Unified Form Language: A domain-specific language for weak formulations of partial differential equations

MARTIN S. ALNÆS, Simula Research Laboratory
ANDERS LOGG, Simula Research Laboratory and University of Oslo
KRISTIAN B. ØLGAARD, Aalborg University
MARIE E. ROGNES, Simula Research Laboratory
GARTH N. WELLS, University of Cambridge

We present the Unified Form Language (UFL), which is a domain-specific language for representing weak formulations of partial differential equations with a view to numerical approximation. Features of UFL include support for variational forms and functionals, automatic differentiation of forms and expressions, arbitrary function space hierarchies for multi-field problems, general differential operators and flexible tensor algebra. With these features, UFL has been used to effortlessly express finite element methods for complex systems of partial differential equations in near-mathematical notation, resulting in compact, intuitive and readable programs. We present in this work the language and its construction. An implementation of UFL is freely available as an open-source software library. The library generates abstract syntax tree representations of variational problems, which are used by other software libraries to generate concrete low-level implementations. Some application examples are presented and libraries that support UFL are highlighted.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*; D.3.2 [**Programming Languages**]: Language Classifications—*Very high-level languages*; I.1.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation—*Representations (general and polynomial)*; G.1.8 [**Partial Differential Equations**]: Finite element methods; G.1.4 [**Quadrature and Numerical Differentiation**]: Automatic differentiation

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: AD, algorithmic differentiation, automatic functional differentiation, discretization, domain specific language, DSEL, DSL, einstein notation, embedded language, FEM, finite element method, functional, implicit summation, index notation, mixed element, partial differential equation, PDE, symbolic differentiation, tensor algebra, weak form, weak formulation,

ACM Reference Format:
ACM Trans. Math. Softw. V, N, Article A (YYYY), 38 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

We present a language for expressing variational forms of partial differential equations (PDEs) in near-mathematical notation. The language, known as the Unified Form

This research is partially supported by Research Council of Norway through grant no. 209951 and a Center of Excellence grant awarded to the Center for Biomedical Computing at Simula Research Laboratory. Author's addresses: M. S. Alnæs (martinal@simula.no) and A. Logg (logg@simula.no) and M. E. Rognes (meg@simula.no), Center for Biomedical Computing, Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway; K. B. Ølgaard (kbo@civil.aau.dk), Department of Civil Engineering, Aalborg University, Niels Bohrs Vej 8, 6700 Esbjerg, Denmark; G. N. Wells (gnw20@cam.ac.uk) Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, United Kingdom.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Language (UFL), inherits the typical mathematical operations that are performed on variational forms, thereby permitting compact and expressive computer input of mathematical problems. The complexity of the input syntax is comparable to the complexity of the classical mathematical presentation of the problem. The language is expressive in the sense that it provides basic, abstract building blocks which can be used to construct representations of complicated problems; it offers a mostly dimension-independent interface for defining differential equations; and it can be used to define problems that involve an arbitrary number of coupled fields. The language is developed with finite element methods in mind, but most of the design is not restricted to a specific numerical method.

UFL is a *language* for expressing variational statements of partial differential equations and does not provide a problem solving environment. Instead, it generates abstract representations of problems that can be used by *form compilers* to create concrete code implementations in general programming languages. There exist a number of form compilers that generate low-level code from UFL. These include the FEniCS Form Compiler (FFC) [Kirby and Logg 2006; Logg et al. 2012b; Ølgaard and Wells 2010; Rognes et al. 2009], the SyFi Form Compiler (SFC) [Alnæs and Mardal 2010, 2012] and the Manycore Form Compiler [Markall et al. 2012, 2010]. From a common UFL input, these compilers differ in the strategies used to create and optimize a low-level implementation, and in the target low-level language. The code generated by these form compilers can be used in a problem solving environment, linked at compile time or dynamically at runtime.

An example of a problem solving environment that uses code generated from UFL input is DOLFIN [Logg and Wells 2010; Logg et al. 2012c], which is developed as part of the FEniCS Project [Logg et al. 2012a]. Users of DOLFIN may describe a finite element discretization of a partial differential equation in UFL, and call a form compiler such as FFC or SFC to generate low-level code. In the case of FFC and SFC, this low-level code conforms to the UFC specification [Alnæs et al. 2009, 2012], which is a C++ interface for functionality related to evaluation of local stiffness matrices, finite element basis functions and local-to-global mappings of degrees of freedom. The UFC code may then be used by DOLFIN to assemble and solve the linear or nonlinear system corresponding to the finite element discretization described in UFL.

UFL is implemented as a domain-specific embedded language (DSEL) in Python. The distinction between a DSEL and a high-level software component lies in the level of expressiveness; UFL expressions can be composed and combined in arbitrary ways within the language design limits. Paraphrasing P. Hudak [Hudak 1996], a DSEL is the ultimate abstraction, allowing the user to reason about the program within the domain semantics, rather than within the semantics of the programming language. As an embedded language, UFL relies on the parser and grammar of the host language, Python. While it would be possible to select a subset of the Python grammar and write a UFL parser for that subset, we make no such restrictions in practice. UFL is implemented as a Python module which defines types (classes) and operators that together form an expressive language for representing weak formulations of partial differential equations. In addition, UFL provides a collection of algorithms for operating on UFL expressions. By implementing UFL as a DSEL in Python, we sacrifice some control over the syntax, but believe that this is overwhelmingly outweighed by the advantages. First, parsing is inherited and users may rely on all features of the Python programming language when writing UFL code, for example to define new operators. Second, it also permits the seamless integration of UFL into Python-based problem solving environments. The Python interface of the library DOLFIN is an example of this. In particular, the use of just-in-time (JIT) compilation facilitates the incorpora-

tion of UFL in a scripted environment without compromising the performance of a compiled language. This is discussed in detail in Logg and Wells [2010].

There have been a number of efforts to create domain-specific languages for scientific computing applications. Examples include SPL [Xiong et al. 2001] for signal processing and the Tensor Contraction Engine [Baumgartner et al. 2005] for quantum chemistry applications. In the context of partial differential equations, there have been a number of efforts to combine symbolic computing, code generation and numerical methods. In some cases the code generation is explicit, while in other cases, such as when employing templates, implicit. Early examples include FINGER [Wang 1986], the Symbolic Mechanics System [Korelc 1997], and Archimedes [Shewchuk and Ghattas 1993]. Analysa [Bagheri and Scott 2004] is an abstract finite element framework of limited scope built upon Scheme. Feel++ [Prud'homme 2006, 2011] uses C++ templates to create an embedded language for solving partial differential equations using finite element methods. Another example of a domain-specific language embedded in C++ is Sundance [Long et al. 2010]. Sundance relies heavily on automatic differentiation to provide a problem solving environment targeted at PDE-constrained optimization. UFL also provides automated differentiation of functionals and variational forms, but the approach differs in some respects from Sundance. This is discussed later in this work. UFL is distinguished from the aforementioned efforts by its combination of a high level of expressiveness, mathematically-driven abstractions, extensibility, breadth of supported mathematical operations and embedding in a modern, widely-used and freely available language (Python). Moreover, it is deliberately decoupled from a code generator and problem solving environment. This provides modularity and scope to pursue different code generation and/or solution strategies from a common description of a variational problem. This is highlighted by the existence of the different form compilers that support UFL, with each targeting a specific code generation strategy or architecture. Unlike some of the efforts listed above, UFL is freely available under a GNU public license (LGPLv3+).

The syntax used in UFL has its roots in FFC which was first released in 2005. At the time, FFC filled the roles of both form language and form compiler for the FEniCS Project. Much of the UFL syntax is inherited from early versions of FFC, but has since been re-implemented, generalized and extended to provide a more consistent mathematical environment, to cover a richer class of nonlinear forms and to provide a range of abstract algorithms, including differentiation. FFC no longer provides an input syntax, rather it generates code from a UFL representation. The UFL form language was first released in 2009 [Alnæs 2009] and has since then been tested on a wide range of applications. A rich and varied selection of applications that use UFL are presented in Logg et al. [2012a].

The remainder of this work is structured as follows. Section 2 summarizes the main mathematical concepts on which UFL is based. A detailed presentation of the UFL language is then given in Section 3. This is followed in Section 4 by a number of examples that demonstrate the use of UFL for a variety of partial differential equations. The subsequent sections focus on the technical aspects of the UFL design. In Sections 5 and 6, we describe the internal representation of UFL expressions and provide an overview of the algorithms provided by UFL, respectively. Particular emphasis is placed on differentiation. Section 7 provides a brief discussion of validation and code correctness. Some conclusions are then drawn in Section 8.

The implementation of UFL is available at `https://launchpad.net/ufl`. The examples presented in this work, including the UFL code used, are archived at `http://www.dspace.cam.ac.uk/handle/1810/243981`.

2. MATHEMATICAL CONCEPTS AND SCOPE

To clarify the notation, conventions, scope and assumptions of UFL and this paper, we begin by defining some key concepts in mathematical terms. We assume familiarity with variational formulations of PDEs and finite element methods. These variational formulations are assumed to be expressed as sums of integrals over geometric domains. Each integrand is an expression composed from a set of valid functions and geometric quantities, with various operators applied. Each such function is an element of a function space, typically, but not necessarily, a finite element space, while the set of permitted operators include differential operators and operators from tensor algebra. The central mathematical abstractions, including multi-linear variational forms, tensor algebra conventions and the finite element construction, are formally introduced in the subsections below.

When enumerating n objects, we count from 1 to n, inclusive, in the mathematical notation, while we count from 0 to $n-1$, inclusive, in computer code.

2.1. Variational forms

UFL is centered around expressing finite element variational forms, and in particular real-valued *multi-linear forms*. A real-valued multi-linear form a is a map from the product of a given sequence $\{V_j\}_{j=1}^{\rho}$ of function spaces:

$$a : V_\rho \times \cdots \times V_2 \times V_1 \to \mathbb{R}, \tag{1}$$

that is linear in each argument. The spaces V_j are labeled *argument spaces*. For the case $\rho \leq 2$, V_1 is referred to as the *test* space and V_2 as the *trial* space. The *arity* of a form ρ is the number of argument spaces. Forms with arity $\rho = 0$, 1, or 2 are named *functionals*, *linear forms* and *bilinear forms*, respectively. Such forms can be assembled on a finite element mesh to produce a scalar, a vector and a matrix, respectively. Note that the *argument functions* $\langle v_j \rangle_{j=\rho}^{1}$ are enumerated backwards such that their numbering matches the corresponding axis in the assembled tensor.

If the form a is parametrized over one or more *coefficient functions*, we express the form as the mapping from a product of a sequence $\{W_k\}_{k=1}^{n}$ of *coefficient spaces* and the argument spaces:

$$\begin{aligned} &a : W_1 \times W_2 \times \cdots \times W_n \times V_\rho \times \cdots \times V_2 \times V_1 \to \mathbb{R}, \\ &a \mapsto a(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1). \end{aligned} \tag{2}$$

Note that a is assumed to be (possibly) non-linear in the coefficient functions w_k and linear in the argument functions v_j. For a detailed exposition on finite element variational forms and assembly, we refer to [Kirby and Logg 2012] and references therein. To make matters concrete, we here list examples of some forms with different arity ρ and number of coefficients n:

$$a(u, v) := \int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x, \qquad \rho = 2, \quad n = 0, \tag{3}$$

$$a(\epsilon; u, v) := \int_\Omega \epsilon^2 \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x, \qquad \rho = 2, \quad n = 1, \tag{4}$$

$$a(f; v) := \int_\Omega f v \, \mathrm{d}x, \qquad \rho = 1, \quad n = 1, \tag{5}$$

$$a(u, v;) := \int_\Omega |\operatorname{grad}(u - v)|^2 \, \mathrm{d}x \qquad \rho = 0, \quad n = 2, \tag{6}$$

where Ω is the geometric domain of interest.

2.1.1. Geometric domains and integrals. UFL supports multi-linear forms defined via integration over geometric domains in the following manner. Let $\Omega \subset \mathbb{R}^d$ be a domain with boundary $\partial\Omega$ and let $\mathcal{T} = \{T\}$ be a suitable tessellation such that $\Omega = \bigcup_{T\in\mathcal{T}} T$. We denote the induced tessellation of $\partial\Omega$ by $\mathcal{F} = \{F\}$, and let $\mathcal{F}^0$ denote the set of internal facets of $\mathcal{T}$. Each of the three sets, the *cells* in $\mathcal{T}$, the *exterior facets* in $\mathcal{F}$ and the *interior facets* in $\mathcal{F}^0$, is assumed to be partitioned into one or more disjoint subsets:

$$\mathcal{T} = \bigcup_{k=1}^{n_c} \mathcal{T}_k, \quad \mathcal{F} = \bigcup_{k=1}^{n_f} \mathcal{F}_k, \quad \mathcal{F}^0 = \bigcup_{k=1}^{n_f^0} \mathcal{F}_k^0, \tag{7}$$

where n_c, n_f and n_f^0 denote the number of subsets of cells, exterior facets and interior facets, respectively. Given these definitions, it is assumed that the multi-linear form can be expressed in the following canonical form:

$$\begin{aligned} a(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1) = &\sum_{k=1}^{n_c} \sum_{T\in\mathcal{T}_k} \int_T I_k^c(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1)\,\mathrm{d}x \\ &+ \sum_{k=1}^{n_f} \sum_{F\in\mathcal{F}_k} \int_F I_k^f(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1)\,\mathrm{d}s \\ &+ \sum_{k=1}^{n_f^0} \sum_{F\in\mathcal{F}_k^0} \int_F I_k^{f,0}(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1)\,\mathrm{d}s, \end{aligned} \tag{8}$$

where $\mathrm{d}x$ and $\mathrm{d}s$ denote appropriate measures. The integrand I_k^c is integrated over the kth subset $\mathcal{T}_k$ of cells, the integrand I_k^f is integrated over the kth subset $\mathcal{F}_k$ of exterior facets and the integrand $I_k^{f,0}$ is integrated over the kth subset $\mathcal{F}_k^0$ of interior facets.

UFL in its current form does not model geometrical domains, but allows integrals to be defined over subdomains associated with an integer index k. It is then the task of the user, or the problem-solving environment, to associate the integral defined over the subdomain k with a concrete representation of the geometrical subdomain.

2.1.2. Differentiation of forms. Differentiation of variational forms is useful in a number of contexts, such as the formulation of minimization problems and computing Jacobians of nonlinear forms. In UFL, the derivative of a form is based on the Gâteaux derivative as detailed below.

