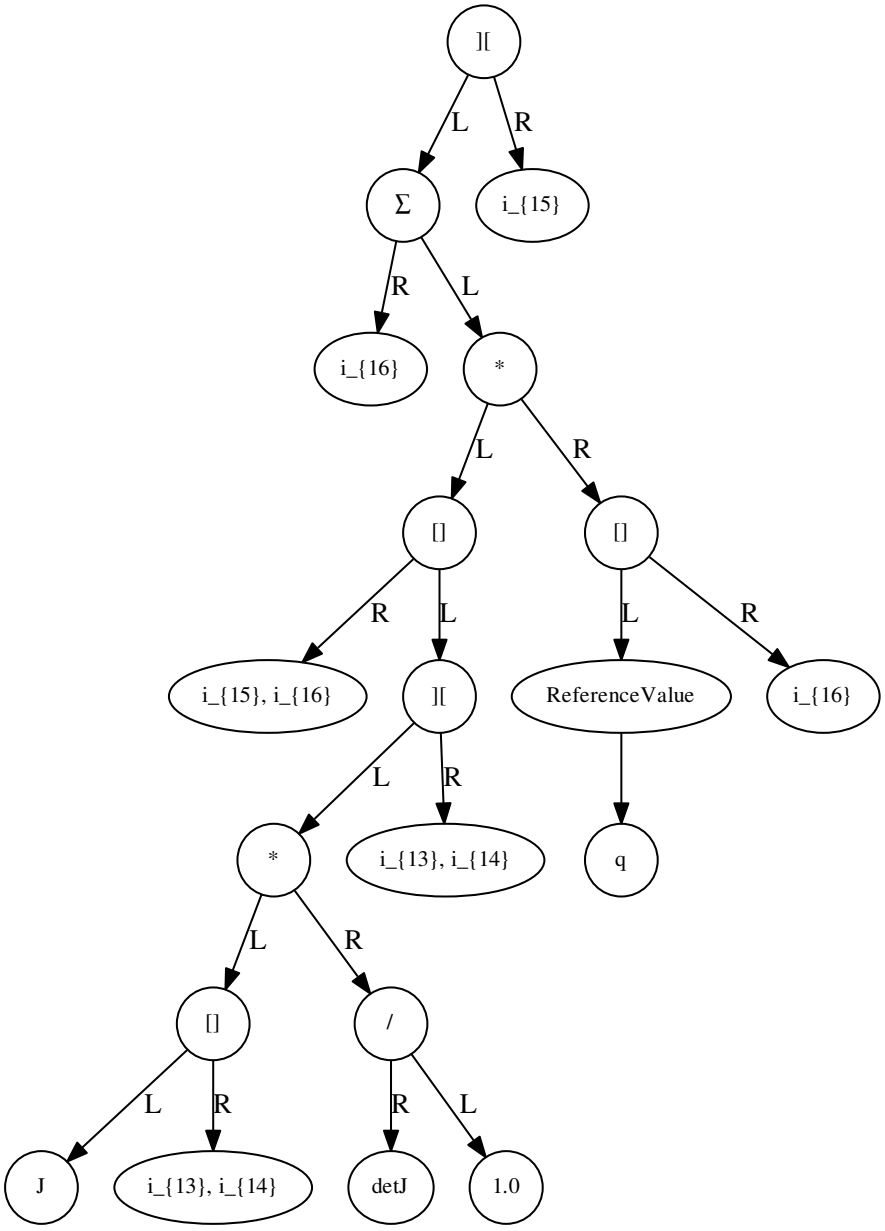

Figure 5.5: The pulled-back representation of q.

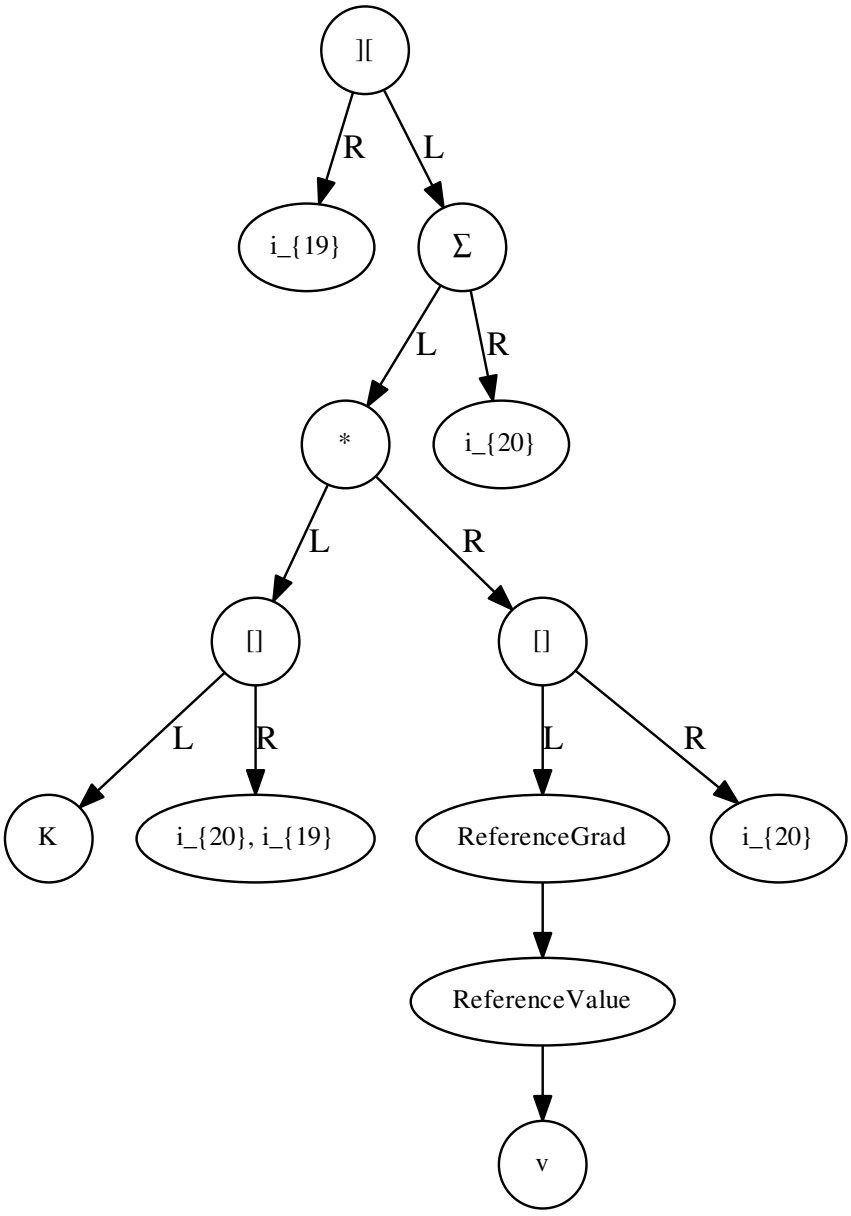

Figure 5.6: The pulled-back representation of v, after the second application of derivatives.

5.2.5 Applying integral scaling

The final step of the conversion to reference element quantities, and the last relevant step of `compute_form_data`, is to apply integral scaling; that is, to add the factor $|\det DF|$, which arises from the change of variables, to the integrands of integrals over the interiors of cells.

The final version of the example form, after this is done, is shown in Figure 5.7.

The two subtrees from Figures 5.5 and 5.6 are evident, as is (near the top) the multiplication by the absolute value of the Jacobian determinant and a quadrature weight factor. The textual representation of the integrand in this figure was shown as Listing 1 in Chapter 1.

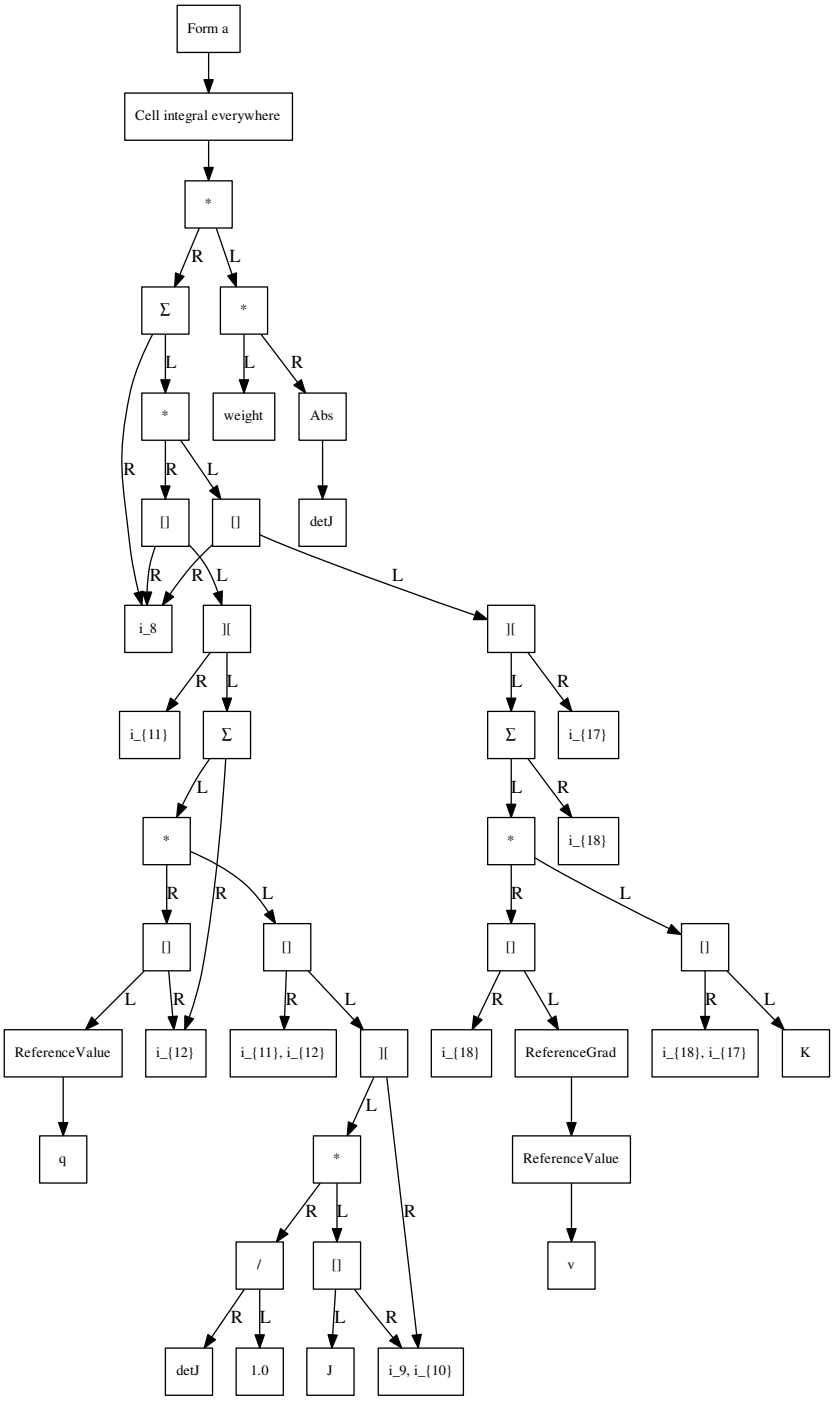

Figure 5.7: The final representatin of $\partial a_f(q, f; v)$.

