A Generative Geometric Kernel

Jacques Carette
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4K1
Canada
carette@mcmaster.ca

Mustafa Elsheikh
PhD Student
Cheriton School of Computer Science
University of Waterloo
Canada
melsheik@uwaterloo.ca

Spencer Smith
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4K1
Canada
smiths@mcmaster.ca

Abstract

We present the design and implementation of a generative geometric kernel[1]. The kernel generator is generic, type-safe, parametrized by many design-level choices and extensible. The resulting code has minimal traces of the design abstractions. We achieve genericity through a layered design deriving concepts from affine geometry, linear algebra and abstract algebra. We achieve parametrization and type-safety by using OCaml's module system, including higher order modules. The cost of abstraction is removed by using MetaOCaml's support for code generation coupled with some annotations atop the code type.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation; D.2.2 [*Software Engineering*]: Design Tools and Techniques; I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems—Evaluation strategies

General Terms Code Generation, Geometric Kernel, Methodology, Generic Programming

Keywords MetaOCaml, Geometry, Generative, Generic

1. Introduction

Our work developed from a simple observation [12, 39, 40]: mesh generation and computational geometry software forms a family of programs. Furthermore, generative programming [6], especially when coupled with domain-specific simplifications [7], can be an effective method for "coding" such families. There are however some well-known drawbacks to the most common implementation languages, to wit C++ templates [8]. We instead chose a typed methodology, using MetaOCaml [29, 45] for higher assurance.

More precisely, a past case-study on using typed metaprogramming for capturing a program family of Gaussian Elimination algorithms [2, 4] was rather promising. But there was still a serious doubt: was Gaussian Elimination somehow especially well-suited to such an approach? We needed to know if the previously developed techniques (from the above papers as well as those of [5, 24, 44] for example) would really be transferable. Furthermore, we were interested in moving beyond case-studies of feasibility requiring the invention of new techniques, and rather extend the methodological work of [4] towards a real recipe for typed metaprogramming, at least in the context of scientific computation software.

Thus we embarked on another case study, but this time, we were quite careful to *not* invent new techniques, but rather to reuse existing techniques, or at worst to adapt them slightly. Furthermore, we always tried to use the simplest possible technique for solving each problem, even when we were aware of more powerful techniques, which could definitely solve the problem. Throughout the design and development process we paid close attention to our design decisions and the rationale behind each of them. This has allowed us to compare the current decisions and their rationale with previous work, and document the common ideas.

Motivating Example

First we need to establish that geometric computations are indeed a good topic for such a case study. Consider the following problem: Let $\{p_i\}_{i\in[1..n]}$ be n points in an n-dimensional Euclidean space equipped with an orthogonal coordinate system. Let H be the hyperplane defined by these points. That is, H is a line in 2D, a plane in 3D, or in general, a subspace of codimension 1. Let the coordinates of a point p_i be $(p_{i1}, p_{i2}, \cdots, p_{in})$. The relative position of a point $x = (x_1, \cdots, x_n)$ with regard to H is called the *orientation* of x and H. The orientation test is a fundamental geometric primitive in many computational geometry algorithms (such as computing the convex hull and triangulations [18]). The usual method for the orientation test relies on computing the following determinant:

$$\begin{vmatrix} p_{11} & p_{12} & \cdots & p_{1n} & 1 \\ p_{21} & p_{22} & \cdots & p_{2n} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} & 1 \\ x_1 & x_2 & \cdots & x_n & 1 \end{vmatrix}$$

From a software point of view, a dimension-generic implementation of the orientation test requires computing this determinant at runtime, which is an expensive operation. If the dimension is known before runtime, then a straightforward optimization is to use the expanded closed-form expression instead. For example, for $n = 2$, the determinant simplifies to

$$(p_{22} - x_2)p_{11} - (p_{12} - x_2)p_{21} + (p_{12} - p_{22})x_1,$$

[1] The code is available from http://www.cas.mcmaster.ca/~carette/ggk/

PEPM '11 January 24–25, 2011, Austin, Texas, USA.