Let f and v be coefficient and argument functions, respectively, with compatible domain and range. Considering a functional $M = M(f)$, the Gâteaux derivative of M with respect to f in the direction v is defined by

M(f;v)指M函数用于将f映射到v

$$M'(f; v) \equiv D_f M(f)[v] = \frac{d}{d\tau}\left[M(f + \tau v)\right]_{\tau=0}. \tag{9}$$

Given a linear form $L(f; v)$ (which could be the result of the above derivation) and another compatible argument function u, we can continue by computing the bilinear form $L'(f; u, v)$; that is, the derivative of L with respect to f in the direction u, defined by

$$L'(f; u, v) \equiv D_f L(f; v)[u] = \frac{d}{d\tau}\left[L(f + \tau u; v)\right]_{\tau=0}. \tag{10}$$

In general, this process can be applied to forms of general arity $\rho \geq 0$ to produce forms of arity $\rho + 1$. Note that if the form to be differentiated involves an integral, we assume that the integration domain does not depend on the differentiation variable. To express

the differentiation of a general form, consider the following compact representation of the canonical form (8):

$$F(\langle w_i \rangle_{i=1}^{n}; \langle v_j \rangle_{j=\rho}^{1}) = \sum_k \int_{D_k} I_k(\langle w_i \rangle_{i=1}^{n}; \langle v_j \rangle_{j=\rho}^{1}) \, \mathrm{d}\mu_k, \tag{11}$$

where $\{D_k\}$ and $\{\mathrm{d}\mu_k\}$ are the geometric domains and corresponding integration measures, and $\{I_k\}$ are the integrand expressions. We can then write the derivative of the general form (11) with respect to, for instance, w_1 in the direction $v_{\rho+1}$ as

$$D_{w_1} F(\langle w_i \rangle_{i=1}^{n}; \langle v_j \rangle_{j=\rho}^{1})[v_{\rho+1}] = \sum_k \int_{D_k} \frac{d}{d\tau} \left[I_k(w_1 + \tau v_{\rho+1}, \langle w_i \rangle_{i=2}^{n}; \langle v_j \rangle_{j=\rho}^{1}) \right]_{\tau=0} d\mu_k. \tag{12}$$

2.2. Tensors and tensor algebra

A core feature of UFL is its tensor algebra support. We summarize here some elementary tensor algebra definitions, notation and operations that will be used throughout this paper.

First, an *index* is either a fixed positive[1] integer value, in which case it is labeled a *fixed-index*, or a symbolic index with no value assigned, in which case it is called a *free-index*. A *multi-index* is an ordered tuple of indices, each of which can be free or fixed. Moreover, a *dimension* is a strictly positive integer value. A *shape* s is an ordered tuple of zero or more dimensions: $s = (s_1, \ldots, s_r)$; the corresponding rank $r \geqslant 0$ equals the length of the shape tuple.

Any tensor can be represented either as a mathematical object with a (tensor) shape or in terms of its scalar components with reference to a given basis. More precisely, the following notation and bases are used. Scalars are considered rank zero tensors. We denote by $\{e^i\}_{i=1}^{d}$ the standard orthonormal Euclidean basis for $\mathbb{R}^d$ of dimension d. A basis $\{E^\alpha\}_\alpha$ for $\mathbb{R}^s$, where $s = (s_1, \ldots, s_r)$ is a shape, is naturally defined via outer products of the vector basis:

$$E^\alpha \equiv e^{\alpha_1} \otimes \cdots \otimes e^{\alpha_r}, \tag{13}$$

where the range of the multi-index $\alpha = (\alpha_1, \ldots, \alpha_r)$ is such that $1 \leqslant \alpha_i \leqslant s_i$ for $i = 1, \ldots, r$. In general, whenever a multi-index α is used to index a tensor of shape s, it is assumed that $1 \leqslant \alpha_i \leqslant s_i$ for $i = 1, \ldots, r$. Then, the scalar component of index i of a vector v defined relative to the basis $\{e^i\}_i$ is denoted v_i. More generally, for a tensor C of shape s, C_α denotes its scalar component of multi-index α with respect to the basis $\{E^\alpha\}_\alpha$ of $\mathbb{R}^s$. Moreover, whenever we write $\sum_i v_i$, we imply $\sum_{i=1}^{d} v_i$, where d is the dimension of v. Correspondingly, for sums over multi-indices, $\sum_\alpha C_\alpha$ implies $\sum_{\alpha_1} \cdots \sum_{\alpha_r} C_\alpha$ with the deduced ranges.

Whenever one or more free-indices appear twice in a monomial term, summation over the free-indices is implied. Tensors v, A and C of rank 1, 2 and r, respectively, can be expressed using the summation convention as:

$$v = v_i e^i, \quad A = A_{ij} E^{ij}, \quad C = C_\alpha E^\alpha. \tag{14}$$

We will also consider general tensor-valued functions $f : \Omega \to \mathbb{R}^s$, $\Omega \subset \mathbb{R}^d$, where the shape s in this context is termed the *value shape*. Indexing of tensor-valued functions follows the same notation and assumptions as for tensors. Furthermore, derivatives with respect to spatial coordinates may be compactly expressed in index notation with the comma convention in subscripts. For example, for coordinates $(x_1, \ldots, x_d) \in \Omega$ and

[1] Indices are positive in the mathematical base 1 notation used here and non-negative in the base 0 notation used in the computer code.

a function $f : \Omega \to \mathbb{R}$, a vector function $v : \Omega \to \mathbb{R}^n$ or a tensor function $C : \Omega \to \mathbb{R}^s$, we write for indices i and j, and multi-indices α and β, with the length of β denoted by p:

$$f_{,i} \equiv \frac{\partial f}{\partial x_i}, \quad v_{i,j} \equiv \frac{\partial v_i}{\partial x_j}, \quad C_{\alpha,\beta} \equiv \frac{\partial^p C_\alpha}{\partial x_{\beta_1} \cdots \partial x_{\beta_p}}. \tag{15}$$

2.3. Finite element functions and spaces

A finite element space V_h is a linear space of piecewise polynomial fields defined relative to a tessellation $\mathcal{T}_h = \{T\}$ of a domain Ω. Such spaces are typically defined locally; that is, each field in the space is defined by its restriction to each cell of the tessellation. More precisely, for a finite element space V_h of tensor-valued functions of value shape s, we assume that

$$V_h = \{v \in \mathcal{H} \,:\, v|_T \in \mathcal{V}_T\}, \quad \mathcal{V}_T = \{v : T \to \mathbb{R}^s \,:\, v_\alpha \in \mathcal{P}(T) \quad \forall\, \alpha\}, \tag{16}$$

where the space $\mathcal{H}$ indicates the global regularity and where $\mathcal{P} = \mathcal{P}(T)$ is a specified (sub-)space of polynomials of degree $q \geqslant 0$ defined over T. In other words, the global finite element space V_h is defined by patching together local finite element spaces $\mathcal{V}_T$ over the tessellation $\mathcal{T}_h$. Note that the polynomial spaces may vary over the tessellation; however, this dependency is usually omitted for the sake of notational brevity.

The above definition may be extended to mixed finite element spaces. For given local finite element spaces $\{\mathcal{V}_i\}_{i=1}^n$ of respective value shapes $\{s^i\}_{i=1}^n$, we define the mixed local finite element space $\mathcal{W}$ by:

$$\mathcal{W} = \mathcal{V}_1 \times \mathcal{V}_2 \times \cdots \times \mathcal{V}_n = \{w = (v_1, v_2, \ldots, v_n) \,:\, v_i \in \mathcal{V}_i, \quad i = 1, 2, \ldots, n\}. \tag{17}$$

The extension to global mixed finite element spaces follows as in (16). Note that all element factors in a mixed element are assumed to be defined over the same cell T. The generalization to nested hierarchies of mixed finite elements follows immediately, and hence such hierarchies are also admitted

Any $w \in \mathcal{W}$ has the representation $w : T \to \mathbb{R}^t$, with a suitable shape tuple t, and the value components of w must be mapped to components of $v_i \in \mathcal{V}_i$. Let r^i be the corresponding rank of the value shape s^i, and denote by $p^i = \prod_{j=1}^{r^i} s^i_j$ the corresponding *value size*; i.e, the number of scalar components. In the general case, we choose a rank 1 shape $t = (\sum_{i=1}^n p^i)$, and map the flattened components of each v^i to components of the vector-valued w; that is,

$$w = (v^1_1, \ldots, v^1_{p^1}, \ldots, v^n_1, \ldots, v^n_{p^n}). \tag{18}$$

This mapping permits arbitrary combinations of value shapes s^i. In the case where $\mathcal{V}_i$ coincide for all $i = 1, \ldots, n$, we refer to the resulting specialized mixed element as a vector element of dimension n, and choose $t = (n, s^1_1, \ldots, s^1_{r^1})$. As a generalization of vector elements, we allow tensor elements[2] of shape c with rank q, which gives the value shape $t = (c_1, \ldots, c_q, s^1_1, \ldots, s^1_{r^1})$. Tensor elements built from scalar subelements may have symmetries, represented by a symmetry mapping from component to component. The number of subelements then equals $n = \left(\prod_{i=1}^q c_i\right) - m$, where m is the number of value components that are mapped to another.

For some finite element discretizations, it is helpful to represent the local approximation space as the *enrichment* of one element with another. More precisely, for local finite element spaces $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_n$ defined over a common cell T and of common value shape s, we define the space

$$\mathcal{W} = \mathcal{V}_1 + \mathcal{V}_2 + \cdots + \mathcal{V}_n = \{v_1 + v_2 + \cdots + v_n \,:\, v_i \in V_i, \quad i = 1, \ldots, n\}. \tag{19}$$

[2]Tensor-valued elements, not to be confused with tensor product elements.

Table I. Overview of element classes defined in UFL.

Finite element class specification
`FiniteElement(family, cell, degree)`
`VectorElement(family, cell, degree, (dim))`
`TensorElement(family, cell, degree, (shape), (symmetry))`
`MixedElement(elements)`
`EnrichedElement(elements)`
`RestrictedElement(element, domain)`

In addition to the arguments given here, a specific quadrature scheme can be given for the primitive finite elements (those defined by family name). Arguments in parentheses are optional.

Again, the extension to global enriched finite element spaces follows as in (16). The MINI element [Arnold et al. 1984] for the Stokes equations is an example of an enriched element.

3. OVERVIEW OF THE LANGUAGE

UFL can be partitioned into sublanguages for finite elements, expressions, and forms. We will address each separately below. Overall, UFL has a declarative nature similar to functional programming languages. Side effects, sequences of statements, subroutines and explicit loops found in imperative programming languages are all absent from UFL. The only branching instructions are inline conditional expressions, which will be further detailed in Section 3.2.6.

3.1. Finite elements

The UFL finite element sublanguage provides syntax for finite elements and operations over finite elements, including mixed and enriched finite elements, as established in Section 2.3.

3.1.1. Finite element abstractions and classes. UFL provides four main finite element abstractions: primitive finite elements, mixed finite elements, enriched finite elements and restricted finite elements. Each of these abstractions provides information on the value shape, the cell and the embedding polynomial degree of the element (see (16)), and each is further detailed below. We remark that UFL is primarily concerned with properties of local finite element spaces: the global continuity requirement and the specific implementation of the element degrees of freedom or basis functions are not covered by UFL. For an overview of finite element abstractions with initialization arguments, see Table I. Example usage will be presented in Section 4.

In the literature, it is common to refer to finite elements by their family parametrized by cell type and order: for instance, the "Nédélec face elements of the second kind over tetrahedra of second order" [Nédélec 1986]. The global continuity requirements are typically implied by the family: for instance, it is generally assumed that the aforementioned Nédélec face element functions indeed do have continuous normal components across faces. Moreover, finite elements may be known by different family names, for instance the aforementioned Nédélec face elements coincide with the Brezzi–Douglas–Marini elements on tetrahedra [Brezzi et al. 1985], which again coincide with the $\mathcal{P}\Lambda^2(T)$ family on tetrahedra [Arnold et al. 2006].

UFL mimics the literature in the sense that primitive finite elements are defined in terms of a family, a cell and a polynomial degree via the `FiniteElement` class (see Table I). Additionally, a quadrature scheme label can be given as an optional argument. The family must be an identifying string, while the cell is a description of the cell type of the geometric domain. The UFL documentation contains the comprehensive list of

preregistered families and cells. Multiple names in the literature for the same finite element are handled via family aliases. UFL supports finite element exterior calculus notation for simplices in one, two or three dimensions via such aliases. By convention, elements of a finite element family are numbered in terms of their polynomial degree q such that their fields are indeed included in the complete polynomial space of degree q. This facilitates internal consistency, although it might conflict with some notation in the literature. For instance, the lowest order Raviart–Thomas elements have degree 1 in UFL.

Syntax is provided for defining vector elements. The `VectorElement` class accepts a family, a cell, a degree and the dimension of the vector element. The dimension defaults to the geometric dimension d of the cell. The value shape of a vector element is then (d, s) where s is the value shape of the corresponding finite element of the same family. This corresponding element may be vector-valued, e.g. a `VectorElement("BDM", triangle, p)` has value shape $(2, 2)$. Moreover, further structure can be imposed for (higher-dimensional) vector elements with a rank two tensor structure. The `TensorElement` class accepts a family, a cell and a degree, and in addition a shape and a symmetry argument. The shape argument defaults to the tuple (d, d) and the value shape of the tensor element is then (d, d, s). The symmetry argument may be boolean true to define the symmetry $A_{ij} = A_{ji}$ if the value rank is two. It may also be a mapping between the component tuples that should be equal, such as $\{$`(0,0):(0,1), (1,0):(1,1)`$\}$ to define the symmetries $A_{11} = A_{12}$, $A_{21} = A_{22}$. The vector and tensor element classes can be viewed as optimized, special cases of mixed finite elements.

In general, mixed finite elements in UFL are created from a tuple of subelements through the `MixedElement` class. Each subelement can be a finite, vector, or tensor element as described above, or in turn a general mixed element. The latter can lead to nested mixed finite elements of arbitrary, though finite, depth. All subelements must be defined over the same geometric cell and utilize the same quadrature scheme (if prescribed). The degree of a mixed finite element is defined to be the maximal degree of the subelements. Note that mixed finite elements are recursively flattened. Their value shape is $(s,)$ where s is the total number of scalar components.

Enriched elements can be defined via the `EnrichedElement` class, given a tuple of finite, vector, tensor, or mixed subelements. The subelements must be defined on the same cell and have the same value shape. These then define the cell and value shape of the enriched element. The degree is inferred as the maximal degree of the subelements.

Finally, UFL also offers a restricted element abstraction via the `RestrictedElement` class, taking as arguments any of the element classes described above and a cell or the string `"facet"`. The term restricted in this setting refers to the elimination of element functions that vanish on the given cell entities; Labeur and Wells [2012] provide an example utilizing elements restricted to cell facets. The value shape, cell and degree of a restricted element are directly deduced from the defining element.