5.3 Implementation of form transformations

In this section, we discuss the implementation of the form transformations described above, modifications to which will be described in Chapter 6.

The algebra lowering, derivative application and pull-back form transformations all use the same structure, the core of which is the the function `map_expr_dags` and the class `MultiFunction`. Here DAG means "directed acyclic graph"; the expression trees are DAGs.

Each subclass of `MultiFunction` represents a transformation on UFL expressions. Each of these subclasses has methods, called handlers, named after UFL expression types; the superclass provides an exception-generating default handler.

If the expression type has no operands (e.g. an argument, a coefficient, or a fixed numerical value), then the method has two arguments: the multi-function itself, and the expression. The return value is the transformed expression. For example, the application of the gradient to a coefficient is given by the following method in the `GradRuleset` subclass of `MultiFunction`:

```
def coefficient(self, o):
    if is_cellwise_constant(o):
        return self.independent_terminal(o)
    return Grad(o)
```

These handlers are typically uninteresting; in this one, the gradient of the coefficient is left unchanged, unless the coefficient is constant, in which case the gradient is zero.

If the expression type has operands, then the method has one extra argument for each operand, which represents the transformed operand. For example, the handler for sums in `GenericDerivativeRuleset`, which provides default handlers for the application of all derivative types, is:

```
def sum(self, o, da, db):
    return da + db
```

This expresses that the derivative of a sum is always the sum of the derivatives.

A multi-function method for an expression type with operands can have just two arguments, in which case the expression type is a *cut-off type* for that transformation.

The transformation implied by a multi-function is applied to a list of UFL expressions by calling the function `map_expr_dags`. The arguments to this call are an instance of the multi-function and the list of expressions. For each expression, a list is created of the nodes of that expression in post-order, so

that any operator appears after its operands. This avoids recursion. Then, for each node in the list that is not of a cut-off type, the appropriate handler is called, and the transformed result is stored. At the start of the list, near the leaves of the expression tree, the nodes typically have no operands; later in the list, when processing nodes that have operands, the transformed operands are passed to the handler as necessary. The transformed version of the last element of the list is the transformed version of the original expression.

An expression type should be marked as a cut-off type for a particular transformation if it is impossible to express the transformed expression as a function of the transformations of its operands. Before this project, cut-off types were not particularly noticeable in the transformations that we consider, but they are important for this work. For example, the divergence of a dot product cannot be expressed as a function of the divergences of the operands, and so the dot product will be a cut-off type for the application of divergences.

An optional third argument to `map_expr_dags` enables (by default) or disables a compression algorithm, in which the transformed results are inserted into and extracted from a dictionary in such a way that, upon returning, there is no more than one representation in memory of any sub-expression of any of the expressions.

The function `map_expr_dags` is wrapped by a couple of other functions and a lambda to eventually produce `map_integrand_dags`, which takes a multi-function and a form, and applies the multi-function transformation to the integrands of each of the integrals constituting the form.

A typical transformation, then, consists of a subclass of `MultiFunction` for which methods are overridden for all the relevant expression classes, and a function that wraps the creation of this multi-function and a call to `map_integrand_dags`.

Algebra lowering follows this pattern in a very straightforward way, leaving most nodes unchanged, but rewriting nodes involving vector operators.

The application of function pull-backs needs to transform only argument and coefficient nodes, so the associated multi-function has just a single non-trivial method, named for the common parent of arguments and coefficients.

The application of derivatives also follows the pattern, but in a more complicated fashion, because it must deal with various types of derivatives. The traversal of the tree proceeds as follows: `apply_derivatives` creates a `DerivativeRuleDispatcher` which traverses the tree in post-order, *not transforming nodes at all* until it encounters a node that represents a derivative: a gradient, a divergence, a curl, a reference gradient, a variable derivative (which we do not deal with at all) or a functional derivative. Once it encounters such a node, it instantiates another subclass of `MultiFunction`

to deal with this type of derivative (`GradRuleset`, say), and passes this and the derivative node to `map_expr_dags`. This call then starts a nested traversal of the associated subtree, performing the appropriate transformations. Once the nested traversal has reach the original derivative node, the original traversal continues.

Integral scaling does not follow this pattern at all, as it needs only to add a factor to each integrand.

5.4 Operations in computing form data

The function `compute_form_data` weaves together algebra lowering, derivative application, function pull-backs, integral scaling and much else besides. Listing 3 shows the structure of these operations from the start of the function call until the last call to any function affected by this work.

This function's primary argument is a form, but it also has many optional arguments (the `do...` flags) allowing one to specify precisely which operations are performed on that form. When `compute_form_data` is called by TSFC, Firedrake's form compiler, all of these flags are set to true by default, and the geometry types to be preserved are cell volumes and facet areas.

We are able to ignore the calls to functions not affected by this work, and may also ignore the last call to `apply_derivatives` (if it had any effect for us, it would not be dependent on a otherwise unimportant flag). Thus the important control path to consider is the following:

```
form = apply_algebra_lowering(form)
form = apply_derivatives(form)
form = apply_function_pullbacks(form)
form = apply_integral_scaling(form)
form = apply_derivatives(form)
```

The path consisting of lines 1 and 2 and the path consisting of lines 1, 2, 3 and 5 are also possible.

It is helpful for us to consider the effects of these operations on forms. These effects are shown in Table 5.1 for the largest set of operations. For each of three types of nodes or subtrees (vector operators, unapplied derivatives, and reference values) the table indicates whether or not they are present in the form at the start of the call to `compute_form_data` ("Original form"), after the call to `apply_algebra_lowering`, and so on. Note that applying function pull-backs leads to terms such as `Grad(ReferenceValue(f))`, which we consider to be unapplied derivatives.

```
form = apply_algebra_lowering(form)
form = apply_derivatives(form)
form = group_form_integrals(form, self.original_form.ufl_domains())
if do_estimate_degrees:
    form = attach_estimated_degrees(form)
if do_apply_function_pullbacks:
    form = apply_function_pullbacks(form)
if do_apply_integral_scaling:
    form = apply_integral_scaling(form)
if do_apply_default_restrictions:
    form = apply_default_restrictions(form)
if do_apply_geometry_lowering:
    form = apply_geometry_lowering(form, preserve_geometry_types)
if do_apply_function_pullbacks or do_apply_geometry_lowering:
    form = apply_derivatives(form)
    # Neverending story: apply_derivatives introduces new Jinvs,
    # which needs more geometry lowering
    if do_apply_geometry_lowering:
        form = apply_geometry_lowering(form, preserve_geometry_types)
        # Lower derivatives that may have appeared
        form = apply_derivatives(form)
```

Listing 3: The control flow of `compute_form_data` as it stood before the start of this project.

Table 5.1: The effects of the successive operations of the original `compute_form_data`. VO, UD and RV stand for "vector operators", "unapplied derivatives" and "reference values" respectively.

	VO	UD	RV
Original form	✓	✓	✗
Apply algebra lowering	✗	✓	✗
Apply derivatives	✗	✗	✗
Apply function pull-backs	✗	✓	✓
Apply integral scaling	✗	✓	✓
Apply derivatives	✗	✗	✓

Chapter 6

Structure preservation and Jacobian cancellation

This chapter discusses the changes made to Unified Form Language during the course of this project. These changes enable Jacobian cancellation in UFL.

6.1 Choice of vector operators

The objective of this project was to preserve certain vector- or tensor-related operations through the application of derivatives and function pull-backs, and to apply the Jacobian cancellations that this preservation allows.

It is not obvious which operators should be preserved. Clearly, given the pull-back formulae (4.15) and (4.28), the divergence and curl operators must be preserved. The gradient operator was already preserved. It is clear that the dot product must be preserved, as this is necessary to allow Jacobian cancellation. To this list, we add the inner product, which is sometimes used as a synonym for the vector dot product, and is in general highly tractable.

It is tempting to add also the nabla-gradient and nabla-divergence operators, as the appropriate manipulations are obvious given those for the gradient and divergence operators. However, this leads to significant near-repetition of the code for the non-nabla operators. Additionally, in the usual cases (scalar and vector arguments, respectively), the nabla-gradient and nabla-divergence are identical to the gradient and divergence, and are reduced to them before the application of derivatives and function pull-backs. Finally, it appears from the Firedrake tests that the nabla operators are used much more seldom than the usual operators. In conclusion, it appears that preserving the nabla operators offers too little advantage to justify developing

and maintaining the appropriate code, and we do not preserve them.

Thus, this project adds the preservation of the following operators through the application of derivatives and function pull-backs: the dot and inner products, and the divergence and curl operators. In the sequel we loosely refer to these and the gradient (which is affected by this work) as the "vector operators" for brevity, or the "tractable vector operators" if it necessary to distinguish them from the vector operators that are not preseved, such as the cross product.

6.2 Algebra lowering

In order to allow the application of derivatives and function pull-backs while vector operators are still present in expressions, algebra lowering is split into two parts, one before these operations and one after. The tractable vector operators are removed in the second round of algebra lowering, after the application of derivatives and pull-backs. The intractable vector operators (matrix inverses, cross products and so on) are removed immediately.

The implementation consisted of splitting the `LowerCompoundAlgebra` multi-function into `LowerIntractableCompoundAlgebra`, which deals with the intractable operators of the first round, and `LowerAllCompoundAlgebra`, which inherits from `LowerIntractableCompoundAlgebra` and deals additionally with the preserved vector operators. These two multi-functions are then packaged into the functions `apply_minimal_algebra_lowering` and `apply_algebra_lowering`.