which can be implemented with a fairly small number of arithmetic operations. For $n = 1$, this simplifies to a subtraction of two numbers $p_{11} - x_1$. It is also noticeable that the constant (1 in this case) no longer appears. This is a seemingly trivial difference, but in the context of larger computations, we cannot assume that all compilers will be able to reduce to such "trivial" computations. This means that writing an *efficient* generic orientation test requires writing and maintaining n different specialized versions for n different variations. Our goal is to write and maintain one abstract version, which can then be specialized *optimally* for each n.

In other words, in our human-maintained source code, we want the most mathematically meaningful version of the "orientation test" to appear, but in the code that we will ultimately compile (for each dimension), we want the most simplified expression. Most geometric computations have a similar character, where the abstract mathematical definition is simple and elegant, but can be computationally inefficient.

One note on efficiency: our aim is to produce *source code* which is comparable to hand-written code for the same task. Our efficiency goals are that we are no slower than hand-written (but not micro-optimized) code. Some modern compilers, which do aggressive inlining, may well produce very efficient code for highly generic algorithms; our aim is to guarantee this, rather than hope that the compiler is "sufficiently smart".

Contribution

We provide the first typed geometric kernel generator. But, more importantly, we clearly document our design decisions and their rationale, which allows us to compare to our previous work on typed generators, and extract some further methodological aspects. We have explicitly not used "new" techniques, so as to enable a cleaner extraction of methodological aspects.

We have made a very explicit design decision to encapsulate "code generation". This is the key for showing that for high-level algorithms, "generic" and "generative" can be done with the same code – and in fact resemble pseudo-code versions of mathematical algorithms. To achieve this, we need to abstract away all issues of "code generation", by localizing *all* uses of code construction operations (i.e. the features added by MetaOCaml to OCaml) to a single module.

In particular, the software architecture for our generator is mostly driven by mathematical clarity, elegance and generality. This leads to an unusually modular design, whilst the code we generate is essentially free of abstraction overheads. In fact, we achieve more inlining than what is found in many geometric kernels, from code that is written at a much more generic and abstract level.

Plan of the paper

We start with some necessary background, covering some of the basic techniques we need. Then in section 3, we cover the global design of the geometric kernel generator, in a top-down manner. Sections 4–8 then give more details of each layer, in a bottom-up manner. In section 9, we first give an illustrative example of our methodology, before providing a higher-level description of the steps we successfully followed. After this we provide further general discussion, which also covers future work. In section 11, we cover some of the related work that is not otherwise covered in the main text, before wrapping up with some conclusions in section 12.

2. Background

We give a quick overview of the basic tools that we use for all of our work. Throughout we assume the reader is familiar with Objective Caml [21].

2.1 Metaprogramming

Meta-programming involves writing programs that manipulate programs [36]. The programs being manipulated are called *object-programs*. Our use is of the simplest kind: we want to write program generators. Our generators are parametrized by *design decisions*, in the context of *program families* [32, 48]. Since this is a difficult task at the best of times, we wanted to have as many "free" correctness guarantees as possible. Currently, this means using MetaOCaml, the only language in which a type-safe code generator also guarantees that all object-programs will also be type-safe (as well as being syntactically well-formed) [42].

2.2 Programming in MetaOCaml

MetaOCaml is an extension of Objective Caml [21] (henceforth abbreviated to OCaml). It provides mechanisms for constructing and combining code expressions, which can be executed in future stages. MetaOCaml extends OCaml's syntax with three new concepts: *Bracket*, *Escape*, and *Run*. It also extends the type system by an additional type: `code`. Although MetaOCaml allows arbitrarily many stages, we only use 2, which we will sometimes refer to as the *generation* stage and the *run-time* stage.