3.1.2. Operators over finite elements. For readability and to reflect mathematical notation, UFL provides some operators over the finite element classes defined in the previous section. These operators include the binary operators multiplication (`*`) and addition (`+`), and an indexing operator (`[]`). These operators and their long-hand equivalents are presented in Table II. The multiplication operator acts on two elements to produce a mixed element with the two elements as subelements in the given order. Note that the multiplication operator (in Python) is binary, so multiplication of three or more elements produces a nested mixed element. Similarly, the addition operator acts on two elements to yield an enriched element with the two given elements as subelements. Finally, the indexing operator restricts an element to the cell entity given by the argument to `[]`, thus returning a restricted element.

Table II. An overview of UFL operators over elements: examples of operator usage matched with the equivalent verbose syntax.

Operation	Equivalent syntax
`M = U * V`	`M = MixedElement(U, V)`
`M = U * V * W`	`M = MixedElement((U, V), W)`
`M = U + V`	`M = EnrichedElement(U, V)`
`M = V['facet']`	`M = RestrictedElement(V, 'facet')`

3.2. Expressions

The language for declaring expressions consists of a set of terminal expression types and a set of operators acting on other expressions. Each expression is represented by an object of a subclass of `Expr`. Each operator acts on one or more expressions and produces a new expression. An operator result is uniquely defined by its operator type and its operand expressions, and cannot have non-expression data associated with it. A terminal expression does not depend on any other expressions and typically has non-expression data associated with it, such as a finite element, geometry data or the values of literal constants. Terminal expression types are subclasses of `Terminal` and operator results are represented by subclasses of `Operator`, both of which are subclasses of `Expr`. Any UFL expression object is the root of a self-contained expression tree in which each tree node is an `Expr` object. The references from objects of `Operator` subtypes to the operand expressions represent directed edges in the tree and objects of `Terminal` subtypes terminate the tree.

As an embedded language, UFL allows the use of Python variables to store subexpression references for reuse. However, UFL itself does not have the concept of mutable variables. In fact, a key property of all UFL expressions, including terminal types, is their immutable state.[3] Immutable state is a prerequisite for the reuse of subexpression objects in expression trees by reference instead of by copying. This aspect is critical for an efficient symbolic software implementation.

The dependency set of an expression is the set of non-literal terminal expressions that can be reached from the expression root. An expression with an empty dependency set can be evaluated symbolically, but in general the evaluation of a UFL expression can only be carried out when the values of its dependencies are known. Numerical evaluation of the symbolic expression without code generation is possible when such values are provided, but this is an expensive operation and not suitable for large scale numerical computations.

Every expression is considered to be tensor-valued and its shape must always be defined. Furthermore, every expression has a set of free-indices. Note that the free-index set of any particular expression object is not associated with its shape; for instance, if A is a rank two tensor with shape $(3, 3)$ (and no free indices), then A_{ij} is a rank zero tensor expression; in other words, scalar-valued and with the associated free-indices i and j. Mathematically one could see A and A_{ij} as being the same, but represented as objects in software they are distinct. While A represents a matrix-valued expression, A_{ij} represents any scalar value of A. Because the tensor properties of all subexpressions are known, dimension errors and inconsistent use of free-indices can be detected early. The following sections describe terminal expressions, the index notation and various operators in more detail.

3.2.1. Terminal expressions. Terminal expressions in UFL include literal constants, geometric quantities and functions. In particular, UFL provides a domain-specific set of

[3] In the PyDOLFIN library, UFL function types are subclassed to carry additional mutable state which does not affect their symbolic meaning. UFL algorithms carefully preserve this information.

Table III. Tables of literal tensor constants.

Mathematical notation	UFL notation
I	`I = Identity(2)`
ϵ	`eps = PermutationSymbol(3)`
e_x, e_y	`ex, ey = unit_vectors(2)`
$e_x e_x, e_x e_y, e_y e_x, e_y e_y$	`exx, exy, eyx, eyy = unit_matrices(2)`

Table IV. Tables of non-literal terminal expressions.

Geometric quantities		*Functions*	
Math.	UFL notation	Math.	UFL notation
x	`x = cell.x`	$c \in \mathbb{R}$	`c = Constant(cell)`
n	`n = cell.n`	$g \in \mathcal{V}$	`g = Coefficient(element)`
$\|T\|$	`h = cell.volume`	$w \in \mathcal{V}$	`w = Argument(element)`
$r(T)$	`r = cell.circumradius`	$u \in \mathcal{V}$	`u = TrialFunction(element)`
$\|F\|$	`fa = cell.facetarea`	$v \in \mathcal{V}$	`v = TestFunction(element)`
$\sum_{F \subset T} \|F\|$	`ca = cell.cellsurfacearea`		

The examples are given with reference to a predefined cell $T \subset \mathbb{R}^d$ denoted `cell`, with coordinates $x \in \mathbb{R}^d$, facets $\{F\}$ and facet normal n, and a predefined local finite element space $\mathcal{V}$ of some finite element denoted `element`. $|\cdot|$ denotes the volume, while $r(T)$ denotes the circumradius of the cell T; that is, the radius of the circumscribed sphere of the cell.

types within these three fairly generic groups. Tables III and IV provide an overview of the literal constants, geometric quantities and functions available.

Literal constants include integer and real-valued constants, the identity matrix, the Levi–Civita permutation symbol and unit tensors. Geometric quantities include spatial coordinates and cell derived quantities, such as the facet normal, facet area, cell volume, cell surface area and cell circumradius. Some of these are only well defined when restricted to facets, so appropriate errors are emitted if used elsewhere.

Functions are cell-wise or spatially varying expressions. These are central to the flexibility of UFL. However, in contrast to other UFL expressions, functions are merely symbols or placeholders. Their values must generally be determined outside of UFL. All functions are defined over function spaces, introduced in Section 3.1, such that their tensor properties, including their shape, can be derived from the function space. Functions are further grouped into coefficient functions and argument functions. Expressions must depend linearly on any argument functions; see Section 2.1. No such limitations apply to dependencies on geometric quantities or coefficient functions. Functions are counted, or assigned a count, as they are constructed, and so the order of construction matters. In particular, different functions are assumed to have different counts. The ordering of the arguments to a form of rank 2 (or higher) is determined by the ordering of these counts. For convenience, UFL provides constructors for argument functions called `TestFunction` and `TrialFunction` which apply a fixed ordering to avoid accidentally transposing bilinear forms.

In the FEniCS pipeline, functions are evaluated as part of the form compilation or the assembly process. Argument functions are interpreted by the form compiler as placeholders for each basis function in their corresponding local finite element spaces, which are looped over when computing the local element tensor. Moreover, the ordering of the argument functions (defined by their counts) determines which global tensor axis each argument is associated with when assembling the global tensor (such as a sparse matrix) from a form. Form compilers typically specialize the evaluation of the basis functions during compilation, but may theoretically keep the choice of element space open until runtime. On the other hand, coefficient functions are used to repre-

Table V. Table of indexing operators: `i`, `j`, `k`, `l` are free-indices, while `a`, `b`, `c` are other expressions.

Mathematical notation	UFL notation
A_i	`A[i]`
B_{ijkl}	`B[i,j,k,l]`
$\langle a, b, c \rangle$	`as_vector((a, b, c))`
$A = B_i e_i, \quad (A_i = B_i)$	`as_vector(B[i], i)`
$A = B_{ji} E^{ij}, \quad (A_{ij} = B_{ji})$	`as_matrix(B[j,i], (i,j))`
$A = B_{klij} E^{ijkl}, \quad (A_{ijkl} = B_{klij})$	`as_tensor(B[k,l,i,j], (i,j,k,l))`

Table VI. Table of tensor algebraic operators

Math. notation	UFL notation
$A + B$	`A + B`
$A \cdot B$	`dot(A, B)`
$A : B$	`inner(A, B)`
$AB \equiv A \otimes B$	`outer(A, B)`
$A \times B$	`cross(A, B)`
A^T	`transpose(A)`
$\operatorname{sym} A$	`sym(A)`
$\operatorname{skew} A$	`skew(A)`
$\operatorname{dev} A$	`dev(A)`
$\operatorname{tr} \equiv A_{ii}$	`tr(A)`

Math. notation	UFL notation
$\det A$	`det(A)`
$\operatorname{cofac} A$	`cofac(A)`
A^{-1}	`inv(A)`
$v \quad \vert \quad v_i = A_{ii}$ (no sum)	`diag_vector(A)`
$A \quad \vert \quad A_{ij} = \begin{cases} B_{ij}, \text{ if } i = j, \\ 0, \text{ otherwise} \end{cases}$	`diag(B)`
$A \quad \vert \quad A_{ij} = \begin{cases} v_i, \text{ if } i = j, \\ 0, \text{ otherwise} \end{cases}$	`diag(v)`

sent global constants, finite element fields or any function that can be evaluated at spatial coordinates during finite element assembly. The limitation to functions of spatial coordinates is necessary for the integration of forms to be a cell-wise operation.

3.2.2. Index notation. UFL mirrors conventional index notation by providing syntax for defining fixed and free-indices and for indexing objects. For convenience, free-index objects with the names `i, j, k, l, p, q, r, s` are predefined. However, these can be redefined and new ones created. A single free-index object is created with `i = Index()`, while multiple indices are created with `j, k, l = indices(3)`.

The main indexing functionality of UFL is summarized in Table V. The indexing operator `[]` applied to an expression yields a component-wise representation. For instance, for a rank two tensor A (`A`) and free-indices `i, j`, `A[i, j]` yields the component-wise representation A_{ij}. The mapping from a scalar-valued expression with components identified by free-indices to a tensor-valued expression is performed via `as_tensor(A[i, j], (i, j))`. The `as_vector`, `as_matrix`, `as_tensor` functions can also be used to construct tensor-valued expressions from explicit tuples of scalar components. Note how the combination of indexing and `as_tensor` allows the reordering of the tensor axes in the expression $A_{ijkl} = B_{klij}$. Finally, we remark that fixed and free-indices can be mixed during indexing and that standard slicing notation is available.

3.2.3. Arithmetic and tensor algebraic operators. UFL defines arithmetic operators such as addition, multiplication, l^2 inner, outer and cross products and tensor algebraic operators such as the transpose, the determinant and the inverse. An overview of common operators of this kind is presented in Table VI. These operators will be familiar to many readers and we therefore make only a few comments below.

Addition and subtraction require that the left and right operands have the same shape and the same set of free-indices. The result inherits those properties.

In the context of tensor algebra, the concept of a product is heavily overloaded. Therefore, the product operator $*$ has no unique intuitive definition. Our choice in

Table VII. (Left) Table of elementary, nonlinear functions. (Right) Table of trigonometric functions.

Math. notation	UFL notation
a/b	`a/b`
a^b	`a**b`, `pow(a,b)`
$\sqrt{f}$	`sqrt(f)`
$\exp f$	`exp(f)`
$\ln f$	`ln(f)`
$\|f\|$	`abs(f)`
$\operatorname{sign} f$	`sign(f)`

Math. notation	UFL notation
$\cos f$	`cos(f)`
$\sin f$	`sin(f)`
$\tan f$	`tan(f)`
$\arccos f$	`acos(f)`
$\arcsin f$	`asin(f)`
$\arctan f$	`atan(f)`

Table VIII. (Left) Table of special functions. (Right) Table of element-wise operators.

Math. notation	UFL notation
$\operatorname{erf} f$	`erf(f)`
$J_\nu(f)$	`bessel_J(nu, f)`
$Y_\nu(f)$	`bessel_Y(nu, f)`
$I_\nu(f)$	`bessel_I(nu, f)`
$K_\nu(f)$	`bessel_K(nu, f)`

Math. notation			UFL notation
C	$\mid$	$C_\alpha = A_\alpha B_\alpha$	`elem_mult(A, B)`
C	$\mid$	$C_\alpha = A_\alpha / B_\alpha$	`elem_div(A, B)`
C	$\mid$	$C_\alpha = A_\alpha^{B_\alpha}$	`elem_pow(A, B)`
C	$\mid$	$C_\alpha = f(A_\alpha, \ldots)$	`elem_op(f, A, ...)`

UFL is to define $*$ as the product of two scalar-valued operands, one scalar and one tensor valued operand, and additionally as the matrix–vector and matrix–matrix product. The `inner` function defines the inner product between tensors of the same shape, while `dot` acts on two tensors by contracting the last axis of the first argument and the first axis of the second argument. If both arguments are vector-valued, the action of `dot` coincides with that of `inner`.

When applying the product operator $*$ to operands with free indices, summation over repeated free-indices is implied. Implicit summation is only allowed when at least one of the operands is scalar-valued, and the tensor algebraic operators assume that their operands have no repeated free-indices.

3.2.4. Nonlinear scalar functions. UFL provides a number of familiar nonlinear scalar real functions, listed in Tables VII and VIII. All of these elementary, trigonometric or special functions assume a scalar-valued expression with no free-indices as argument. Their mathematical meaning is well established and implementations are available in standard C++ or Boost [Boost 2012]. To apply any scalar function to the components of a tensor-valued expression, the element-wise operators listed in Table VIII can be used.

3.2.5. Differential operators and explicit variables. UFL supports a range of differential operators. Most of these mirror common ways of expressing spatial derivatives in partial differential equations. A summary is presented in Table IX (left).

Basic partial derivatives, $\partial/\partial x_i$, can be written as either `A.dx(i)` or `Dx(A,i)`, where `i` is either a free-index or a fixed-index in the range $[0, d)$. In the literature, there exist (at least) two conventions for the gradient and the divergence operators depending on whether the spatial derivative axis is appended or prepended; or informally, whether gradients and divergences are taken column-wise or row-wise. We denote the former convention by $\operatorname{grad}(v)$ and the latter by ∇v. The two choices are reflected in UFL via two gradient operators: `grad(A)` corresponding to $\operatorname{grad}(v)$, and `nabla_grad(A)` corresponding to ∇v. The two divergence operators `div(A)` and `nabla_div(A)` follow the corresponding traditions. Also available are the curl operator and its synonym rot, as well as a shorthand notation for the normal derivative.

Expressions can also be differentiated with respect to user-defined *variables*. An expression `e` is annotated as a variable via `v = variable(e)`. Letting `A` be some ex-

Table IX. (Left) Table of differential operators. (Right) Table of conditional operators.