6.3 Applying derivatives

The second step in processing a form is to apply the derivatives present in that form, which for our purposes are the functional derivatives and the spatial derivatives grad, div and curl.

6.3.1 Functional derivatives

Dot and inner products

The implementations of the application of functional derivatives to dot products and inner products are direct representations of the rules (3.32) and (3.33), for example the following method of `GateauxDerivativeRuleset`:

```
def dot(self, o, fp, gp):
    f, g = o.ufl_operands
```

```
    return Dot(fp, g) + Dot(f, gp)
```

In each case, the arguments passed to the handler are the rule-set itself, the original product node, and the functional derivatives of the two arguments; in each case, the result is the sum of the two appropriate products.

Gradient, divergence and curl

Prior to this project, the application of a functional derivative to a gradient was already handled, in a complicated fashion. This is now replaced by the following direct implementation of (3.19):

```
def grad(self, o, op):
    if is_cellwise_constant(op):
        return self.independent_operator(o)
    return apply_derivatives(Grad(op))
```

The functional derivative of the gradient of a function is transformed to the gradient of the functional derivative of the function.

There are, however, two additional features. First, there is a check for the case where the application of the functional derivative to the argument of the gradient has resulted in a zero function, in which case the gradient is also zero (`independent_operator` returns a zero of the appropriate size).

Second, the new gradient is applied using `apply_derivatives`. In most cases, the new argument to the gradient will simply be an `Argument`, and so this application will have no effect. There are, however, at least two cases when the application is necessary in order for `apply_derivatives` to complete successfully:

1. It is possible in UFL to take a functional derivative with respect to a component of a vector coefficient, like this:

   ```
   derivative(grad(u), u[0], w)
   ```

   where `u` is a `Coefficient` on a vector finite element and `w` is a scalar `Argument`. In the body of `derivative`, this is converted to the functional derivative of `grad(u)` with respect to `u` in the direction of a `ListTensor` whose first element is `w` and whose other elements are zeros. Thus the application of the functional derivative will result in the gradient of this `ListTensor`. This gradient must be passed through the `ListTensor` by a nested call to `apply_derivatives` in order for the final result to have all its derivatives fully applied.

2. When taking a functional derivative, it is possible to explicitly specify relationships between apparently independent coefficients in a form by

specifying the `coefficient_derivatives` argument to `derivative`. In this case, the application of the functional derivative to the gradient of a coefficient can result in the gradient of the product of an argument and another coefficient. Specifically, suppose that we have coefficients `f` and `h`, and that we take the functional derivative of `grad(f)` with respect to `h` in the direction `w`, specifying that the derivative of `f` with respect to `h` is `df`. Then, applying derivatives, the functional derivative of `f` is `w*df`, and so the functional derivative of `grad(f)` is `grad(w*df)`. The gradient must be applied to this product by a nested call to `apply_derivatives`.

There may also be other cases in which the gradient must be applied, so we simplify our code by always calling `apply_derivatives` after wrapping the result of a functional differentation in a gradient. This unified handling risks inefficiency, but relieves us of the burden of identifying cases in which the gradient must be applied, which may be complicated by the capability to take function derivatives with respect to tuples of coefficients (a possibility that is addressed directly in the previous implementation, but which is naturally handled correctly here).

The implementations of the applications of functional derivatives to divergences and curls correspond directly to that for the gradient, using (3.20) and (3.21). Again, nested calls to `apply_derivatives` are used.

6.3.2 Spatial derivatives: Gradient

The application of the gradient to most existing expression types was already handled before the start of this project, so implementations are only needed for the vector operators. Additionally, the application of the gradient to a `ReferenceValue` is changed, but as this forms part of the pull-back process it is discussed in Section 6.4.3.

Dot and inner products

In the usual case, where the arguments are vectors, the gradient of a dot product is given by (2.25); in the general case, it is given by (2.22). The implementation of these rules is shown in Listing 4 to give an example of the associated index manipulations.

The gradient of an inner product is given by (2.43) if the arguments are both vectors (a case identical to the dot product), or (2.42) in general. The implementation is very similar to that for the dot product.

```
def dot(self, o, grad_f, grad_g):
    f, g = o.ufl_operands
    if len(f.ufl_shape) == 1 and len(g.ufl_shape) == 1:
        return Dot(g, grad_f) + Dot(f, grad_g)
    else:
        fi = indices(len(f.ufl_shape)-1)
        gi = indices(len(g.ufl_shape)-1)
        grad_index = indices(1)
        sum_index = indices(1)
        term1 = (grad_f[fi + sum_index + grad_index]
                 * g[sum_index + gi])
        term2 = (f[fi + sum_index]
                 * grad_g[sum_index + gi + grad_index])
        return as_tensor(term1 + term2,
                         fi + gi + grad_index)
```

Listing 4: The application of the gradient to a dot product.

Gradient, divergence and curl

The previous implementation of the application of a gradient to a gradient is retained. It is notable that this implementation makes the inner gradient a cut-off node, taking into account that the inner gradient will always have been applied so that its operand is an argument, a coefficient or a third gradient.

The gradients of divergences and curls are not implemented; they are not used in Firedrake's tests.

6.3.3 Spatial derivatives: Divergence

The rules for applying the divergence operator are all new. Fortunately, in most ways they follow the pattern of the handling of gradients; this section highlights the non-trivial implementations. The handling of the divergence of a `ReferenceValue` is discussed in Section 6.4.3 because it forms part of the pull-back process.

Coefficients and arguments

These are cut-off nodes; their divergences are left as such (except in the case of a cellwise constant coefficient, when the divergence is reduced to zero).

Tensors and indexing

The handling of `ListTensor`, `ComponentTensor` and `Indexed` nodes, which handle the joining of scalars into tensors and the splitting of tensors into scalars, has some complexity, the upshot of which is that only in one case (a `ListTensor` of non-scalar quantities) can the divergence be passed through; in the other cases, the divergence is lowered to a sum of gradient components and applied in that form.

Gradient, divergence and curl

The divergence of the gradient of a function is its Laplacian. The divergence operator cannot pass through the gradient operator, though, so gradients are cut-off nodes.

The divergence of a divergence is not supported.

The divergence of the curl of any function is zero, which is reflected in the implementation.

Dot and inner products

The divergence of a dot product is handled in three cases, according to (2.33), (2.38) and (2.35).

Since the result of an inner product is always a scalar, and the divergence of a scalar is not defined, there is no need to handle this case.

6.3.4 Spatial derivatives: Curl

The majority of quantities are handled in the same way for curl as they are for the divergence, with the obvious changes. The fact that the curl of the gradient of any function is zero is reflected in the implementation. The curl of a `ReferenceValue` is discussed in Section 6.4.3.

6.4 Applying pull-backs

The process of applying pull-backs consists of two steps: first, the pull-backs are applied to the physical-space functions occurring in an expression, giving reference-space functions; and then, through a call to `apply_derivatives`, physical-space gradients, divergences and curls of these reference-space functions are converted to reference-space gradients, divergences and curls. These two steps are described in the following two subsections.

6.4.1 Function pull-backs

The function `apply_function_pullbacks` already implemented the pull-backs of coefficients and arguments on finite elements that use the covariant and contravariant Piola mappings (4.13) and (4.26). A result of the latter implementation was demonstrated in Section 5.2.3. These implementations, though, were based on scalar operations rather than vector operations – a form of preemptive algebra lowering. We replace the original implementations with ones based on the dot product.

In these implementations, the input function is `g`, the value to be returned is `f` and we have the following definitions:

```
r = ReferenceValue(g)
domain = g.ufl_domain()
J = Jacobian(domain)
detJ = JacobianDeterminant(domain)
Jinv = JacobianInverse(domain)
transform_hdiv = (1.0/detJ) * J
```

Each mapping is in fact handled twice: once for the case where the entire finite element uses this mapping, and once for the case where it is used only on part of the finite element, as may be the case on a mixed element. We discuss only the first of these; the implementation for the other is clear.

We note here that the Jacobian cancellations described later do not work if the coefficients and arguments are taken directly from a mixed element. This is discussed further under the suggestions for further work, in Section 8.2.2.

For the covariant Piola mapping, the change is simple: we replace

```
f = as_vector(Jinv[j, i]*r[j], i)
```

by

```
f = Dot(r, Jinv)
```

For the contravariant Piola mapping, this is made slightly more difficult by the factor $\frac{1}{|\det DF|}$. The `Product` expression type only allows its arguments to be scalars, so we cannot directly express $\frac{1}{|\det DF|} DF\, v$. More precisely, we can write

```
1.0/detJ * Dot(J, r)
```

but this will immediately be reduced to

```
ComponentTensor(
  Product(Indexed(Dot(J, r), MultiIndex(i,))),
          Division(FloatValue(1.0), detJ)),
  MultiIndex((i,)))
```

by the implementation of the `*` operator. From this expression, it would be difficult to recognise any possible cancellation of `J` with a `Jinv` appearing elsewhere in the form.

Accordingly, we add a new expression type `ScalarTensorProduct` to UFL. It appears that this type should be used in the implementation of `*`, so that `1.0/detJ * Dot(J, r)` is automatically converted to

```
ScalarTensorProduct(
  Division(FloatValue(1.0), detJ)),
  Dot(J, r))
```

However, this turns out to be impossible: if it were done, then scalar-tensor products would propagate to other parts of Firedrake that do not know how to handle them, because `*` is occasionally used in scalar-tensor situations where no further algebra lowering occurs. Instead, we only introduce scalar-tensor products when performing function pull-backs. They can be introduced manually by a user, but will never be produced by `*`. (Unfortunately, this means that the use of `*` in an expression within which there is Jacobian cancellation can stop the recognition of this cancellation by the algorithm described in Section 6.5, as the appropriate structure may be lost.)