Math. notation	UFL notation
$A_{,i}$ or $\frac{\partial A}{\partial x_i}$	`A.dx(i)` or `Dx(A, i)`
$\frac{dA}{dn}$	`Dn(A)`
$\operatorname{div} A$	`div(A)`
$\operatorname{grad} A$	`grad(A)`
$\nabla \cdot A$	`nabla_div(A)`
$\nabla A \equiv \nabla \otimes A$	`nabla_grad(A)`
$\operatorname{curl} A \equiv \nabla \times A$	`curl(A)`
$\operatorname{rot} A$	`rot(A)`
$v = e$	`v = variable(e)`
$\frac{dA}{dv}$	`diff(A, v)`
$\mathrm{d}f$	`exterior_derivative(f)`

Math. notation	UFL notation
$\begin{cases} T, \text{ if } c \\ F, \text{ otherwise} \end{cases}$	`conditional(c, T, F)`
$a = b$	`eq(a, b)`
$a \neq b$	`ne(a, b)`
$a \leq b$	`le(a, b)` or `a <= b`
$a \geq b$	`ge(a, b)` or `a >= b`
$a < b$	`lt(a, b)` or `a < b`
$a > b$	`gt(a, b)` or `a > b`
$l \wedge r$	`And(l, r)`
$l \vee r$	`Or(l, r)`
$\neg c$	`Not(c)`

Table X. (Left) Table of discontinuous Galerkin operators. (Right) Table of subdomain integrals.

Mathematical notation	UFL notation
f^+, f^-	`f('+'), f('-')`
$\langle f \rangle$	`avg(f)`
$[\![f]\!]$	`jump(f)`
$[\![f]\!]_n$	`jump(f, n)`

Mathematical notation	UFL notation
$\int_{\mathcal{T}_{k+1}} I \,\mathrm{d}x$	`I * dx(k)`
$\int_{\mathcal{F}_{k+1}} I \,\mathrm{d}s$	`I * ds(k)`
$\int_{\mathcal{F}^0_{k+1}} I \,\mathrm{d}s$	`I * dS(k)`

pression of `v`, the derivative of `A` with respect to `v` then reads as `diff(A, v)`. Note that `diff(e, v) == 0`, because the expression `e` is not a function of `v`, just as $\frac{\mathrm{d}f(x)}{\mathrm{d}g(x)} = 0$ even if the equality $f(x) = g(x)$ holds for two functions f and g since f is defined in terms of x and not in terms of g.

3.2.6. Conditional operators. The lack of control flow statements and mutable variables in UFL is offset by the inclusion of conditional statements, which are equivalent to the ternary operator known from, for example, the C programming language. To avoid issues with overloading the meaning of some logical operators, named operators are available for all boolean UFL expressions. In particular, the equivalence operator `==` is reserved in Python for comparing if objects are equal, and the delayed evaluation behavior of Python operators `and`, `or` and `not` preclude their use as DSL components. The available conditional and logical operators are listed in Table IX (right) and follow the standard conventions.

3.2.7. Discontinuous Galerkin operators. UFL facilitates compact implementation of discontinuous Galerkin methods by providing a set of operators targeting discontinuous fields. Discontinuous Galerkin discretizations typically rely on evaluating jumps and averages of (discontinuous) piecewise functions, defined relative to a tessellation, on both sides of cell facets. More precisely, let F denote an interior facet shared by the cells T^+ and T^-, and denote the restriction of an expression f to T^+ and T^- by f^+ and f^-, respectively. In UFL, the corresponding restrictions of an expression `f` are expressed via `f('+')` and `f('-')`.

Two typical discontinuous Galerkin operators immediately derive from these restrictions: the *average* $\langle f \rangle = (f^+ + f^-)/2$, and *jump* operators $[\![f]\!] = f^+ - f^-$. For convenience, these two operators are available in UFL via `avg(f)` and `jump(f)`. Moreover, it is common to use the outward unit normal to the interior facet, n, when defining the jump operator such that for a scalar-valued expression $[\![f]\!]_n = f^+ n^+ + f^- n^-$, while for a vector- or tensor-valued expression $[\![f]\!]_n = f^+ \cdot n^+ + f^- \cdot n^-$. These two definitions are implemented in a single UFL operator `jump(f, n)` by letting UFL automatically determine the rank of the expression and return the appropriate definition. The available operators are presented in Table X (left).

Since UFL is an embedded language, a user can easily implement custom operators for discontinuous Galerkin methods from the basic restriction building blocks. The reader is referred to Ølgaard et al. [2008] for more details on discontinuous Galerkin methods in the context of a variational form language, developed for FFC, and automated code generation.

3.3. Integrals and variational forms

In addition to expressions, UFL provides concepts and syntax for defining integrals over facets and cells and, via integrals, for defining variational forms. Variational forms can further be manipulated and transformed via form operations. This sublanguage is described in the sections below.

3.3.1. Integrals and forms. The integral functionality provided by UFL is centered around the integrals featuring in variational forms as summarized by the canonical expression (8). An integral is generally defined by an integration domain, an integrand and a measure. UFL admits integrands defined in terms of terminal expressions and operators as described in the previous section. The mathematical concept of an integration domain together with a corresponding measure is however embodied in a single abstraction in UFL, namely the `Measure` class. This abstraction was inherited from the original FFC form language.

A UFL measure (for simplicity just a measure from here onwards) is defined in terms of a domain type, a domain identifier and, optionally, additional domain meta-data. The allowed domain types include `"cell"`, `"exterior_facet"` and `"interior_facet"`. These domain types correspond to Lebesgue integration over (a subset of) cells, (a subset of) exterior facets or (a subset of) interior facets. The domain identifier must be a non-negative integer (the index k in (8)). For convenience, three measures are predefined by UFL: `dx`, `ds` and `dS` corresponding to measures over cells, exterior facets and interior facets, respectively, each with the default domain identifier 0. However, new measures can be created directly, or by calling a measure with an integer k yielding a new measure with domain identifier k. Several other, less common, domain types are allowed, such as `"surface"`, `"point"` and `"macro_cell"`, and new measures are easily added. We refer to the UFL documentation for the complete description of these.

A UFL integral is then defined via an expression, acting as the integrand, and a measure object. In particular, multiplying an expression with a measure yields an integral. This is illustrated in Table X (right), with `k` denoting the domain identifier. We remark that all integrand expressions featuring in an integral over interior facets must be restricted for the integral to be admissible. The integrand expression will depend on a number of distinct argument functions: the number of such is labeled the arity of the integral.

Finally, a UFL form is defined as the sum of one or more integrals. A form may have integral terms of different arities. However, if all terms have the same arity ρ we term this ρ the arity of the form. Forms are labeled according to their arity: forms of arity 0 are called functionals, forms of arity 1 are called linear forms, and forms of arity 2 are called bilinear forms. We emphasize that the number of coefficient functions does not affect the arity of an integral or a form. Table XI shows simple examples of functionals, linear forms and bilinear forms, while more complete examples are given in Section 4.

3.3.2. Form operators. UFL provides a select but powerful set of algorithms that act on forms to produce new forms. An overview of such algorithms is presented and exemplified in Table XII.

The first three operators in Table XII, `lhs`, `rhs` and `system`, extract terms of certain arities from a given form. More precisely, `lhs` returns the form that is the sum of all integrals of arity 2 in the given form, `rhs` returns the form that is the *negative* sum

Table XI. Table of various form examples.

Mathematical notation	UFL notation	Form type
$M(f;) = \int_{\mathcal{T}_1} \frac{1-f^2}{1+f^2}\,\mathrm{d}x$	`M = (1-f**2)/(1+f**2) * dx`	Functional
$M(f,g;) = \int_{\mathcal{T}_1} \operatorname{grad} f \cdot \operatorname{grad} g\,\mathrm{d}x$	`M = dot(grad(f), grad(g)) * dx`	Functional
$L(v) = \int_{\mathcal{T}_1} \sin(\pi x) v\,\mathrm{d}x$	`L = sin(pi*x[0])*v * dx`	Linear form
$L(g;v) = \int_{\mathcal{F}_1} (\operatorname{grad} g \cdot n) v\,\mathrm{d}s$	`L = Dn(g)*v * ds`	Linear form
$a(u,v) = \int_{\mathcal{T}_1} uv\,\mathrm{d}x$	`a = u*v * dx`	Bilinear form
$a(u,v) = \int_{\mathcal{T}_1} uv\,\mathrm{d}x + \int_{\mathcal{F}_2} fuv\,\mathrm{d}s$	`a = u*v * dx(0) + f*u*v * ds(1)`	Bilinear form
$a(u,v) = \int_{\mathcal{F}_1^0} \langle u\rangle\langle v\rangle\,\mathrm{d}s$	`a = avg(u)*avg(v) * dS`	Bilinear form
$a(A;u,v) = \int_{\mathcal{T}_1} A_{ij} u_{,i} v_{,j}\,\mathrm{d}x$	`a = A[i,j]*u.dx(i)*v.dx(j) * dx`	Bilinear form

With reference to coefficient functions `f`, `g`; argument functions `u`, `v`; predefined integration measures `dx`, `ds`, and `dS`; and subdomain notation as introduced in Section 2.1.1. Recall that the predefined UFL measures `dx`, `ds` and `dS` default to `dx(0)`, `ds(0)` and `dS(0)`, respectively, and that the mathematical notation starts counting at 1, while the code starts counting at 0.

Table XII. Overview of common form operators.

Mathematical notation	UFL notation	Description
-	`L = lhs(F)`	Extract terms of arity 2 from `F`
-	`a = rhs(F)`	Extract terms of arity 1 from `F`, multiplied with -1
-	`(a, L) = system(F)`	Extract both lhs and rhs terms from `F`
$a \mapsto a^*$	`a_star = adjoint(a)`	Derive adjoint form of bilinear form `a`
$F(f;\cdot) \mapsto F(g;\cdot)$	`G = replace(F, {f:g})`	Replace coefficient `f` with `g` in `F`
$F(;\cdot) \mapsto F(f;\cdot)$	`M = action(F, f)`	Replace argument function 1 in `F` by `f`
$F(f;\cdot) \mapsto D_f F(f;\cdot)[v]$	`dF = derivative(F, f, v)`	Differentiate `F` w.r.t `f` in direction `v`

With reference to a given form `F` of possibly mixed arity, coefficient functions `f`, `g` and an argument function `v`. Note that all form operators return a new form as the result of the operation; the original form is unchanged.

of all integrals of arity 1 and `system` extracts the tuple of both the afore results; that is, `system(F) = (lhs(F), rhs(F))`. These operators are named by and typically used for extracting 'left-hand' and 'right-hand' sides of variational equations expressed as $F(u,v) = 0$. Next, the `adjoint` operator acts on bilinear forms to return the adjoint form a^*: $a(;u,v) \mapsto a^*(;u,v) = a(;v,u)$. The `replace` operator returns a version of a given form in which given functions are replaced by other given functions. The `action` operator can be viewed as special case of replace: the argument function with the lowest count is replaced by a given function.

The Gâteaux derivative (9) is provided by the operator `derivative`, and is possibly the most important form operator. Table XIII provides some usage examples of this operator. With reference to Table XIII, we observe that forms can be differentiated with respect to coefficients separately (L_1 and L_2) or with respect to simultaneous variation of multiple coefficients (L_3). Note that in the latter case, the result becomes a linear form with an argument function in the mixed space. Differentiation with respect to a single component in a vector-valued coefficient is also supported (L_4).

The high-level operations on forms provided by UFL can enable the expression of algorithms at a higher abstraction level than what is possible or practical with a traditional implementation. Some concrete examples using UFL and operations on forms can be found in [Farrell et al. 2012; Rognes and Logg 2012].

Table XIII. Example derivative calls.

Mathematical operation	UFL notation
$L_1(u,p;v) = \frac{d}{d\tau}\left[M(u+\tau v,p)\right]_{\tau=0}$	`L1 = derivative(M, u, v)`
$L_2(u,p;q) = \frac{d}{d\tau}\left[M(u,p+\tau q)\right]_{\tau=0}$	`L2 = derivative(M, p, q)`
$L_3(u,p;w) = \frac{d}{d\tau}\left[M(u+\tau v,p+\tau q)\right]_{\tau=0}$	`L3 = derivative(M, (u, p), w)`
$L_4(u,p;s) = \frac{d}{d\tau}\left[M(u+\tau s e_y,p)\right]_{\tau=0}$	`L4 = derivative(M, u[1], s)`

With reference to a functional $M : W = V \times Q \to \mathbb{R}$ for a vector-valued function space V with a scalar subspace V_1, and a scalar-valued function space Q; coefficient functions $u \in V$, $p \in Q$; and argument functions $v \in V$, $q \in Q$, $s \in V_1$ and $w \in W$.

3.4. The `.ufl` file format

UFL may be integrated into a problem solving environment in Python or written in `.ufl` files and compiled offline for use in a problem solving environment in a compiled language such as C++. The `.ufl` file format is simple: the file is interpreted by Python with the full `ufl` namespace imported, and forms and elements are extracted by inspecting the resulting namespace. In a typical `.ufl` file, only minor parts of the Python language are used, although the full language is available. It may be convenient to use the Python `def` statement to define reusable functions within a `.ufl` file. By default, forms with the names `a`, `L`, `M`, `F` or `J` are exported (i.e. compiled by the form compiler). The convention is that `a` and `L` define bilinear and linear forms for a linear equation, `M` is a functional, and `F` and `J` define a nonlinear residual form and its Jacobian. Forms with any names may be exported by defining a list `forms = [form0, form1]`. By default, the elements that are referenced by forms are compiled, but elements may also be exported without being used by defining a list `elements = [element0, element1]`. Note that meshes and values of coefficients are handled by accompanying problem solving environment (such as DOLFIN) and `Coefficient` and `Constant` instances in `.ufl` files are purely symbolic.

4. EXAMPLES

We now present a collection of complete examples illustrating the specification, in UFL, of the finite element discretizations of a number of partial differential equations. We have chosen problems that compactly illustrate particular features of UFL. We stress, however, that UFL is not limited to simple equations. On the contrary, the benefits of using UFL are the greatest for complicated, non-standard equations that cannot be easily or quickly solved using conventional libraries. Computational results produced using UFL have been published in many works, covering a vast range of fields and problem complexity, by third-parties and by the developers of UFL, including [Abert et al. 2012; Arnold et al. 2012; Brandenburg et al. 2012; Brunner et al. 2012; Funke and Farrell 2013; Hake et al. 2012; Labeur and Wells 2012; Logg et al. 2012a; Maraldi et al. 2011; Mortensen et al. 2011; Rosseel and Wells 2012; Wells 2011] to mention but a few.

4.1. Poisson equation

As a first example, we consider the Poisson equation and its discretization using the standard H^1-conforming formulation, a L^2-conforming formulation and a mixed $H(\mathrm{div})/L^2$-conforming formulation. The Poisson equation with boundary conditions is given by

$$-\operatorname{div}(\kappa \operatorname{grad} u) = f \quad \text{in } \Omega, \qquad u = u_0 \quad \text{on } \Gamma_{\mathrm{D}}, \qquad -\kappa \partial_n u = g \quad \text{on } \Gamma_{\mathrm{N}}, \tag{20}$$

where $\Gamma_{\mathrm{D}} \cup \Gamma_{\mathrm{N}} = \partial\Omega$ is a partitioning of the boundary of Ω into Dirichlet and Neumann boundaries, respectively.