Specifically, for the contravariant Piola mapping we replace

```
f = as_vector(transform_hdiv[i, j]*r[j], i)
```

with

```
f = ScalarTensorProduct(1.0/detJ, Dot(J, r))
```

6.4.2 Special cases

There are two special cases to be considered. The first is the divergence of a function that uses the contravariant Piola mapping, where we have

$$\operatorname{div} v = \frac{1}{\det DF} \widehat{\operatorname{div}}\, \hat{v}. \qquad \text{(repeat of 4.15)}$$

The second is the curl of an $\mathbb{R}^3$-valued function that uses the covariant Piola mapping, for which the relevant result is

$$\operatorname{curl} v = \frac{1}{|\det DF|} DF\, \widehat{\operatorname{curl}}\, \hat{v}. \qquad \text{(repeat of 4.28)}$$

It appears that these cases could be implemented as part of the application of derivatives, as are the other conversions from physical-space to reference-space derivatives. In particular, one could add these cases to the

handling of reference values in the multi-functions for applying divergences and curls. However, by the point at which the associated divergence or curl would be applied, its argument would no longer be a reference value: instead it would be the product of the reference value with $\frac{1}{\det DF} DF$ or DF^{-T}, as a result of the transformations of the previous section. Thus these special cases must be handled in the same algorithmic step as the initial function pull-backs.

Special care is taken in the implementations to ensure that reference-space divergences and curls are *not* introduced: instead, the appropriate combinations of the components of the reference-space gradient are used. This has no apparent disadvantages, since the reference-space divergence and curl do not allow any helpful simplifications during later transformations; and it has the advantage that these two types do not need to be supported in derivative application, Jacobian cancellation or algebra lowering, as they are not used elsewhere in UFL.

6.4.3 Spatial derivatives of pulled-back functions

Following function pull-backs, forms may contain physical-space derivatives of reference-space functions. A call to `apply_derivatives` replaces these physical-space derivatives with reference-space derivatives.

Gradient

The physical-space gradient of a reference-space function was already implemented before this project began. As with some of the pull-backs described in the previous section, the work of this project was only to convert the pre-emptively lowered expressions into their more natural forms, replacing the implementation of Listing 5 with the one of Listing 6.

Divergence

The divergence of a reference-space function was not handled in UFL before this project was begun, as pull-backs were performed after algebra lowering. The case of the contravariant Piola mapping is handled directly, as described above, so here we need only deal with the general case. However, in the general case there is no need to treat the divergence specially, and so it can be implemented in terms of the rule for gradients by being lowered to an indexed sum, with a nested call to `apply_derivatives` to ensure that the gradient is transformed appropriately. The implementation added to `DivRuleset` is shown in Listing 7.

```
def reference_value(self, o):
    # grad(o) == grad(rv(f)) -> K_ji*rgrad(rv(f))_rj
    f = o.ufl_operands[0]
    if not f._ufl_is_terminal_:
        error("ReferenceValue can only wrap a terminal")
    domain = f.ufl_domain()
    K = JacobianInverse(domain)
    r = indices(len(o.ufl_shape))
    i, j = indices(2)
    Do = as_tensor(K[j, i]*ReferenceGrad(o)[r + (j,)], r + (i,))
    return Do
```

Listing 5: The application of the gradient to a reference value prior to this work.

```
def reference_value(self, o):
    f, = o.ufl_operands
    if not f._ufl_is_terminal_:
        error("ReferenceValue can only wrap a terminal")
    K = JacobianInverse(f.ufl_domain())
    return Dot(ReferenceGrad(o), K)
```

Listing 6: The application of the gradient to a reference value subsequent to this work.

It is not immediately clear that it is worth applying the divergence in this way rather than allowing it to remain and waiting for later algebra lowering and derivative application to achieve precisely the same transformation. However, for the clarity and brevity of `compute_form_data`, it is helpful to impose the following post-condition on the call to `apply_derivatives` currently under discussion: *following this call, no physical-space derivative operators remain in the expression.* In fact, because the special cases of the previous section do not introduce reference-space divergences or curls, this can be strengthened: *after application of derivatives, the only derivative operators remaining in the expression are reference-space gradients.* This constraint is used in Section 6.7.

Curl

The curl of a reference-space function was not previously handled in UFL. The special case of the covariant Piola mapping is covered separately, as described in Section 6.4.2, and so, as with the divergence, the implementation

```
def reference_value(self, o):
    f, = o.ufl_operands
    if not f._ufl_is_terminal_:
        error("ReferenceValue can only wrap a terminal")
    return apply_derivatives(self.div_ito_grad(o))
```

Listing 7: The application of the divergence to a reference value.

```
def reference_value(self, o):
    f, = o.ufl_operands
    if not f._ufl_is_terminal_:
        error("ReferenceValue can only wrap a terminal")
    return apply_derivatives(self.curl_ito_grad(o))
```

Listing 8: The application of the curl to a reference value.

in `CurlRuleset` needs only to handle the general case, in which there is no harm in lowering curl to components of the gradient. The implementation is shown in Listing 8.

6.5 Jacobian cancellation

The aim of this project was to recognise cases in which the Jacobian and its inverse both emerge during pull-back, and to ensure that they are cancelled if possible. In some cases, this cancellation is implicit. For example, we have that for $v = \mathcal{F}^{\mathrm{div}}(\hat{v})$,

$$\operatorname{div} v = \frac{1}{\det DF} \widehat{\operatorname{div}}\, \hat{v}, \qquad \text{(repeat of 4.15)}$$

which is a result, roughly speaking, of the cancellation of the Jacobian emerging from converting v to $\hat{v}$ with the inverse Jacobian emerging from converting the physical-space divergence to the reference-space divergence. This rule is directly implemented in UFL, and so the cancellation is implicit. The original intention of this work was to implement only such implicit cancellations.

However, consider the corresponding rule for the curl of $v = \mathcal{F}^{\mathrm{curl}}(\hat{v})$:

$$\operatorname{curl} v = \frac{1}{|\det DF|} DF\, \widehat{\operatorname{curl}}\, \hat{v}. \qquad \text{(repeat of 4.28)}$$

There is no similar cancellation here.

Instead, we can see cancellation in the following cases: for $v = \mathcal{F}^{\mathrm{id}}(\hat{v})$, $q = \mathcal{F}^{\mathrm{div}}(\hat{q})$, and $u = \mathcal{F}^{\mathrm{curl}}(\hat{u})$ we have

$$\int_K q \cdot \operatorname{grad} v \,\mathrm{d}x = \pm \int_{\hat{K}} \hat{q} \cdot \widehat{\operatorname{grad}\, \hat{v}} \,\mathrm{d}\hat{x} \qquad \text{(repeat of 4.29)}$$

and

$$\int_K u \cdot q \,\mathrm{d}x = \pm \int_{\hat{K}} \hat{u} \cdot \hat{q} \,\mathrm{d}\hat{x}. \qquad \text{(repeat of 4.30)}$$

In each of these cases, the Jacobian cancellation emerges from the dot product of two functions or their derivatives, rather than from a single derivative of a single function.

Directly recognising these two cases in UFL expressions appears possible. It would also be reasonable to recognise the same expressions with the arguments to the dot product reversed. However, this would lead to a situation where one would have to maintain a list of all the situations in which helpful Jacobian cancellation occurs. This list would have to recognise that the sum of two functions with a particular mapping is equivalent for these purposes to a single such function; that multiplication of a dot product by a scalar is irrelevant, and so on and so forth. It appeared preferable instead to insert the Jacobians and recognise where they cancel.

In general, recognising cancellations in expressions is difficult. However, this is an extremely simple cancellation: it can only occur with a dot product, it is unlikely to occur more than once in a given expression, and we can enumerate the four ways in which it can occur. For A and B some tensors, and J and K the Jacobian and its inverse, we have

$$\begin{aligned}
\operatorname{dot}(\operatorname{dot}(A, J), \operatorname{dot}(K, B)) &= \operatorname{dot}(A, \operatorname{dot}(J, \operatorname{dot}(K, B))) && (6.1)\\
&= \operatorname{dot}(A, \operatorname{dot}(\operatorname{dot}(J, K), B)) && (6.2)\\
&= \operatorname{dot}(A, \operatorname{dot}(I, B)) && (6.3)\\
&= \operatorname{dot}(A, B) && (6.4)
\end{aligned}$$

using the easily-proved associativity of the dot product, and denoting by I the identity matrix of the same size as the Jacobian. Similarly,

$$\operatorname{dot}(\operatorname{dot}(A, K), \operatorname{dot}(J, B)) = \operatorname{dot}(A, B). \qquad (6.5)$$

If the two arguments to the dot product are vectors, then we can transpose them without changing the result, and so there are two more ways in which the cancellation can occur: for A and B vectors,

$$\operatorname{dot}(\operatorname{dot}(J, A), \operatorname{dot}(B, K)) = \operatorname{dot}\left((\operatorname{dot}(J, A))^T, (\operatorname{dot}(B, K))^T\right) \qquad (6.6)$$

$$= \operatorname{dot}(\operatorname{dot}(A, J), \operatorname{dot}(K, B)) \tag{6.7}$$
$$= \operatorname{dot}(A, B) \tag{6.8}$$

and similarly

$$\operatorname{dot}(\operatorname{dot}(K, A), \operatorname{dot}(B, J)) = \operatorname{dot}(A, B). \tag{6.9}$$

These are the only four cases in which Jacobian cancellation is known to occur.

It appears that there should be four more cases, based on the transposes of the Jacobian and its inverse, but in fact these are never introduced during function pull-back: the Piola transformations and the expression (4.12) for the gradient of an $\mathcal{F}^{\text{id}}$-mapped function involve the untransposed matrices, and the expression (4.28) for the curl of an $\mathcal{F}^{\text{curl}}$-mapped function, which does involve the transpose, is rewritten to use the Jacobian directly without loss of generality.

Jacobian cancellation, being a form transformation, is naturally implemented with `MultiFunction` and `map_expr_dags` as described in Section 5.3. To understand the necessary implementation, it is helpful to consider an extension to (4.29) and (4.30) that we hope the cancellation to be successful on: with $v = \mathcal{F}^{\text{id}}(\hat{v})$, $u = \mathcal{F}^{\text{id}}(\hat{u})$ and $q = \mathcal{F}^{\text{div}}(\hat{q})$,

$$q \cdot (\operatorname{grad} v + \operatorname{grad} u) \tag{6.10}$$
$$= q^T(\operatorname{grad} v + \operatorname{grad} u) \tag{6.11}$$
$$= \left[\frac{1}{\det DF} DF\hat{q}\right]^T \left[(DF)^{-T}\widehat{\operatorname{grad}}\,\hat{v} + (DF)^{-T}\widehat{\operatorname{grad}}\,\hat{u}\right] \tag{6.12}$$
$$= \frac{1}{\det DF}\hat{q}^T(DF)^T(DF)^{-T}\left(\widehat{\operatorname{grad}}\,\hat{v} + \widehat{\operatorname{grad}}\,\hat{u}\right) \tag{6.13}$$
$$= \frac{1}{\det DF}\hat{q}^T\left(\widehat{\operatorname{grad}}\,\hat{v} + \widehat{\operatorname{grad}}\,\hat{u}\right) \tag{6.14}$$
$$= \frac{1}{\det DF}\hat{q} \cdot \left(\widehat{\operatorname{grad}}\,\hat{v} + \widehat{\operatorname{grad}}\,\hat{u}\right). \tag{6.15}$$

Expressed in terms of UFL's dot product, we have

$$\operatorname{dot}(q, \operatorname{grad} v + \operatorname{grad} u) = \frac{1}{\det DF}\operatorname{dot}\left(\hat{q}, \widehat{\operatorname{grad}}\,\hat{v} + \widehat{\operatorname{grad}}\,\hat{u}\right). \tag{6.16}$$

For Jacobian cancellation to occur in this expression, at the point at which the dot product node is processed, the algorithm must be aware that the right-hand operand (the sum) contains an inverse Jacobian, so the tree must be traversed in post-order, child before parent. However, once the dot product node is processed and it is clear that the inverse Jacobian must be cancelled in the sum, one must start another algorithm processing the sum subtree in pre-order to cancel the Jacobian. This appears to result in multiple traversals of various sub-trees.

```
def jacobian(self, o):
    dim1, dim2 = o.ufl_shape
    if dim1 == dim2:
        return (o, Identity(dim1), Identity(dim1), None, None)
    else:
        return (o, None, None, None, None)

def jacobian_inverse(self, o):
    dim1, dim2 = o.ufl_shape
    if dim1 == dim2:
        return (o, None, None, Identity(dim1), Identity(dim1))
    else:
        return (o, None, None, None, None)

def terminal(self, o):
    return (o, None, None, None, None)
```

Listing 9: The Jacobian cancellation methods for Jacobians, their inverses, and terminals (arguments, coefficients and similar).

In the implementation given as part of this project, we complete the cancellation in a single post-order traversal of the tree. This is achieved by, at each node, returning not just the node with any Jacobian cancellation completed, but a five-tuple with the following elements:

- The node, with any Jacobian cancellation completed (call it N).
- N with any Jacobian on the left removed; if there is no Jacobian on the left, then `None`. Alternatively phrased, this element of the tuple contains N multiplied by the inverse Jacobian on the left, provided that this results in some cancellation, or otherwise `None`.
- N with any Jacobian on the right removed, or `None`.
- N with any inverse Jacobian on the left removed, or `None`.
- N with any inverse Jacobian on the right removed, or `None`.

This implemented as the `JacobianCancellation` multi-function. The handlers for several leaf nodes are shown in Listing 9. (Note that the implementation allows for non-square Jacobians, but does not support their cancellation.)

These tuples provide the handler for dot products precisely enough information to check for the cancellations (6.1) to (6.9). For example, for (6.1), the handler checks if the tuple from the left-hand operand has a non-`None`

```
def dot(self, o, left_tuple, right_tuple):
    # _sjl means "sans Jacobian on the left", etc.
    left, left_sjl, left_sjr, left_skl, left_skr = left_tuple
    right, right_sjl, right_sjr, right_skl, right_skr = right_tuple
    transpose_allowed = o.ufl_shape == ()
    if left_sjr and right_skl:
        return (Dot(left_sjr, right_skl),
                None, None, None, None)
    elif left_skr and right_sjl:
        return (Dot(left_skr, right_sjl),
                None, None, None, None)
    elif transpose_allowed and left_sjl and right_skr:
        return (Dot(left_sjl, right_skr),
                None, None, None, None)
    elif transpose_allowed and left_skl and right_sjr:
        return (Dot(left_skl, right_sjr),
                None, None, None, None)
    else:
        return (Dot(left, right),
                Dot(left_sjl, right) if left_sjl else None,
                Dot(left, right_sjr) if right_sjr else None,
                Dot(left_skl, right) if left_skl else None,
                Dot(left, right_skr) if right_skr else None)
```

Listing 10: The core of the implementation of Jacobian cancellation.

third entry (a version with the Jacobian cancelled on the right) and the tuple from the right-hand operand has a non-`None` fourth entry (a version with the inverse Jacobian cancelled on the left); if so, then the result from the dot product node is the dot product of those two versions, along with four `None`s. The handler for dot products, which is the core of the implementation, is shown in Listing 10.

An extra level of indirection is required in wrapping this multi-function because it returns tuples instead of expressions; it is wrapped first into a function that cancels Jacobians in expressions, using `map_expr_dag` and discarding, on completion, the four extra elements of the final tuple. Then this function is used with `map_integrands` to cancel Jacobians in each integrand of a form. This does not impede the compression algorithm usually used in traversing expression trees. The top-level function is called `apply_jacobian_cancellation`.

6.6 Jacobian determinant cancellation

Following full algebra lowering, integral scaling may be applied: each integrand may be multiplied by the absolute value of the determinant of the Jacobian, $|\det DF|$.

Following this, an attempt can be made to cancel this factor with any factors $\frac{1}{|\det DF|}$ or $\frac{1}{\det DF}$ that already existed in the integrand. This algorithm, `apply_det_j_cancellation`, follows the same pattern as the Jacobian cancellation, except that it uses a triple instead of a 5-tuple, because there are only two ways in which cancellation can occur. The implementation is also particularly simple because many expression types behave similarly to each other: four types of products share an implementation, as do three indexing-related types.

6.7 Operations in computing form data

This section discusses the changes made to the structure of `compute_form_data` in the course of this project to support the operations described in the previous sections. The original structure of `compute_form_data` was shown in Listing 3.

The first and most obvious change is that the call to `apply_algebra_lowering` is replaced by a call to `apply_minimal_algebra_lowering`, and a call to `apply_algebra_lowering` is added after all the other transformations discussed here.

The second change is the addition of `apply_jacobian_cancellation`. Since this is only required if function pull-backs are applied, it is added to the block in which `apply_function_pullbacks` is called. However, an application of derivatives is necessary between the calls to `apply_function_pullbacks` and `apply_jacobian_cancellation`, so this is added too.

Finally, after the call to `apply_algebra_lowering`, another call to `apply_derivatives` is necessary, at least in the case that function pull-backs were not applied.

It turns out that these are the only changes required, but this fact is not as obvious as it seems. In particular, it depends upon our post-condition that the call to `apply_derivatives` following function pull-backs reduces all spatial derivative operators to reference gradients. If, for example, this call left physical-space divergences of identity-mapped functions unchanged, because there is no vector operator simplification, then another application of derivatives and two full algebra lowerings would be required to cover the case of such a divergence.

Table 6.1: The effects of the successive operations of the new `compute_form_data`. IVO, TVO, UD and RV stand for "intractable vector operators", "tractable vector operators", "unapplied derivatives" and "reference values" respectively.

	IVO	TVO	UD	RV
Original form	✓	✓	✓	✗
Apply minimal algebra lowering	✗	✓	✓	✗
Apply derivatives	✗	✓	✗	✗
Apply function pull-backs	✗	✓	✓	✓
Apply derivatives	✗	✓	✗	✓
Apply Jacobian cancellation	✗	✓	✗	✓
Apply integral scaling	✗	✓	✗	✓
Apply Jacobian determinant cancellation	✗	✓	✗	✓
Apply algebra lowering	✗	✗	✓	✓
Apply derivatives	✗	✗	✗	✓

The results of the sequence of operations are shown in Table 6.1.

If function pull-backs are not performed, then the full algebra lowering follows immediately on from the first call to `apply_derivatives`, again resulting in an expression that is free from all vector operators and from unapplied derivatives.

Chapter 7

Evaluation

In this chapter, the success of Jacobian cancellation is demonstrated on three example forms.

The first of these forms has already been discussed: Chapter 1 showed the results of the transformations before and after this work in textual form (Listings 1 and 2); and the effect of the transformations prior to this work were displayed graphically in Chapter 5 (Figures 5.1 to 5.7). Section 7.1 shows the effects of the new transformations graphically. Sections 7.2 and 7.3 will show the results on two other forms textually.

In each case, let the physical space K, reference space $\hat{K}$, and the mapping F between them be given.

7.1 First form

Let $q = \mathcal{F}^{\mathrm{div}}(\hat{q})$ and $f \in H^1(K)$. Define

$$a(q, f) = \int_K q \cdot \operatorname{grad} f \,\mathrm{d}x \tag{7.1}$$

and let $\partial a_f(q, f; v)$ be the functional derivative of a with respect to f in the direction of $v = \mathcal{F}^{\mathrm{id}}(\hat{v}) \in H^1(K)$.

It was already shown, in Section 3.6.1, that

$$\partial a_f(q, f; v) = \int_K q \cdot \operatorname{grad} v \,\mathrm{d}x. \tag{7.2}$$

Thus the rule (4.29) is directly applicable, and we have

$$\partial a_f(q, f; v) = \pm \int_{\hat{K}} \hat{q} \cdot \widehat{\operatorname{grad}}\, \hat{v} \,\mathrm{d}\hat{x} \tag{7.3}$$

where $\pm$ denotes the sign of the determinant of the Jacobian on $\hat{K}$.

This form illustrates:

- Structure-preserving functional derivatives of the gradient operator and the dot product.
- Jacobian cancellation in the dot product of an $\mathcal{F}^{\text{div}}$-mapped function and the gradient of an $\mathcal{F}^{\text{id}}$-mapped function, according to (4.29).

The representation of the form after minimal algebra lowering is shown in Figure 7.1. This is identical to Figure 5.2, as minimal algebra lowering has no effect on this form: the dot product is preserved.

Following the application of derivatives, then, one obtains Figure 7.2.

The representation of this form following pull-backs and the subsequent application of derivatives is shown in Figure 7.3. The subtrees representing the pulled-back versions of `q` and `v` use the dot product and the scalar-tensor product, and so are much simpler than the corresponding quantities before (Figures 5.5 and 5.6). The potential for Jacobian cancellation is clear.

The representation following Jacobian cancellation is shown in Figure 7.4, and clearly illustrates that the cancellation has occurred.