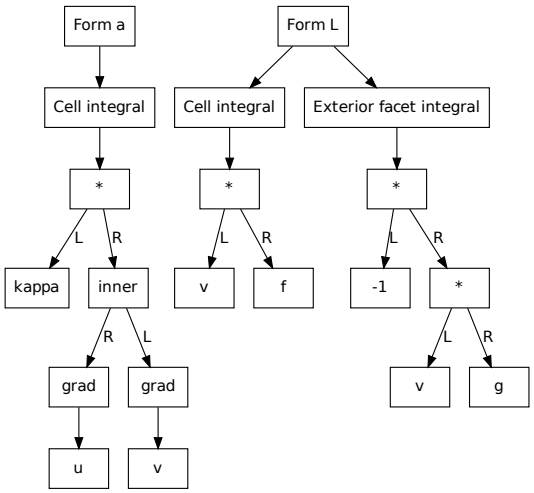

Fig. 1. Expression trees for the H^1 discretization of the Poisson equation (21).

4.1.1. H^1-conforming discretization. The standard H^1-conforming finite element discretization of (20) reads: find $u \in V_h$ such that

$$a(u, v) \equiv \int_\Omega \kappa \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x - \int_{\Gamma_\mathrm{N}} g v \, \mathrm{d}s \equiv L(v) \tag{21}$$

for all $v \in \hat{V}_h$, where V_h is a continuous piecewise polynomial trial space incorporating the Dirichlet boundary conditions on Γ_D and $\hat{V}_h$ is a continuous piecewise polynomial test space with zero trace on Γ_D. We note that as a result of the zero trace of the test function v on Γ_D, the boundary integral in (21) may be expressed as an integral over the entire boundary $\partial\Omega$. A complete specification of the variational problem (21) in UFL is included below. The resulting expression trees for the bilinear and linear forms are presented in Figure 1.

UFL code

```
element = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)

f = Coefficient(element)
g = Coefficient(element)
kappa = Coefficient(element)

a = kappa*inner(grad(u), grad(v))*dx
L = f*v*dx - g*v*ds
```

4.1.2. L^2-conforming discretization. For the L^2 discretization of the Poisson equation, the standard discontinuous Galerkin/interior penalty formulation [Arnold 1982; Ølgaard

et al. 2008] reads: find $u \in V_h$ such that

$$\int_\Omega \kappa \operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x - \int_{\Gamma_\mathrm{D}} \kappa \partial_n u v \,\mathrm{d}s - \int_{\Gamma_\mathrm{D}} \kappa \partial_n v u \,\mathrm{d}s + \int_{\Gamma_\mathrm{D}} \frac{\gamma\kappa}{h} uv \,\mathrm{d}s$$
$$+ \sum_{F \in \mathcal{F}^0} \left(- \int_F \langle \kappa \operatorname{grad} u \rangle \cdot [\![v]\!]_n \,\mathrm{d}s - \int_F \langle \kappa \operatorname{grad} v \rangle \cdot [\![u]\!]_n \,\mathrm{d}s + \int_F \frac{\gamma \langle \kappa \rangle}{\langle h \rangle} [\![u]\!] [\![v]\!] \,\mathrm{d}s \right)$$
$$= \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_\mathrm{N}} g v \,\mathrm{d}s - \int_{\Gamma_\mathrm{D}} \partial_n u_0 v \,\mathrm{d}s - \int_{\Gamma_\mathrm{D}} \partial_n v u_0 \,\mathrm{d}s + \int_{\Gamma_\mathrm{D}} \frac{\gamma\kappa}{h} u_0 v \,\mathrm{d}s \tag{22}$$

for all $v \in V_h$, where $[\![v]\!]$, $[\![v]\!]_n$ and $\langle v \rangle$ are the standard jump, normal jump and average operators (see Section 3.2.7).

The corresponding implementation in UFL is shown below. The relevant operators for the specification of discontinuous Galerkin methods are provided by the operators `jump()` and `avg()`. We also show in Figures 2 and 3 the expression tree for the bilinear form of the L^2-discretization, which is notably more complex than for the H^1-discretization.

UFL code

```
element = FiniteElement("Discontinuous Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)

f = Coefficient(element)
g = Coefficient(element)
kappa = Coefficient(element)
u0 = Coefficient(element)

h = 2*triangle.circumradius
n = triangle.n

gamma = 4

a = kappa*dot(grad(u), grad(v))*dx \
  - dot(kappa*grad(u), v*n)*ds(0) \
  - dot(kappa*grad(v), u*n)*ds(0) \
  + (gamma*kappa/h)*u*v*ds(0) \
  - dot(avg(kappa*grad(u)), jump(v, n))*dS \
  - dot(avg(kappa*grad(v)), jump(u, n))*dS \
  + (gamma*avg(kappa)/avg(h))*jump(u)*jump(v)*dS

L = f*v*dx - g*v*ds(1) \
  - dot(kappa*grad(u0), v*n)*ds(0) - dot(kappa*grad(v), u0*n)*ds(0) \
  + (gamma*kappa/h)*u0*v*ds(0)
```

4.1.3. $H(\mathrm{div})/L^2$-conforming discretization. Finally, we consider the discretization of the Poisson equation with a mixed formulation, where the second-order PDE in (20) is replaced by a system of first-order equations:

$$\operatorname{div} \sigma = f \quad \text{in } \Omega, \qquad \sigma + \kappa \operatorname{grad} u = 0 \quad \text{in } \Omega, \qquad u = u_0 \quad \text{on } \Gamma_\mathrm{D}, \qquad \sigma \cdot n = g \quad \text{on } \Gamma_\mathrm{N}. \tag{23}$$

A direct discretization of the system (23) with, say, continuous piecewise linear elements is unstable. Instead a suitable pair of finite element spaces must be used for σ and u. An example of such a pair of elements is the $H(\mathrm{div})$-conforming BDM (Brezzi–Douglas–Marini) element [Brezzi et al. 1985] of the first degree for σ and discontinu-

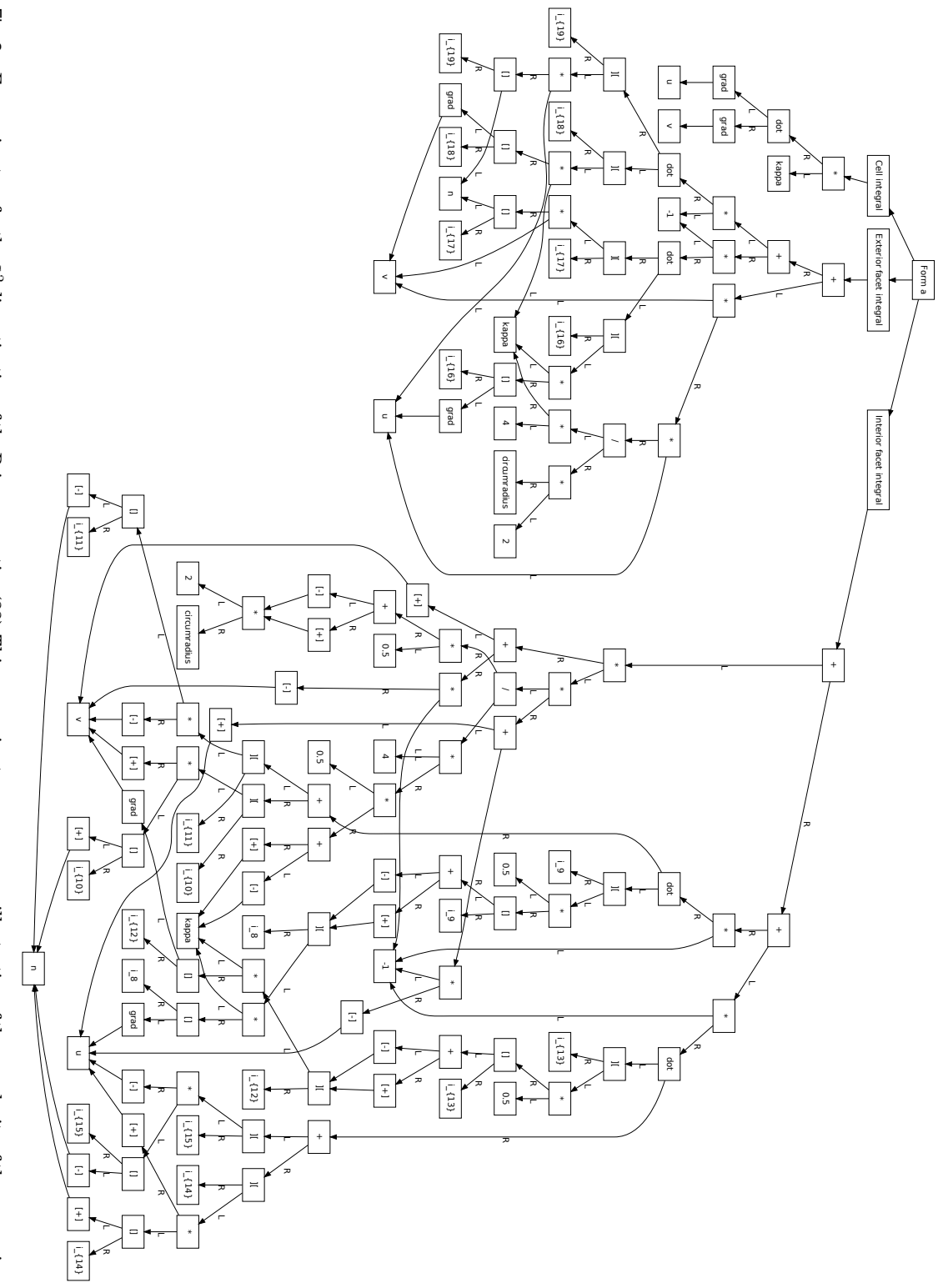

Fig. 2. Expression tree for the L^2 discretization of the Poisson equation (22). This expression tree serves as an illustration of the complexity of the expression tree already for a moderately simple formulation like the L^2 discretization of the Poisson equation. A detail of this expression tree is plotted in Figure 3.

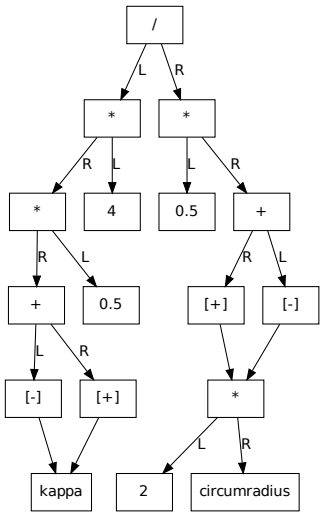

Fig. 3. Detail of the expression tree of Figure 2 for the L^2 discretization of the Poisson equation (22). The expression tree represents the expression `gamma*avg(kappa)/avg(h)`.

ous piecewise constants for u. Multiplying the two differential equations in (23) with test functions τ and v, respectively, integrating by parts and summing the variational equations, we obtain the following variational problem: find $(\sigma, u) \in V_h = V_h^\sigma \times V_h^u$ such that

$$\int_\Omega (\operatorname{div} \sigma) v \,\mathrm{d}x + \int_\Omega \sigma \cdot \tau \,\mathrm{d}x - \int_\Omega u \operatorname{div}(\kappa\tau) \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_D} \kappa u_0 \tau \cdot n \,\mathrm{d}s \qquad (24)$$

for all $(\tau, v) \in \hat{V}_h$. Note that the Dirichlet condition $u = u_0$ on Γ_D is a *natural* boundary condition in the mixed formulation; that is, it is naturally imposed as a weak boundary condition as part of the variational problem. On the other hand, the Neumann boundary condition $\sigma \cdot n = g$ must be imposed as an essential boundary condition as part of the solution space V_h. The corresponding specification in UFL is given below.

UFL code

```
1  BDM = FiniteElement("BDM", triangle, 1)
2  DG  = FiniteElement("DG", triangle, 0)
3  CG  = FiniteElement("CG", triangle, 1)
4  element = BDM * DG
5
6  (sigma, u) = TrialFunctions(element)
7  (tau, v) = TestFunctions(element)
8
9  f = Coefficient(CG)
10 kappa = Coefficient(CG)
11 u0 = Coefficient(DG)
12
13 n = triangle.n
14
15 a = div(sigma)*v*dx + dot(sigma, tau)*dx - u*div(kappa*tau)*dx
16 L = f*v*dx - kappa*u0*dot(tau, n)*ds
```

4.2. Stokes equations

As a second example of a mixed problem, we consider the Stokes equations given by

$$-\Delta u + \operatorname{grad} p = f \quad \text{in } \Omega, \qquad \operatorname{div} u = 0 \quad \text{in } \Omega, \tag{25}$$

for a velocity u and a pressure p, together with a suitable set of boundary conditions. The first equation is multiplied with a test function v, the second equation is multiplied with a test function q, and after integration by parts, the resulting mixed variational problems is: find $(u, p) \in V_h$ such that

$$\int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x - \int_\Omega (\operatorname{div} v) p \,\mathrm{d}x + \int_\Omega (\operatorname{div} u) q \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x. \tag{26}$$

The corresponding specification in UFL using an inf–sup stable $[P_2]^d$–P_1 discretization (Taylor–Hood element [Taylor and Hood 1973]) is shown below.

UFL code

```
P2 = VectorElement("Lagrange", triangle, 2)
P1 = FiniteElement("Lagrange", triangle, 1)
TH = P2 * P1

(u, p) = TrialFunctions(TH)
(v, q) = TestFunctions(TH)

f = Coefficient(P2)

a = inner(grad(u), grad(v))*dx - div(v)*p*dx + div(u)*q*dx
L = dot(f, v)*dx
```

4.3. Neo-Hookean hyperelastic model

We consider next a hyperelastic problem posed in terms of the minimization of potential energy. For a body $\Omega \subset \mathbb{R}^d$ in a reference configuration, where $1 \leq d \leq 3$, the total potential energy Π reads

$$\Pi(v) = \int_\Omega \psi(v) \,\mathrm{d}x - \int_\Omega B \cdot v \,\mathrm{d}x - \int_\Gamma T \cdot v \,\mathrm{d}s, \tag{27}$$

where ψ is the stored energy density, v is the displacement field, B is the nominal body force and T is the nominal traction on the domain boundary $\Gamma = \partial\Omega$. For the compressible neo-Hookean model, the strain energy density reads

$$\psi(v) = \frac{\mu}{2}(I_c - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2, \tag{28}$$

where J is the determinant of the deformation gradient $F = I + \operatorname{grad} u$ and I_c is the trace of the right Cauchy–Green tensor $C = F^T F$. Solutions u to the hyperelastic problem minimize (27):

$$u = \operatorname*{argmin}_{v \in V} \Pi(v). \tag{29}$$

The classical variational approach to solving (29) involves taking the first variation/Gâteaux derivative of (27) with respect to v and setting this equal to zero for all v, which in turn can be solved using Newton's method by taking a second Gâteaux derivative to yield the Jacobian. The corresponding specification in UFL for such an approach is shown below.