The representation following integral scaling and Jacobian determinant cancellation is shown in Figure 7.5. This form still depends upon the determinant of the Jacobian, but only through its sign. The final form, after full algbra lowering, is not shown.

The fully processed forms, before and after completion of this project, were shown in Listings 1 and 2 respectively.

These listings and those in the following sections are the representations of the integrands of these forms, with `Coefficients` and `Arguments` replaced by their names; for example, `v` stands for

```
Argument(
  FunctionSpace(
    Mesh(
      VectorElement(
        FiniteElement('Lagrange', triangle, 1),
        dim=2),
    -1),
    FiniteElement('Lagrange', triangle, 1)),
  0,
  None)
```

Even in the case where the Jacobian cancellation has occurred, the resulting expression is fairly long and difficult to read, as a result of the full algebra lowering. However, terms related to the Jacobian have been highlighted in red, so it should be clear when cancellation has occurred.

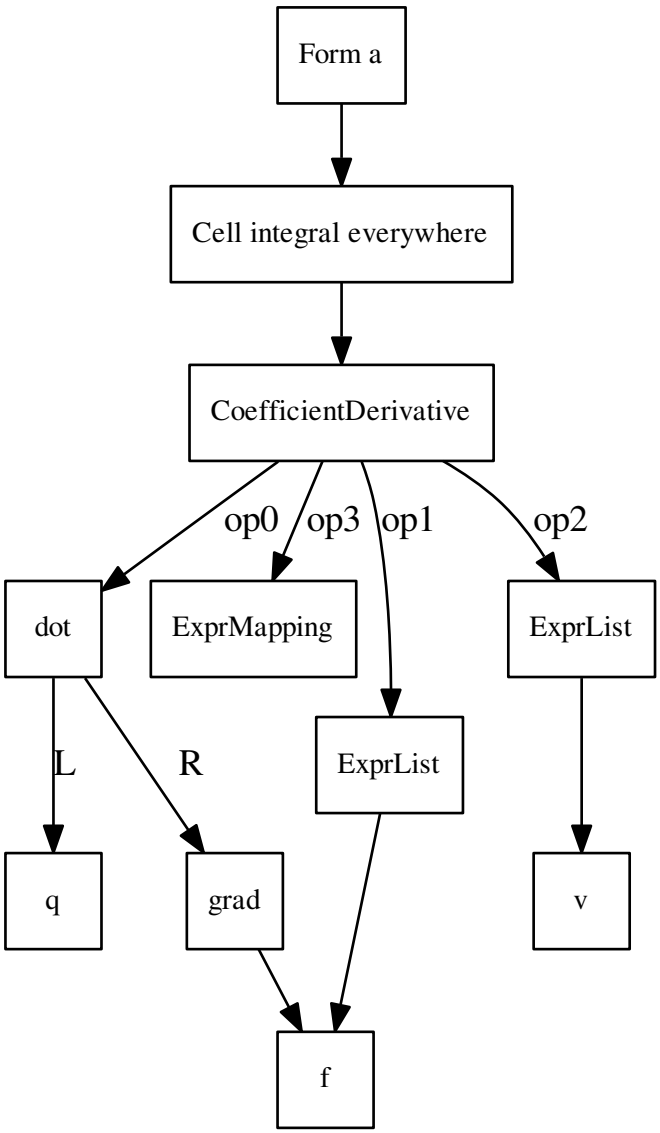

Figure 7.1: The first evaluation form after minimal algebra lowering.

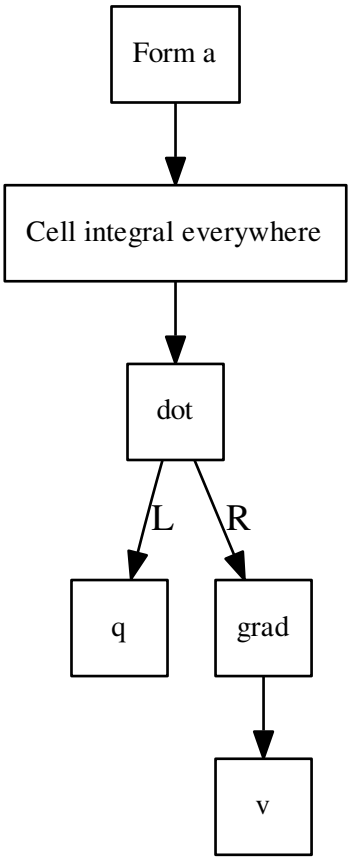

Figure 7.2: The first evaluation form after the application of derivatives.

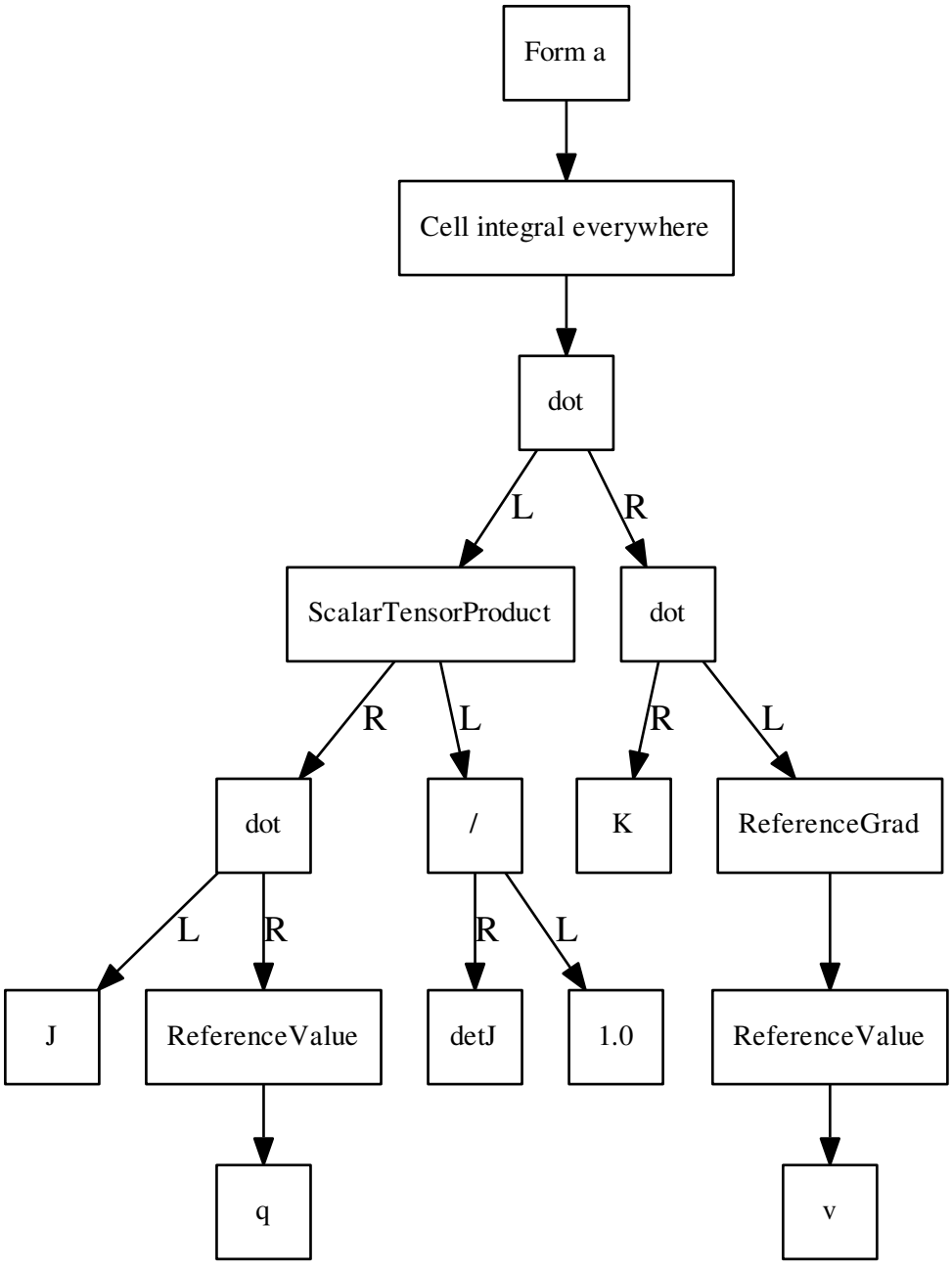

Figure 7.3: The first evaluation form after function pull-backs and the associated application of derivatives.

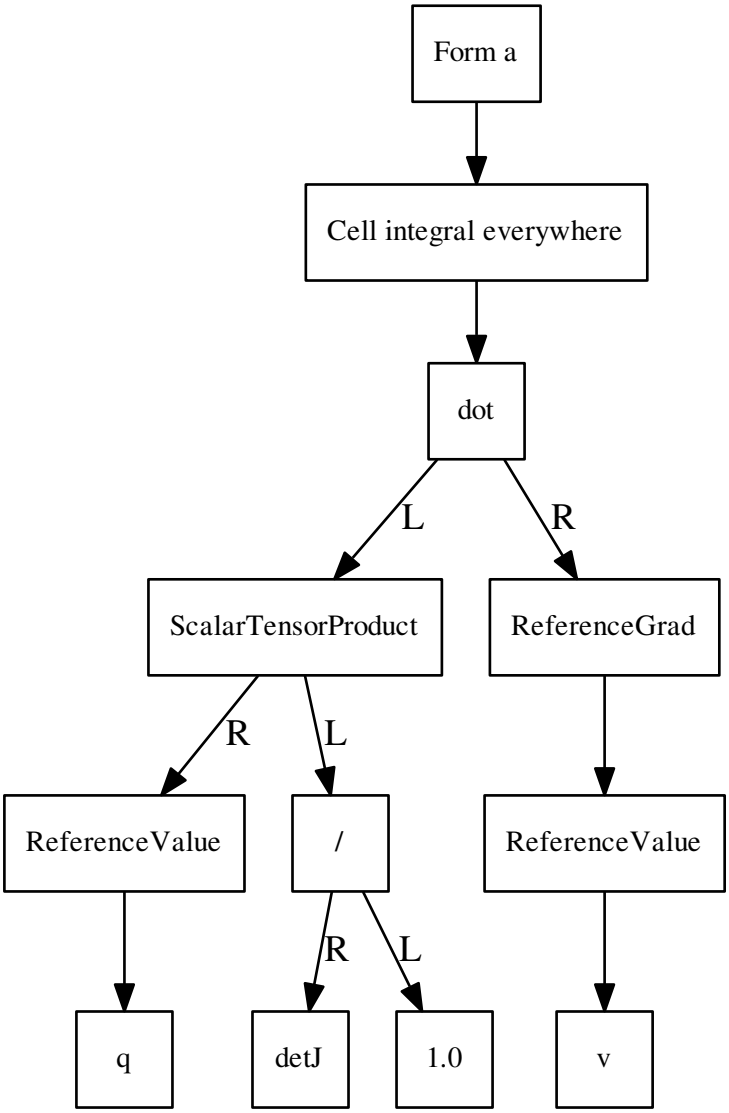

Figure 7.4: The first evaluation form after Jacobian cancellation.