UFL code

```
# Function spaces
element = VectorElement("Lagrange", tetrahedron, 1)

# Trial and test functions
du = TrialFunction(element)      # Incremental displacement
v  = TestFunction(element)       # Test function

# Functions
u = Coefficient(element)         # Displacement from previous iteration
B = Coefficient(element)         # Body force per unit volume
T = Coefficient(element)         # Traction force on the boundary

# Kinematics
I = Identity(element.cell().d)   # Identity tensor
F = I + grad(u)                  # Deformation gradient
C = F.T*F                        # Right Cauchy-Green tensor

# Invariants of deformation tensors
Ic, J = tr(C), det(F)

# Elasticity parameters
mu     = Constant(tetrahedron)
lmbda = Constant(tetrahedron)

# Stored strain energy density (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2

# Total potential energy
Pi = psi*dx - inner(B, u)*dx - inner(T, u)*ds

# First variation of Pi (directional derivative about u in the direction
    of v)
F = derivative(Pi, u, v)

# Compute Jacobian of F
J = derivative(F, u, du)
```

A complete solver using precisely the above formulation is distributed as a demo program as part of DOLFIN [DOLFIN 2012; Logg and Wells 2010].

4.4. Constrained optimization

As a final example, we consider a PDE constrained optimization problem. We wish to determine the control variable p that minimizes the cost functional

$$J(u,p) = \int_\Omega \frac{1}{2}(u-\bar{u})^2 \,\mathrm{d}x + \int_\Omega \frac{1}{2}\alpha p^2 \,\mathrm{d}x, \tag{30}$$

where $\bar{u}$ is a given target function, α is a regularization parameter and u is the state, constrained by the variational problem

$$a(u,v) \equiv \int_\Omega uv + \mathrm{grad}\, u \cdot \mathrm{grad}\, v \,\mathrm{d}x = \int_\Omega pv \,\mathrm{d}x \equiv b(p;v), \tag{31}$$

which should hold for all test functions v in some suitable test space.

A way to solve the optimization problem is to define the Lagrangian functional $\mathcal{L}$ as the sum of the cost functional and the weak constraint, in which v plays the role of a Lagrange multiplier:

$$\mathcal{L}(u,p,v) = J(u,p) + a(u,v) - b(p;v). \tag{32}$$

变分元符号的一种写法

The solution of the optimization problem can be found by seeking the stationary points of the Lagrangian:

$$D_u\mathcal{L}(u,p,v)[\delta u] = 0 \qquad \forall\, \delta u, \tag{33}$$
$$D_p\mathcal{L}(u,p,v)[\delta p] = 0 \qquad \forall\, \delta p, \tag{34}$$
$$D_v\mathcal{L}(u,p,v)[\delta v] = 0 \qquad \forall\, \delta v, \tag{35}$$

which in this case is a linear system of equations. The corresponding implementation in UFL is shown below.

UFL code

```
# Define mixed function space for all variables
V = FiniteElement("Lagrange", triangle, 1)
W = MixedElement(V, V, V)

# Define coefficients
w = Coefficient(W)      # mixed function with all unknowns
u = w[0]                # state
p = w[1]                # control
v = w[2]                # Lagrange multiplier
alpha = Coefficient(V)  # regularization parameter
ubar = Coefficient(V)   # observation
pbar = Coefficient(V)   # analytic control

# Define forms
def J(u, p): return 0.5*(u - ubar)**2*dx + 0.5*alpha*p**2*dx
def a(u, v): return (u*v + dot(grad(u), grad(v)))*dx
def b(p, v): return p*v*dx

# Define Lagrangian
L = J(u, p) + a(u, v) - b(p, v)

# Differentiate
F = derivative(L, w)
dF = derivative(F, w)
mF = -F

# Define some norms
L2p = 0.5*(p-pbar)**2*dx
L2u = 0.5*(u-ubar)**2*dx
J = J(u, p)

# Export forms
forms = [mF, dF, J, L2p, L2u]
```

5. REPRESENTATION OF EXPRESSIONS

UFL is a collection of value types and operators. These types and operators have been presented in the preceding sections from a user perspective, with a focus on their mathematical definitions. To discuss the representation and algorithms in the symbolic framework underlying the language implementation, we will now pursue a more abstract approach.

5.1. Abstract categorization of expression types

As a domain specific language embedded in Python, UFL does not have a formal grammar of its own. However, it is still useful to compactly categorize the various types of expressions before proceeding. An expression e can be either a terminal expression t, or an operator o applied to one or more other expressions. A terminal expression t can be categorized in one of the groups:

i Multi-index.
l Literal constant.
g Geometric quantity.
f Function. Further classified as:
  c Coefficient function.
  a Argument function.

An operator type o can be categorized in one of the following groups:

I Indexing related operators, which manipulate the indices and shapes of expressions. These operations do not introduce any computation.
A Basic arithmetic operators, including the operators `+`, `-`, `*` and `/`.
F Nonlinear scalar real functions, such as $\ln(x)$, $\exp(x)$, $\sin(x)$ and $\arccos(x)$.
T Tensor algebraic operators. These are convenience operators acting on nonscalar expressions, such as the dot, inner, and outer products, and the transpose, determinant or deviatoric part.
D Differential operators, including both spatial derivatives, derivatives with respect to expressions annotated as variables and directional derivatives with respect to coefficient functions.
R Restrictions of functions to cells and related operators such as the jump and average between cells.
B Boolean operators, including $=$, $\neq$, $<$, $>$, $\leq$, $\geq$, $\wedge$, $\vee$, $\neg$. These can only be used within a conditional operator.
C The conditional operator: the ternary operation "f if condition else g". This is the only explicit branching instruction, not counting restrictions.

With these definitions, the following diagram gives a compact semi-formal overview of the types and operators that an expression e can be recursively built from:

$$
\begin{aligned}
t &= [i|l|g|c|a], \\
o &= [I|A|F|T|D|R|B|C], \\
e &= [t|o(e_1, \ldots, e_n)].
\end{aligned}
$$

As before, t represents any terminal expression, while o represents any operator. If an expression e is an application of an operator, we say that it *depends* on the expressions $e_1, \ldots, e_n$. Dependencies between expressions are, by construction, one-way relations and can therefore be viewed as the edges of a directed acyclic graph (DAG), where each vertex is itself an expression e_i. Each graph vertex has an associated type, which is either a terminal type or operator type. Vertices representing a terminal expression may have additional descriptive data associated with them, while vertices representing non-terminal subexpressions are fully identified by the type and sequence of child vertices. We note here that the ordering of child vertices (dependencies) is important (for nonsymmetric operators).

5.2. Representation of directed acyclic graphs

From an implementation viewpoint, a DAG can be represented by link-based or list-based data structures. In a link-based data structure, each vertex of the graph is represented by a single typed object. Objects of operator types store references to the objects representing their operands. These references are the edges of the DAG. Objects of terminal types may store additional data. This is the natural DAG representation for a symbolic library while an expression is being built, and is the way UFL expressions are represented most of the time. When a UFL operator is evaluated in Python, the

result is a new object[4] representing the operator. Thus, both the time and storage cost of building an expression is $O(n)$ in the number n of operators applied. Since each operator invocation leads to a new graph vertex without any global knowledge available, some duplicate expressions may occur in this DAG representation.

A list-based data structure for an expression DAG can be constructed from the link-based data structure, if needed. In this data structure, references to each unique subexpression are placed in a topologically sorted vertex list and all edges (dependencies) are stored in another list as a tuple of vertex list indices. While the vertex list is constructed, duplicate expressions with the exact same symbolic representation are automatically mapped to the same vertex. This mapping can be achieved through $O(1)$ insertion into a hash map, which retains the overall $O(n)$ performance of building this data structure. The main advantages of the list-based graph representation are efficient ordered iteration over all vertices and easy access to dependency information. This is beneficial for some algorithms.

5.3. Expression type class hierarchies

Each DAG vertex, or expression object, is an instance of a subclass of `Expr`. The type of terminal or operator is determined by the subclass, from a class hierarchy which is divided into `Terminal` and `Operator` subtypes. Concrete `Operator` subclasses store references to the objects representing their operands. Since a non-terminal expression is uniquely determined by its type and operands, any other data stored by these classes is purely for performance or convenience reasons. Concrete `Terminal` subclasses, however, can take any necessary auxiliary data in their constructor. This data must still be immutable, such that any expression object is hashable. Most operator overloading is applied directly to the `Expr` class. For example, adding any two expressions will result in a new instance of a `Sum`. All expression classes must overload a set of basic methods. One such method is `operands()`, which returns a tuple of expression objects for the operands of an `Operator`, or the empty tuple for a `Terminal` object. We refer to the UFL source code [UFL 2012] for further internal implementation details.

5.4. Representation of indexed expressions

In some symbolic libraries, expressions with shapes or indices are built on top of an otherwise mainly scalar framework. In contrast, UFL considers shapes and indices as an integral part of any expression. To support this, expressions provide three methods, namely, `shape()` which returns a tuple of positive integers representing the value shape of the expression, `free_indices()` which returns an unordered tuple of `Index` objects for each free-index in the expression and `index_dimensions()`, which returns a mapping from `Index` objects to the dimensions they implicitly range over. Together, these methods provide a rich and flexible description of the value shape and free-index set. These properties generalize to arbitrarily shaped expressions with arbitrary sets of free-indices, and are accessible for any expression regardless of type.

In traditional programming languages, the indexing operator `[]` usually extracts a value from a container. In contrast, in a symbolic environment, the result may be an object representing the operation of extracting a component. When indexing with free-indices, the former type of extraction is not possible since the index represents a range of values. Instead, when indexing a tensor-valued expression in UFL, the result is represented as an object of the `Indexed` class, with the original expression and a `MultiIndex` object as operands. A `MultiIndex` object[5] represents a sequence of `Index`

[4]Sometimes more than one new object, but the number of auxiliary objects is always bounded by a small constant.

[5]The type system in UFL is fairly simple, and `MultiIndex` is one of the few exceptions in the `Expr` type hierarchy that does not represent a tensor-valued expression.

and `FixedIndex` objects. The opposite operation from indexing, mapping an expression with free-indices to be seen as a tensor-valued expression, cf. Section 3.2.2 and in particular Table V, is represented with the `ComponentTensor` class, which similarly has the original expression as well as a `MultiIndex` as operands. Expressions which contain layers of mapping back-and-forth between index and tensor notation may appear complex, but a form compiler can reduce this complexity during the translation to low-level code.

The implicit summation notation in UFL is applied early in the application of the `*` operator and the derivative operator `.dx(i)`. Subsequently the summation is represented as explicit sums over free-indices. This way, the implicit summation rules need no special consideration in most algorithms. For example, the sum $u_i v_i \equiv \sum_i u_i v_i$ is expressed in UFL code as `u[i]*v[i]`, but is represented in the DAG as `IndexSum(Product(u[i], v[i]), (i,))`[6]. A `Product` object can be constructed directly in algorithms where implicit summation is not the wanted behavior. When interpreting an `IndexSum` object in symbolic algorithms, the range of the sum is defined by the underlying summand expression; this is possible as any expression in UFL knows its own shape, free-indices and dimensions of its free-indices.

5.5. Simplification of expressions

Automatic simplification of expressions is a central design issue for any symbolic framework. At one end of the design spectrum are conservative frameworks that preserve expressions exactly as written by the user, analogous to an abstract syntax tree of the program input to a compiler. At the other end of the spectrum are frameworks where all expressions are transformed to a canonical form at construction time. The design of UFL is guided by the intention to be the front end to multiple form compilers, and is therefore fairly conservative. If a form compiler does no rewriting, the generated code should transparently resemble the UFL expressions as authored by the end-user. However, UFL is also a collection of symbolic algorithms, and the performance and memory footprint of such algorithms can be significantly improved by certain automatic simplifications of expressions at construction time, if such simplifications reduce symbolic expression growth.

Because form compilers may employ different expression rewriting strategies, we wish to avoid doing any simplifications that might remove information a form compiler could make use of. A canonical sorting of two sum operands eases the detection of common subexpressions, such as in the expression rewriting $(b+a)(a+b) \to (a+b)(a+b)$. However, a canonical sorting of more than two sum operands may hide common subexpressions, such as in the rewriting $((a+c)+b)(a+c) \to (a+b+c)(a+c)$. Therefore, sums and products are represented in UFL as binary operators (as opposed to storing lists of terms or factors), with their two operands sorted canonically. Even more importantly, we avoid any symbolic rewriting that can lead to numerically unstable floating point expressions in generated code. An example of such unsafe operations is the expansion of a factored polynomial $(a-b)(a-b) \to a^2 - 2ab + b^2$, which becomes numerically unstable in inexact floating point arithmetic.

The performance of the symbolic algorithms and the form compilation processes poses a final limitation on the type of automatic simplifications we may apply. It is crucial that the overall form compilation process can be designed to have an asymptotic cost of $O(n)$ in time and memory usage[7], with n a measure of the length of the integrand expression. Because simplification of expressions at construction time is per-

[6]Here, the representations of `u[i], v[i], (i,)` are simplified for compact presentation.

[7]Form compilers are free to apply more expensive strategies, but UFL must not render efficient algorithms impossible.

formed once for each expression object, the cost of applying the simplification must be a local operation independent of the size of the operand expressions. Thus, automatic simplifications which would involve traversing the entire subexpression DAG for analysis or rewriting are never attempted.

The following small set of safe and local simplifications is applied consistently when constructing expressions:

Multiply by one. $1x \to x$. A simplification which keeps one operand intact and throws away the other one is safe.
Add zero. $0 + x \to x$.
Multiply by zero. $0x \to 0$. To avoid losing the tensor properties of x, we annotate the representation of 0 with the same properties. Therefore we have a special zero representation with shape and free-indices.
Constant folding. $f(l_1, l_2) \to l_3$. Here l_1, l_2 and l_3 are literal constants and f is a function or operator that can be computed numerically by UFL. This happens recursively and so constant scalar expressions are effectively folded to a single literal constant.
Canceling indexing. $A_\alpha E^\alpha \to A$. The mappings between tensor-valued and indexed expression representations cancel when the same multi-index α is used in both operations. If the inner and outer multi-indices are not equal, this canceling operation would require rewriting the representation of A, which is not a local operation, and therefore not invoked.