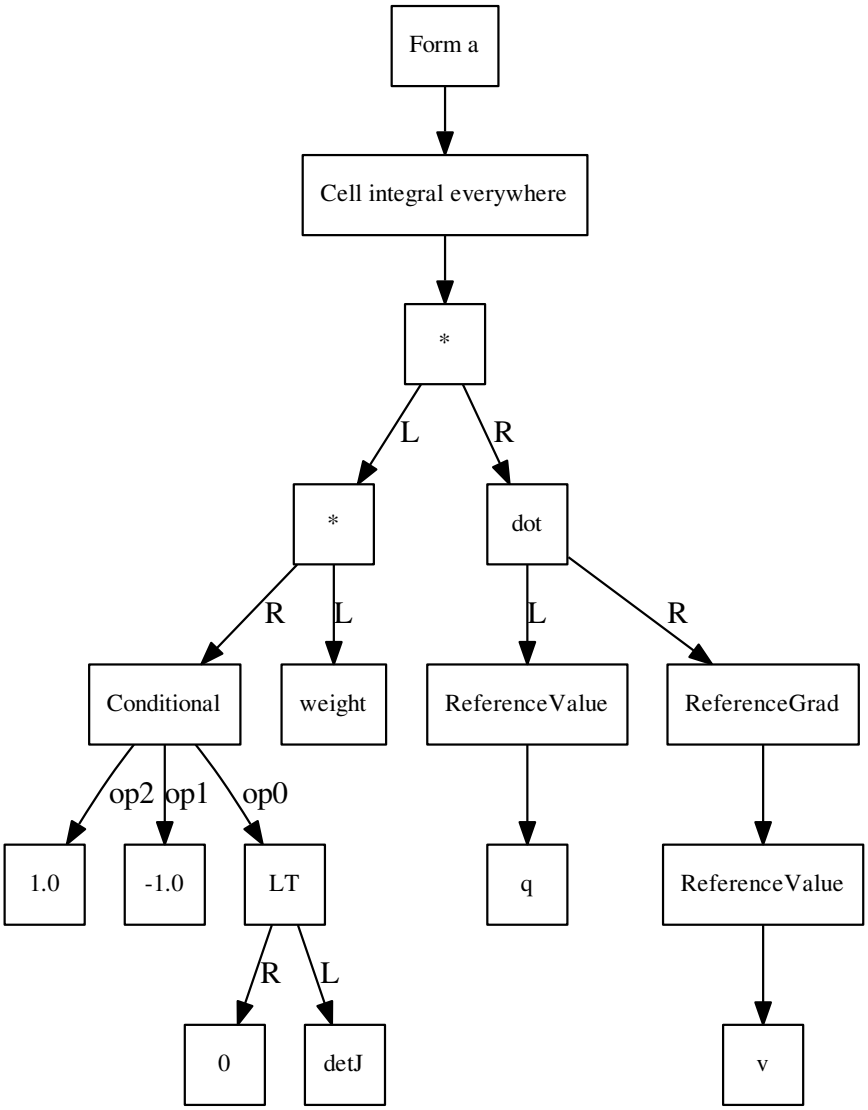

Figure 7.5: The first evaluation form after integral scaling and determinant cancellation.

7.2 Second form

Let $u = \mathcal{F}^{\mathrm{id}}(\hat{u})$ and $f \in H(\mathrm{div}; K)$. Define

$$a(u, f) = \int_K u \,\mathrm{div}\, f \,\mathrm{d}x \tag{7.4}$$

and let $\partial a_f(u, f; v)$ be the functional derivative of a with respect to f in the direction of $v = \mathcal{F}^{\mathrm{div}}(\hat{v})$.

It was already shown, in Section 3.6.2, that

$$\partial a_f(u, f; v) = \int_K u \,\mathrm{div}\, v \,\mathrm{d}x. \tag{7.5}$$

This is now exactly (2.1.73) in Boffi, Brezzi, and Fortin [7], which we have not reproduced; instead, we use (4.15) and a change of variables to get

$$\begin{aligned}
\partial a_f(u, f; v) &= \int_K u(x) \,\mathrm{div}\, v(x) \,\mathrm{d}x && (7.6)\\
&= \int_{\hat{K}} \hat{u}(\hat{x}) \left(\frac{1}{\det DF(\hat{x})} \widehat{\mathrm{div}}\, \hat{v}(\hat{x}) \right) |\det DF(\hat{x})| \,\mathrm{d}\hat{x} && (7.7)\\
&= \pm \int_{\hat{K}} \hat{u}(\hat{x}) \,\widehat{\mathrm{div}}\, \hat{v}(\hat{x}) \,\mathrm{d}\hat{x}. && (7.8)
\end{aligned}$$

This form illustrates:

- The structure-preserving functional derivative of the divergence operator.
- Implicit Jacobian cancellation in the divergence of an $\mathcal{F}^{\mathrm{div}}$-mapped function, based on (4.15).

The results of processing this form before and after the work of this project are shown in Listings 11 and 12 respectively.

```
Product(
  Product(
    QuadratureWeight(domain),
    Abs(JacobianDeterminant(domain))),
  Product(
    IndexSum(
      Indexed(
        ComponentTensor(
          Indexed(
            IndexSum(
              ComponentTensor(
                Product(
                  Indexed(
                    ComponentTensor(
                      Indexed(
                        ComponentTensor(
                          IndexSum(
                            Product(
                              Indexed(
                                JacobianInverse(
                                  domain),
                                MultiIndex((Index(19), Index(18)))),
                              Indexed(
                                ReferenceGrad(
                                  ReferenceValue(v)),
                                MultiIndex((Index(17), Index(19))))),
                            MultiIndex((Index(19),))),
                          MultiIndex((Index(17), Index(18)))),
                        MultiIndex((Index(13), Index(20)))),
                      MultiIndex((Index(20),))),
                    MultiIndex((Index(21),))),
                  Indexed(
                    ComponentTensor(
                      Product(
                        Indexed(
                          Jacobian(domain),
                          MultiIndex((Index(10), Index(11)))),
                        Division(
                          FloatValue(1.0),
                          JacobianDeterminant(domain))),
                      MultiIndex((Index(10), Index(11)))),
                    MultiIndex((Index(12), Index(13))))),
```

Listing 11: The processed second evaluation form, prior to this work (first 42 lines of 50).

```
Product(
  Product(
    QuadratureWeight(domain),
    Conditional(
      LT(
        JacobianDeterminant(domain),
        Zero((), (), ())),
      FloatValue(-1.0),
      FloatValue(1.0))),
  Product(
    IndexSum(
      Indexed(
        ReferenceGrad(
          ReferenceValue(v)),
        MultiIndex((Index(10), Index(10)))),
      MultiIndex((Index(10),))),
    ReferenceValue(u)))
```

Listing 12: The processed second evaluation form, subsequent to this work.

7.3 Third form

Let $u = \mathcal{F}^{\text{curl}}(\hat{u})$, $v = \mathcal{F}^{\text{curl}}(\hat{v})$, and $q = \mathcal{F}^{\text{div}}(\hat{q})$. Define

$$a(u, v, q) = \int_K u \cdot (q + \operatorname{curl} v) \, \mathrm{d}x. \tag{7.9}$$

Then, using a change of variables and applying pull-backs, including the use of (4.28), we have

$$\begin{aligned}
a(u, v, q) &= \int_{\hat{K}} \left([DF]^{-T}\hat{u}\right) \cdot \left(\left(\frac{1}{\det DF}[DF]\hat{q}\right) \right. && (7.10)\\
&\qquad \left. + \left(\frac{1}{|\det DF|}[DF]\,\widehat{\operatorname{curl}}\,\hat{v}\right) \right) |\det DF| \, \mathrm{d}\hat{x} && (7.11)\\
&= \int_{\hat{K}} \left([DF]^{-T}\hat{u}\right)^T \left((\pm[DF]\hat{q}) + \left([DF]\,\widehat{\operatorname{curl}}\,\hat{v}\right)\right) \mathrm{d}\hat{x} && (7.12)\\
&= \int_{\hat{K}} (\hat{u})^T [DF]^{-1}[DF] \left(\pm\hat{q} + \widehat{\operatorname{curl}}\,\hat{v}\right) \mathrm{d}\hat{x} && (7.13)\\
&= \int_{\hat{K}} \hat{u}^T \left(\pm\hat{q} + \widehat{\operatorname{curl}}\,\hat{v}\right) \mathrm{d}\hat{x} && (7.14)\\
&= \int_{\hat{K}} \hat{u} \cdot \left(\pm\hat{q} + \widehat{\operatorname{curl}}\,\hat{v}\right) \mathrm{d}\hat{x}. && (7.15)
\end{aligned}$$

This form illustrates:

- Jacobian cancellation in the dot product of an $\mathcal{F}^{\text{curl}}$-mapped function and an $\mathcal{F}^{\text{div}}$-mapped function according to (4.30).
- Jacobian cancellation in the dot product of an $\mathcal{F}^{\text{curl}}$-mapped function and the curl of another such function, based on (4.26) and (4.28). These circumstances were not discussed earlier.
- Jacobian cancellation in the presence of intermediate operators (in this case, the sum).