Note that some of these operations will occur frequently during automatic differentiation. For example, consider the derivative $d(fg)/df$, where f and g are independent functions:

$$\frac{d(fg)}{df} = \frac{df}{df}g + f\frac{dg}{df} = 1g + f0 \to g + 0 \to g. \tag{36}$$

Similarly, during some symbolic algorithms, tensor-valued subexpressions are indexed to simplify computation and later mapped back to tensor-valued expressions. This process leads to superfluous indexing patterns which causes the DAG to grow needlessly. Applying the indexing cancellations helps in avoiding this DAG growth.

6. ALGORITHMS

The UFL implementation contains a collection of basic algorithms for analysis and transformation of expressions. These algorithms include optimized Python generators for easy iteration over expression nodes in pre- or post-ordering and iteration over terminal expressions or, more generally, expressions of a particular type. In the following, some of the core algorithms and building blocks for algorithms are explained. Particular emphasis is placed on the differentiation algorithm. To avoid verbose technical details in the discussions of symbolic algorithms, we first define some mathematical notation to support abstract algorithm descriptions. For implementation details, we refer to the UFL source code [UFL 2012].

6.1. Evaluation algorithm

Assume that an expression e is represented by a DAG with m terminal and n nonterminal subexpressions, recalling that each subexpression is a DAG vertex. Denote the terminal subexpressions by e_i, for $i = 1, 2, \ldots, m$ and the non-terminal subexpressions by e_i, for $i = m + 1, m + 2, \ldots, m + n$. For each e_i, let $I^i \equiv \langle I^i_j \rangle_{j=1}^{p_i}$ be a sequence of p_i integer labels referring to the operand expressions of e_i. Note that for each terminal expression, this sequence is empty. Moreover, we require the labels to fulfill a

topological ordering such that

$$I_j^i < i, \qquad j = 1, \ldots, p_i, \qquad i = 1, \ldots, m+n. \tag{37}$$

We thus have $k < i$ whenever e_i depends on e_k (directly or indirectly). Equivalently e_i is independent of any subexpression e_k for $k > i$.

We can now formulate two versions of an algorithm for evaluating $e \equiv e_{m+n}$. Note that these algorithms are merely abstract tools for describing the mathematical structure of algorithms that follow. For each specific algorithm, we require an evaluation operator E where $v_i = E(e_i)$ is called the value of e_i. This value can be a floating point value, a new symbolic expression or a generated source code string, depending on the purpose of the algorithm. The implementation of the evaluation operator E will in general depend on the UFL expression type of its argument e, which we denote $\mathrm{type}(e)$.

First, partition the set T of all UFL expression types into disjoint sets of terminal types T_T and operator types T_O, and let E_T and E_O be non-recursive evaluation operators for terminal and non-terminal expressions, respectively. We can then design a simple recursive evaluation algorithm E_R of the form

$$E_R(e_i) = \begin{cases} E_T(e_i), & \text{if } \mathrm{type}(e_i) \in T_T, \\ E_O\left(e_i, \langle E_R(e_{I_j^i})\rangle_{j=1}^{p_i}\right), & \text{if } \mathrm{type}(e_i) \in T_O. \end{cases} \tag{38}$$

The algorithmic structure of (38) assumes that any subexpression can be evaluated given the values of its operands, which is not true for operators which provide a context for the evaluation of their operands. For example, in a derivative evaluation algorithm, each type of derivative operator provides a different differentiation variable which affects the evaluation rules, and similarly in a restriction propagation algorithm, the restricted type provides which side the terminals of the subexpression should be restricted to. We therefore partition T_O further into algorithm-specific disjoint sets T_V and T_C, where T_V includes types of expressions that can be evaluated given the values of its operands as in (38), and T_C includes types of expressions which provide a context for the evaluation of the operands. By defining corresponding evaluation operators E_V and E_C, we can then extend (38) to

$$\tilde{E}_R(e_i) = \begin{cases} E_T(e_i), & \text{if } \mathrm{type}(e_i) \in T_T, \\ E_V\left(e_i, \langle \tilde{E}_R(e_{I_j^i})\rangle_{j=1}^{p_i}\right), & \text{if } \mathrm{type}(e_i) \in T_V, \\ E_C\left(e_i, \langle e_{I_j^i}\rangle_{j=1}^{p_i}\right), & \text{if } \mathrm{type}(e_i) \in T_C. \end{cases} \tag{39}$$

Note that the evaluation operator E_C may make further recursive calls to $\tilde{E}_R$ or a related recursive algorithm, but it is assumed that its operands cannot be pre-evaluated without the provided context. The difference between how E_V and E_C are applied corresponds to a post-order versus pre-order evaluation of the DAG vertices.

The recursive operator E_R defined by (38) can be implemented by traversing the link-based DAG representation with a post-order traversal algorithm, to recursively visit and evaluate child vertices before their parent. The more flexible operator $\tilde{E}_R$ defined by (39) can be implemented similarly but with a mix of post-order and pre-order traversals depending on the visited types. The evaluation of e can then be written simply

$$v = \tilde{E}_R(e) \equiv \tilde{E}_R(e_{m+n}). \tag{40}$$

The simpler recursive operator (38) can also be implemented as a loop over subexpressions in a topological ordering, as shown in Algorithm 1. This non-recursive algorithm can be implemented by first constructing the list-based DAG representation

and then iterating over the vertices. Using the list-based representation and the non-recursive algorithm has the advantage of never visiting a vertex twice, even if it is reachable through multiple paths. We remark that a cache mechanism may, however, remove such duplicate evaluation in the recursive implementation as well. The list-based representation also assigns an integer label to each vertex which can be used to store associated data efficiently in arrays during the algorithm. However, the construction of the list-based DAG representation is not free, and the fixed labeling is a downside in algorithms where new expressions are constructed. In the following exposition, the choice of algorithm structure, recursive or non-recursive, is considered mainly an implementation detail, controlled by performance and convenience considerations and not by functionality.

Algorithm 1 Non-recursive algorithm for evaluation of expressions.

1: **for** $i = 1, \ldots, m$ **do**
2: $\quad v_i = E_T(e_i)$
3: **end for**
4: **for** $i = m+1, \ldots, m+n$ **do**
5: $\quad v_i = E_O(e_i, \langle v_{I_j^i} \rangle_{j=1}^{p_i})$
6: **end for**
7: $v := v_{m+n}$

6.2. Type based function dispatch and the visitor pattern

In an implementation of the evaluation algorithm described in the previous section, the specific evaluation actions must be selected dynamically based on the type of the expression argument. By subclassing the UFL provided class `MultiFunction` and implementing a handler function for each expression type, calls to an instance of this class are dynamically dispatched to the correct handler based on the type of the first argument. If a handler is missing, the closest superclass handler is used instead, which makes it easy to implement default rules for groups of types. An example is shown below.

Python code

```
1 from ufl import triangle
2 from ufl.algorithms import MultiFunction
3 class ExampleAlgorithm(MultiFunction):
4     def terminal(self, e):  return "unhandled terminal type"
5     def zero(self, e):      return repr(e)
6 E = ExampleAlgorithm()
7 print E(triangle.n)   # Prints "unhandled terminal type"
8 print E(0*triangle.n) # Prints "Zero((2,), (), {})"
```

Building on this same dynamic multifunction design and the Visitor pattern [Gamma et al. 1993], the class `Transformer` can be subclassed in the same way to implement many recursive symbolic algorithms following the structure of (38) or (39). Calling upon an object of a `Transformer` subclass to visit an expression will result in a recursive application of type-specific rules to subexpressions. The example below shows a numerical evaluation of a simple expression, using a pure post-order implementation as in (38). Whether to visit expressions post- or pre-order is specified per handler simply by taking visited expressions for child nodes as arguments or not. In more detail, and with reference to the example below, a type will be placed in the T_C set

if the corresponding handler omits the `*values` argument. The `visit` method will then automatically call the handler without first handling the operands of its argument.

Python code

```python
from ufl import triangle
from ufl.algorithms import Transformer
class ExampleAlgorithm(Transformer):
    def cell_volume(self, e):         return 0.5
    def scalar_value(self, e):        return float(e)
    def product(self, e, *values):    return values[0] * values[1]
    def division(self, e, *values):   return values[0] / values[1]
E = ExampleAlgorithm()
print E.visit(3 * triangle.volume / 2) # Prints 0.75
```

6.3. Partial evaluation

Some symbolic algorithms involve modification of subexpressions, and such algorithms share a need to apply an operator to a new sequence of operands. We will designate the notation $\text{type}(e)(\langle f_j \rangle_j)$ to the construction of an operator of the same type as e with the given operands. If the operands are unchanged, the original expression can be reused since all expressions are considered immutable, thus saving memory. This can easily be accomplished for UFL-based algorithms by subclassing the `ReuseTransformer` class. In this case, the algorithm inherits the fallback rules given by

$$E_{\text{reuse}}(e_i) = \begin{cases} e_i, & \text{if } \text{type}(e_i) \in T_T, \\ \text{type}(e_i)\left(\langle E_{\text{self}}(e_{I_j^i})\rangle_{j=1}^{p_i}\right), & \text{if } \text{type}(e_i) \in T_O, \end{cases} \tag{41}$$

where E_{self} refers to E_{reuse} or overridden rules in a subclass. Based on E_{reuse}, algorithms can be written to just modify what they need and let the fallback rules in (41) rebuild the surrounding expression with no additional algorithm-specific code. This allows a very compact implementation for algorithms such as the partial evaluation in which terminal expressions are replaced with other expressions through a given mapping. As an example, consider a partial evaluation algorithm mathematically described by:

$$E_{\text{replace}}(e_i) = \begin{cases} \text{map}(e_i), & \text{if } \text{type}(e_i) \in T_T, \\ \text{delegate to } E_{\text{reuse}}(e_i), & \text{otherwise}, \end{cases} \tag{42}$$

where *delegate to* $E_{\text{reuse}}(e_i)$ represents delegation to inherited rules from the superclass. This can be compactly implemented in UFL as follows:

Python code

```python
from ufl.algorithms import ReuseTransformer

class Replacer(ReuseTransformer):
    def __init__(self, mapping):
        ReuseTransformer.__init__(self)
        self._mapping = mapping # A Python dict
    def terminal(self, e):
        return self._mapping.get(e, e)
```

6.4. Differentiation

The differentiation algorithm in UFL is a two-level algorithm. The two levels are used to handle expressions involving derivatives with respect to different types of vari-

ables. In a first step, a simple outer algorithm is employed to evaluate the innermost derivatives first; that is, the derivative expressions closest to the terminal expressions. This outer algorithm then calls a single-variable differentiation algorithm for each derivative expression visited. This, in turn, allows the inner algorithm to assume that no nested derivatives are encountered. Thus, for instance in the evaluation of $d(c\,\mathrm{grad}(vu))/dv$, an inner algorithm is called first to evaluate $\mathrm{grad}(vu)$, and second to evaluate $d(c(\mathrm{grad}(v)u + v\,\mathrm{grad}(u)))/dv$. Note that the grad operator is kept in the expression DAG after derivative evaluation, but is guaranteed to only apply directly to spatially varying terminal expressions. More sophisticated approaches to nested differentiation have been explored in Karczmarczuk [2001], Pearlmutter and Siskind [2007] and Siskind and Pearlmutter [2008], however we have considered this additional complexity unnecessary for the purpose of UFL.

The inner algorithm handles differentiation of an expression e with respect to a single differentiation variable u. Both e and u may be tensor-valued, but in the following we illustrate using scalars for the sake of clarity. By setting the value in Algorithm 1 to a tuple of the subexpression and its derivative: $v_i = (e_i, de_i/du)$, we obtain the standard Forward-mode Automatic Differentiation algorithm[8] (see [Griewank 1989]). This algorithm can also be written in recursive form as in (38). However, because of a few exceptions specific to differentiation variable types, the actual algorithm in UFL requires the more flexible framework given by (39). For simplicity, we will use $v_i = de_i/du$ to define the evaluation rules below.

Generic differentiation rules are implemented as handler functions in a transformer class corresponding to

$$E_{\mathrm{AD}}(e_i) = \begin{cases} 0, & \text{if } \mathrm{type}(e_i) \in T_T, \\ \sum_{j=1}^{p_i} \frac{\partial e_i}{\partial e_{I_j^i}} E_{\mathrm{AD}}(e_{I_j^i}), & \text{if } \mathrm{type}(e_i) \in T_O. \end{cases} \tag{43}$$

For each type of differentiation variable, the default differentiation rules in E_{AD} are subclassed to encode the dependency of expression types with respect to the differentiation variable.

For spatial derivatives, the full gradient is used to represent the derivatives of functions, giving the evaluation operator:

$$E_{\mathrm{XD}}(e) = \begin{cases} I, & \text{if } e \text{ is the spatial coordinate vector}, \\ 0, & \text{if } e \text{ is a piecewise constant function}, \\ \mathrm{grad}\, e, & \text{if } e \text{ is a non-constant function}, \\ \mathrm{grad}\,\mathrm{grad}\, f, & \text{if } e \text{ is } \mathrm{grad}\, f, \\ \text{delegate to } E_{\mathrm{AD}}(e), & \text{otherwise}. \end{cases} \tag{44}$$

For differentiation with respect to user-annotated variables (see 3.2.5), the operator rules are instead modified as:

$$E_{\mathrm{VD}}(e) = \begin{cases} 1, & \text{if } e \text{ is the variable instance } u, \\ E_{\mathrm{VD}}(f), & \text{if } e \text{ is another variable instance annotating the expression } f, \\ \text{delegate to } E_{\mathrm{AD}}(e), & \text{otherwise}. \end{cases} \tag{45}$$

[8]However, output of the algorithm is a new symbolic UFL expression, and the algorithm is therefore clearly a symbolic differentiation algorithm. In UFL context, the distinction between Automatic Differentiation and Symbolic Differentiation is therefore irrelevant.

Note that in this case the `Variable` type lies in the T_C set, and is thus visited before the annotated expression. Hence, the underlying expression is visited by a further recursive call to $E_{\mathrm{VD}}(f)$ only if the variable e is different from u.

Finally, for directional derivatives of an expression e with respect to a coefficient function u in the direction of v, with possibly user-specified $\partial g/\partial u = h$ for a subexpression g, the rules become:

$$E_{\mathrm{DD}}(e) = \begin{cases} v, & \text{if } e \text{ is the function } u, \\ \operatorname{grad} E_{\mathrm{DD}}(f), & \text{if } e \text{ is } \operatorname{grad} f, \\ hv, & \text{if } e \text{ is the function } g, \\ \text{delegate to } E_{\mathrm{AD}}(e), & \text{otherwise.} \end{cases} \tag{46}$$

Note that in this case, we consider the gradient of a function as a terminal entity, although it is represented as two expression nodes.

To support differentiation with respect to specific components of a mixed function (or a vector-valued function), the same rules can be applied by choosing an appropriate expression for v. As an example, consider the functions

$$u : X \to \mathbb{R}^3, \qquad \hat{u} = (u_1, u_3), \qquad \hat{v} : X \to \mathbb{R}^2. \tag{47}$$

To compute $D_{\hat{u}} M(u)[\hat{v}]$, the differentiation rule for u must yield a vector-valued derivative of the same value shape as u. This is accomplished by padding $\hat{v}$ and using $v = (\hat{v}_1, 0, \hat{v}_2)$ as the direction. That is,

$$D_{\hat{u}} M(u)\,[\hat{v}] = \frac{d}{d\tau}\left[M((u_1 + \tau\hat{v}_1, u_2 + 0\tau, u_3 + \tau\hat{v}_2))\right]_{\tau=0} = D_u M(u)\,[v]\,. \tag{48}$$

This concept of padding to support component-wise derivatives extends to arbitrary tensor-valued functions and functions in mixed element spaces, as well as differentiation of variable components that are part of different functions.

We conclude this section by commenting on the relation between the UFL differentiation algorithms and the algorithms implemented in Sundance [Long et al. 2010]. In the Sundance approach, the derivative is computed numerically on the fly, and the use of BLAS amortizes the cost of expression DAG traversal at run time. However, the traversal cost does increase with the size of the expression DAG. Therefore, Sundance avoids computing the DAG for the derivative. In contrast, as UFL is typically used in combination with code generation tools, we may differentiate and then simplify. This allows us to produce the expression DAG for the derivative and then produce efficient code for it. We can see the symbolic traversal cost as a part of the software build time, and it does not affect the runtime for computing/assembling variational forms.

7. VALIDATION

Several validation steps are performed by UFL at the stage where an operator is applied to an expression. All UFL operator types validate the properties of its operands in various ways to ensure that the expressions are meaningful. Most importantly, each operator validates the operand value shapes and verifies that the use of free-indices is consistent. This type of validation catches common indexing bugs at an early stage. Other examples of validations include: checking that value shapes and indices match when adding expressions; checking value shapes for tensor algebra products; and checking of index ranges for explicit indexing of tensor-valued expressions. Most indexing in UFL expressions uses implicit ranges, which reduces repetition and common sources of errors. When defining a form, an integrand expression must always be scalar-valued without any free-indices. This is also checked. When the form is preprocessed, prior to form compilation, a number of properties are checked such as: all integrals must depend linearly on the same set of argument functions; and in interior

facet integrals all functions must be restricted. The latter in particular forces increased clarity in the formulations.

Testing of symbolic frameworks is hard because every algorithm must be tested with an appropriate selection of expression type combinations to achieve high test coverage. In an attempt to answer to this challenge in UFL, we have used multiple layers of defensive programming with assertions, unit testing and integration testing in other FEniCS components. Static code analysis with PyChecker [PyChecker 2011] was very useful during the main development phase. Algorithms in UFL are sprinkled with assertions to document assumptions and catch any that fail. Unit tests cover many common (and uncommon) combinations of operators and applications of algorithms such as differentiation. In addition to the unit tests in UFL, unit and regression tests in FFC and DOLFIN test the use of UFL for many PDE application examples.

As an example for testing differentiation, a numerical evaluation of the symbolic derivative can be compared with the evaluation of manually derived derivative expressions or reference values. Of particular interest when considering validation of UFL is a set of integration tests in DOLFIN which exploit Green's theorem in 1D or nD for a scalar function $f(x)$ or a vector-valued function $v(x)$, that is

$$\int_a^b f'(x)\,\mathrm{d}x = \big[f(x)\big]_a^b, \quad \int_\Omega \operatorname{div} v\,\mathrm{d}x = \int_{\partial\Omega} v\cdot n\,\mathrm{d}s. \tag{49}$$

This identity is ideal for combined testing of symbolic differentiation, numerical differentiation, and symbolic evaluation, or even higher order derivatives by setting $v = \operatorname{grad} f$ in (49). Exploiting such mathematical identities is key to robust testing of mathematical software, and combining the symbolic and numerical paradigms provides good opportunities for discovering errors.

A software stack such as that provided in the FEniCS Project with (i) a DSL and symbolic framework in UFL, (ii) automated code generation in the form compiler FFC [Logg et al. 2012b], and (iii) library code in the problem solving environment DOLFIN [Logg et al. 2012c], naturally introduces additional complexity to the debugging process if something goes wrong. However, the high abstraction level allows us to check the correctness of end-user programs in various ways. Automated validation of expressions and forms by UFL allows consistency checks and catching of user errors at various levels of abstraction. Last, but not least, UFL has been tested in active use by researchers for more than three years as part of the FEniCS Project.

8. CONCLUSIONS

We have presented the Unified Form Language and shown how the language and its associated algorithms allow compact, readable and efficient specification of mathematical expressions related to variational formulations of partial differential equations. We have presented both high-level and detailed views of UFL to communicate its practical use and to provide developers and technical users a firm grounding in the design principles of UFL for understanding and building upon UFL.

UFL is a stand-alone Python module that has been extensively used as part of the FEniCS software pipeline since 2009. The UFL functionality has been crucial in enabling advanced automated finite element algorithms in the FEniCS context, especially for complicated coupled systems of equations and for problems for which automatic differentiation dramatically reduced the burden on the application developer. UFL has also proven to be extensible beyond the core implementation, as exemplified by Nikbakht and Wells [2009] and Massing et al. [2012] in the context of extended finite element methods, and Markall et al. [2012] in relation to code generation for different architectures. UFL is an actively developed project and continues to further extend the power and expressiveness of the language.

ACKNOWLEDGMENTS

The authors wish to thank Kent-Andre Mardal and Johannes Ring for their contributions to UFL, and Pearu Peterson for discussions about symbolic representations during the initial design phase.

REFERENCES

C. Abert, L. Exl, G. Selke, A. Drews, and T. Schrefl. Numerical methods for the stray-field calculation: A comparison of recently developed algorithms. *Journal of Magnetism and Magnetic Materials*, 2012.

M. S. Alnæs. *A compiler framework for automatic linearization and efficient discretization of nonlinear partial differential equations*. PhD thesis, University of Oslo, 2009.

M. S. Alnæs and K.-A. Mardal. On the efficiency of symbolic computations combined with code generation for finite element methods. *ACM Trans. Math. Software*, 37(1), 2010.

M. S. Alnæs and K.-A. Mardal. SyFi and SFC: Symbolic finite elements and form compilation. In A. Logg, K.-A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 15. Springer, 2012.

M. S. Alnæs, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen. Unified framework for finite element assembly. *Int. J. Computational Science and Engineering*, 4(4):231–244, 2009. URL `http://dx.doi.org/10.1504/IJCSE.2009.029160`.

M. S. Alnæs, A. Logg, and K.-A. Mardal. UFC: a finite element code generation interface. In A. Logg, K.-A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 16. Springer, 2012.

D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM J. Numer. Anal.*, 19(4):742–760, 1982.

D. N. Arnold, F. Brezzi, and M. Fortin. A stable finite element for the Stokes equations. *Calcolo*, 21(4):337–344, 1984. URL `http://dx.doi.org/10.1007/BF02576171`.

D. N. Arnold, R. S. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numer.*, 15:1–155, 2006.

D. N. Arnold, R. S. Falk, J. Guzmán, and G. Tsogtgerel. On the consistency of the combinatorial codifferential. *arXiv preprint arXiv:1212.4472*, 2012.

B. Bagheri and L. R. Scott. About Analysa. Technical Report TR-2004-09, Departments of Computer Science, University of Chicago, 2004. URL `http://www.cs.uchicago.edu/research/publications/techreports/TR-2004-09`.

G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of *ab Initio* quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, 2005. URL `http://dx.doi.org/10.1109/JPROC.2004.840311`.

Boost. Boost C++ libraries, 2012. URL `http://www.boost.org/`.

C. Brandenburg, F. Lindemann, M. Ulbrich, and S. Ulbrich. Advanced numerical methods for pde constrained optimization with application to optimal design in navier stokes flow. In *Constrained Optimization and Optimal Control for Partial Differential Equations*, pages 257–275. Springer, 2012.

F. Brezzi, J. Douglas, and L. D. Marini. Two families of mixed elements for second order elliptic problems. *Numer. Math.*, 47:217–235, 1985.

P. Brunner, C. Clason, M. Freiberger, and H. Scharfetter. A deterministic approach to the adapted optode placement for illumination of highly scattering tissue. *Biomedical optics express*, 3(7):1732, 2012.

DOLFIN. DOLFIN library, 2012. URL http://launchpad.net/dolfin.

P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. 2012. URL http://arxiv.org/abs/1204.5577.

S. W. Funke and P. E. Farrell. A framework for automated pde-constrained optimisation. *CoRR*, abs/1302.3894, 2013.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *ECOOP93–Object-Oriented Programming*, pages 406–431, 1993.

A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

J. Hake, A. G. Edwards, Z. Yu, P. M. Kekenes-Huskey, A. P. Michailova, J. A. McCammon, M. J. Holst, M. Hoshijima, and A. D. McCulloch. Modelling cardiac calcium sparks in a three-dimensional reconstruction of a calcium release unit. *The Journal of Physiology*, 590(18):4403–4422, 2012.

P. Hudak. Building domain-specific embedded languages. *ACM computing surveys*, 28 (4es):196, 1996.

J. Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, 2001. URL http://dx.doi.org/10.1023/A:1011501232197.

R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Trans. Math. Software*, 32(3), 2006. URL http://dx.doi.org/10.1145/1163641.1163644.

R. C. Kirby and A. Logg. Finite element variational forms. In A. Logg, K.-A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 5. Springer, 2012.

J. Korelc. Automatic generation of finite-element code by simultaneous optimization of expressions. *Theor. Comput. Sci.*, 187(1–2):231–248, 1997. URL http://dx.doi.org/10.1016/S0304-3975(97)00067-4.

R. J. Labeur and G. N. Wells. Energy stable and momentum conserving hybrid finite element method for the incompressible Navier–Stokes equations. *SIAM J. Sci. Comput.*, 34(2):A889–A913, 2012. URL http://dx.doi.org/10.1137/100818583.

A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Software*, 37(2):20:1–20:28, 2010. URL http://dx.doi.org/10.1145/1731022.1731030.

A. Logg, K.-A. Mardal, and G. N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012a.

A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. FFC: The FEniCS form compiler. In A. Logg, K.-A. Mardal, and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 11. Springer, 2012b.

A. Logg, G. N. Wells, and J. Hake. DOLFIN: A C++/Python finite element library. In *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 10. Springer, 2012c.

K. Long, R. Kirby, and B. Van Bloemen Waanders. Unified embedded parallel finite element computations via software-based Fréchet differentiation. *SIAM J. Sci. Comput.*, 32:3323–3351, 2010.

M. Maraldi, G. N. Wells, and L. Molari. Phase field model for coupled displacive and diffusive microstructural processes under thermal loading. *J. Mech. Phys. Solids*, 59

(8):1596–1612, 2011. URL http://dx.doi.org/10.1016/j.jmps.2011.04.017.

G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *Internat. J. Numer. Methods Fluids*, 2012. URL http://dx.doi.org/10.1002/fld.3648.

G. R. Markall, D. A. Ham, and P. H. J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Proc. Comp. Sci.*, 1(1):1815–1823, 2010.

A. Massing, M. G. Larson, A. Logg, and M. E. Rognes. A stabilized Nitsche fictitious domain method for the Stokes problem. 2012. URL http://arxiv.org/abs/1206.1933.

M. Mortensen, H. P. Langtangen, and G. N. Wells. A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier–Stokes equations. *Adv. Water Res.*, 34(9):1082–1101, 2011. URL http://dx.doi.org/j.advwatres.2011.02.013.

J. C. Nédélec. A new family of mixed finite elements in $\mathbb{R}^3$. *Numer. Math.*, 50(1):57–81, 1986.

M. Nikbakht and G. N. Wells. Automated modelling of evolving discontinuities. *Algorithms*, 2(3):1008–1030, 2009. URL http://dx.doi.org/10.3390/a2031008.

K. B. Ølgaard and G. N. Wells. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Software*, 37(1):8:1–8:23, 2010.

K. B. Ølgaard, A. Logg, and G. N. Wells. Automated code generation for discontinuous Galerkin methods. *SIAM J. Sci. Comput.*, 31(2):849–864, 2008. URL http://dx.doi.org/10.1137/070710032.

B. A. Pearlmutter and J. M. Siskind. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–160, Nice, France, 2007. URL http://dx.doi.org/10.1145/1190216.1190242.

C. Prud'homme. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming*, 14:81–110, 2006.

C. Prud'homme. Feel++, 2011. URL https://forge.imag.fr/projects/life/.

PyChecker. PyChecker: A Python source code checking tool, 2011. URL http://pychecker.sourceforge.net.

M. E. Rognes and A. Logg. Automated goal-oriented error control I: stationary variational problems. 2012. URL http://arxiv.org/abs/1204.6643.

M. E. Rognes, R. C. Kirby, and L. A. Efficient assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming finite elements. *SIAM J. Sci. Comput.*, 31(6):4130–4151, 2009. URL http://dx.doi.org/10.1137/08073901X.

E. Rosseel and G. N. Wells. Optimal control with stochastic PDE constraints and uncertain controls. *Comput. Methods Appl. Mech. Engrg.*, 213–216:152–167, 2012. URL http://dx.doi.org/10.1016/j.cma.2011.11.026.

J. R. Shewchuk and O. Ghattas. A compiler for parallel finite element methods with domain-decomposed unstructured meshes. In D. E. Keyes and J. Xu, editors, *Proceedings of the Seventh International Conference on Domain Decomposition Methods in Scientific and Engineering Computing (Pennsylvania State University), Contemporary Mathematics*, volume 180, pages 445–450. American Mathematical Society, 1993.

J. M. Siskind and B. A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher Order Symbol. Comput.*, 21(4):361–376, 2008. URL http://dx.doi.org/10.1007/s10990-008-9037-1.

C. Taylor and P. Hood. A numerical solution of the Navier–Stokes equations using the finite element technique. *Comput. Fluids.*, 1(1):73–100, 1973.

UFL. Unified Form Language, 2012. URL `http://launchpad.net/ufl`.

P. Wang. FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis. *J. Symbolic Comput.*, 2(3):305–316, 1986. URL `http://dx.doi.org/10.1016/S0747-7171(86)80029-3`.

G. N. Wells. Analysis of an interface stabilised finite element method: The advection-diffusion-reaction equation. *SIAM J. Numer. Anal.*, 49(1):87–109, 2011. URL `http://dx.doi.org/10.1137/090775464`.

J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. *SIGPLAN Not.*, 36:298–308, 2001. URL `http://dx.doi.org/10.1145/381694.378860`.