The results of processing this form before and after the work of this project are shown in Listings 13 and 14 respectively.

```
Product(
  Product(
    QuadratureWeight(domain),
    Abs(JacobianDeterminant(domain))),
  IndexSum(
    Product(
      Indexed(
        ComponentTensor(
          IndexSum(
            Product(
              Indexed(
                JacobianInverse(domain),
                MultiIndex((Index(15), Index(14)))),
              Indexed(
                ReferenceValue(u),
                MultiIndex((Index(15),)))),
            MultiIndex((Index(15),))),
          MultiIndex((Index(14),))),
        MultiIndex((Index(8),))),
      Indexed(
        Sum(
          ListTensor(
            Sum(
              Indexed(
                ComponentTensor(
                  Indexed(
                    IndexSum(
                      ComponentTensor(
                        Product(
                          Indexed(
                            JacobianInverse(
                              domain),
                            MultiIndex((Index(19), Index(18)))),
                          Indexed(
                            ComponentTensor(
                              Indexed(
                                ComponentTensor(
                                  IndexSum(
                                    Product(
                                      Indexed(
                                        JacobianInverse(
                                          domain),
```

Listing 13: The processed third evaluation form, prior to this work (first 42 lines of 256).

```
Product(
  Product(
    QuadratureWeight(domain),
    Conditional(
      LT(
        JacobianDeterminant(domain),
        Zero((), (), ())),
      FloatValue(-1.0),
      FloatValue(1.0))),
  IndexSum(
    Product(
      Indexed(
        Sum(
          ListTensor(
            Sum(
              Indexed(
                ReferenceGrad(
                  ReferenceValue(v)),
                MultiIndex((FixedIndex(2), FixedIndex(1)))),
              Product(
                IntValue(-1),
                Indexed(
                  ReferenceGrad(
                    ReferenceValue(v)),
                  MultiIndex((FixedIndex(1), FixedIndex(2))))))),
            Sum(
              Indexed(
                ReferenceGrad(
                  ReferenceValue(v)),
                MultiIndex((FixedIndex(0), FixedIndex(2)))),
              Product(
                IntValue(-1),
                Indexed(
                  ReferenceGrad(
                    ReferenceValue(v)),
                  MultiIndex((FixedIndex(2), FixedIndex(0))))))),
            Sum(
              Indexed(
                ReferenceGrad(
                  ReferenceValue(v)),
                MultiIndex((FixedIndex(1), FixedIndex(0)))),
              Product(
                IntValue(-1),
                Indexed(
                  ReferenceGrad(
                    ReferenceValue(v)),
                  MultiIndex((FixedIndex(0), FixedIndex(1)))))))),
          ReferenceValue(q)),
        MultiIndex((Index(17),))),
      Indexed(
        ReferenceValue(u),
        MultiIndex((Index(17),)))),
    MultiIndex((Index(17),))))
```

Listing 14: The processed third evaluation form, subsequent to this work.

Chapter 8

Conclusion and further work

This chapter summarises the contributions made by this project, and provides suggestions for further development.

8.1 Contributions

This project has successfully enabled Jacobian cancellation in UFL, as shown in Chapter 7. Specifically, the contributions made by the project are:

- The identification of a set of vector operators that can be helpfully preserved through automatic differentiation and function pull-backs.
- The addition of support for functional and spatial derivatives of expressions involving these vector operators.
- The automatic recognition of the special case of the divergence of a function on a finite element that uses the contravariant Piola mapping, and the application in this case of the identity (4.15), a form of implicit Jacobian cancellation.
- The automatic recognition of the special case of the curl of a function on a finite element that uses the covariant Piola mapping, and the application in this case of the identity (4.28), which leads to the emergence of a Jacobian matrix that may later be cancelled.
- An algorithm for performing the explicit cancellation of factors in an expression in a single traversal of the expression tree, in cases where the number of possible cancellations is small.
- Implementations of this cancellation for the Jacobian and its inverse, and for the determinant of the Jacobian.

8.2 Further work

Significant further work remains for the benefits from this solution to reach Firedrake users.

8.2.1 Updates to downstream tools

Tools downstream from UFL in the Firedrake toolchain may not yet recognise forms in which the evaluation of the Jacobian is unnecessary, and so may not yet reap the performance gain of avoiding that computation.

In many cases, despite significant cancellation, the Jacobian remains in the processed form through the sign of its determinant. We know mathematically that this factor is constant on each cell, and hence does not need to be recalculated at each relevant point in the cell, but downstream tools will need modification in order to realize the associated performance benefit. It seems likely that this quantity would have to be represented by its own type, much as the determinant of the Jacobian is, but this change cannot be made at the moment without affecting these tools.

8.2.2 Extension to mixed elements

Coefficients and arguments from different finite element spaces have in this project been obtained from separate elements, like this:

```
cell = triangle
rt_element = FiniteElement("RT", cell, degree=1)
cg_element = FiniteElement("CG", cell, degree=1)
q = Coefficient(rt_element)
v = Coefficient(cg_element)
```

A natural alternative, in UFL, is to obtain these coefficients and arguments from a mixed element, like this:

```
cell = triangle
rt_element = FiniteElement("RT", cell, degree=1)
cg_element = FiniteElement("CG", cell, degree=1)
element = MixedElement(rt_element, cg_element)
coeff = Coefficient(element)
q, v = split(coeff)
```

This use of a mixed element disallows Jacobian cancellation, as each split coefficient is a collection (a `ListTensor`) of the elements of the mixed element coefficient, a situation which the current Jacobian cancellation algorithm cannot handle. The extension to this case does not appear to be trivial.

Bibliography

[1] Martin S. Alnæs. "UFL: a Finite Element Form Language". In: *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Vol. 84. Lecture Notes in Computational Science and Engineering. 2012. Chap. 13. URL: https://fenicsproject.org/book/.

[2] Martin S. Alnæs et al. "The FEniCS Project Version 1.5". In: *Archive of Numerical Software* 3.100 (2015). DOI: 10.11588/ans.2015.100.20553.

[3] Martin S. Alnæs et al. "Unified Form Language: A Domain-specific Language for Weak Formulations of Partial Differential Equations". In: *ACM Trans. Math. Softw.* 40.2 (Mar. 2014), 9:1–9:37. ISSN: 0098-3500. DOI: 10.1145/2566630.

[4] Satish Balay et al. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Birkhäuser Press, 1997, pp. 163–202.

[5] Satish Balay et al. *PETSc Users Manual*. Tech. rep. ANL-95/11 – Revision 3.7. Argonne National Laboratory, 2016.

[6] Satish Balay et al. *PETSc Web page*. 2016. URL: http://www.mcs.anl.gov/petsc.

[7] Daniele Boffi, Franco Brezzi, and Michel Fortin. *Mixed Finite Element Methods and Applications*. Vol. 44. Springer Series in Computational Mathematics. Springer, 2013. ISBN: 978-3-642-36518-8. DOI: 10.1007/978-3-642-36519-5.

[8] Susanne C. Brenner and L. Ridgway Scott. *The mathematical theory of finite element methods*. Third edition. Vol. 15. Texts in Applied Mathematics. Springer, New York, 2008. ISBN: 978-0-387-75933-3. DOI: 10.1007/978-0-387-75934-0.

[9] Philippe G. Ciarlet. *Linear and nonlinear functional analysis with applications*. Vol. 130. Other Titles in Applied Mathematics. Siam, 2013.

[10] Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Vol. 4. Studies in Mathematics and its Applications. North-Holland, 1978.

[11] Rodney Coleman. "Differentiation". In: *Calculus on Normed Vector Spaces*. New York, NY: Springer New York, 2012, pp. 35–60. ISBN: 978-1-4614-3894-6. DOI: 10.1007/978-1-4614-3894-6_2.

[12] Lisandro D. Dalcin et al. "Parallel distributed computing using Python". In: *Advances in Water Resources* 34.9 (2011). New Computational Methods and Software Tools, pp. 1124–1139. DOI: http://dx.doi.org/10.1016/j.advwatres.2011.04.013.

[13] *Firedrake website*. Sept. 2017. URL: http://firedrakeproject.org.

[14] V. Girault and P.A. Raviart. *Finite Element Approximation of Navier-Stokes Equations*. Vol. 749. Lecture Notes in Mathematics. Springer-Verlag, 1979.

[15] Miklós Homolya et al. "TSFC: a structure-preserving form compiler". In: *CoRR* abs/1705.03667 (2017). URL: http://arxiv.org/abs/1705.03667.

[16] Robert C. Kirby. "Algorithm 839: FIAT, a New Paradigm for Computing Finite Element Basis Functions". In: *ACM Transactions on Mathematical Software* 30.4 (2004), pp. 502–516. DOI: 10.1145/1039813.1039820.

[17] Robert C. Kirby et al. "Common and unusual finite elements". In: *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Vol. 84. Lecture Notes in Computational Science and Engineering. Springer, 2012. Chap. 3. URL: https://fenicsproject.org/book/.

[18] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

[19] Graham R. Markall et al. "Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS". In: *28th International Supercomputing Conference, ISC, Proceedings*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Vol. 7905. Lecture Notes in Computer Science. Springer, 2013, pp. 279–289. DOI: 10.1007/978-3-642-38750-0_21. URL: http://dx.doi.org/10.1007/978-3-642-38750-0_21.

[20] Florian Rathgeber et al. “Firedrake: automating the finite element method by composing abstractions”. In: *ACM Trans. Math. Softw.* 43.3 (2016), 24:1–24:27. ISSN: 0098-3500. DOI: 10.1145/2998441. arXiv: 1501.01809.

[21] Florian Rathgeber et al. “PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes”. In: *High Performance Computing, Networking Storage and Analysis, SC Companion:* Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 1116–1123. ISBN: 978-1-4673-3049-7. DOI: 10.1109/SC.Companion.2012.134.

[22] P.A. Raviart and J.M. Thomas. “A mixed finite element method for second order elliptic problems”. In: *Mathematical Aspects of the Finite Element Method*. Ed. by I. Galligani and E. Magenes. Vol. 606. Lectures Notes in Mathematics. Springer-Verlag, 1977.

[23] Marie E. Rognes, Robert C. Kirby, and Anders Logg. “Efficient Assembly of H(div) and H(curl) Conforming Finite Elements”. In: *SIAM Journal on Scientific Computing* 31.6 (2009), pp. 4130–4151. DOI: 10.1137/08073901X.

[24] Tobias Schwedes et al. *Mesh dependence in PDE-constrained optimisation. An Application in Tidal Turbine Array Layouts*. First edition. SpringerBriefs in Mathematics of Planet Earth. Springer International Publishing, 2017. ISBN: 978-3-319-59483-5. DOI: 10.1007/978-3-319-59483-5.

Index