The Bracket operator (syntax: `.<e>.`) delays the computation of an expression `e`, which is referred to as a *future-stage computation*. If `e` is an OCaml expression of type `t`, then `.<e>.` has type `('a, t) code`[2]. The execution of this expression is delayed to a future stage. However, both the syntax and the type of this expression are checked at the current stage (generation stage in our case). This gives a static guarantee that the generated code will have the appropriate type when executed at a future stage. This means that well-typed code generators will produce only syntactically well-formed and well-typed code.

The Escape operator (syntax: `.~e`) allows running of code-producing expressions in the current stage, in the context of a (future) stage code expression. When evaluating `.~e`, MetaOCaml first evaluates `e` in the current stage, then splices the resulting code expression inside a later stage expression. The power of the Escape operator lies in the fact that the expression `e` gets evaluated *before* splicing. This allows *arbitrary* computations over code expressions at generation time.

The Run operator (syntax: `.!e`) forces the execution of a code expression in the current stage; in other words, if `e` has type `('a, t) code`, `.!e` has type `t`. This is very useful for run-time code generation – which is a feature we do not in fact need here. For our purposes, it would be sufficient for there to be a perfect printer from `('a, t) code` to strings or files, albeit with the guarantee that the results are syntactically well-formed, type-safe OCaml programs. In practice, we end up using Run on code values that always evaluate to functions (at generation time).

As MetaOCaml is a strict superset of OCaml, this gives us an extremely powerful language to use at *generation* time. If we are careful which language features we use inside future-stage (generated) code, we insure that most abstraction costs are at generation time [4].

2.3 Annotations

By design, MetaOCaml treats values of type `code` as black boxes, i.e., they cannot be examined[3]. Naïve code generation will produce sub-optimal code, which we want to avoid. As we cannot optimize code post facto, we need to make sure that we gener-

[2] The polymorphic type parameter `'a` is called the *environment classifier*. The reasons and details behind environment classifiers are outside the scope of this paper. Interested readers are advised to consult [43] for more details.

[3] This is a mechanism that helps preserve the type-safety of generated code in the presence of effects, see [41] for details.

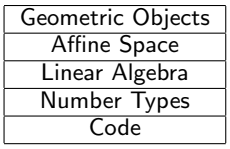

Figure 1. Overview of the layers

ate optimal code on the first (and only) pass. Kiselyov et al. [24] tackle this problem by using abstract interpretation. We use simple annotations (via algebraic types) that capture enough information about future-stage computations to allow us to perform some computations at generation time, so that we may generate the code "just right". Optimizations such as algebraic simplification, constant folding, let insertion and constant propagation are all performed before any code is generated. More precisely, we guarantee that expressions like `x+0` will never be generated [47].

In fact, we indeed use *abstract interpretations* for this purpose, but since the lattices involved are quite trivial, we prefer to refer to them simply as "annotations" here.

2.4 Modules and Functors

Our main goal is to realize a program family of geometric computations with a high degree of parametricity. Following [4], we use OCaml's module system for encoding the domain concepts and the parametrizations. The abstract concepts are encoded as module types (interfaces), with modules being instances of those concepts. The type checker can then reject invalid implementations of these concepts. To express variabilities, we use Functors. Functors are parametrized modules. By adding appropriate type constraints, we can enforce that certain relations must hold between parameters of Functors. This allows us to encode certain domain information into the type system, and leverage the type checker's ability to inform us of improper compositions of variabilities.

3. The Design of the Geometric Kernel

Our original motivation was to be able to experiment with differing meshing algorithms, in the context of solving partial differential equations. All meshing algorithms rely on a core foundation of geometric objects and computations [27]. To be able to conduct such experiments, we need a whole family of (implementations of) geometric objects and computations. The abstraction cost of making all of these choices at run-time, through whatever dispatching mechanism we may choose, is simply too high – we could never hope to be competitive with existing meshing software.

It thus seemed reasonable for us to design a *geometric kernel* that was both parametric and efficient; previous work told us that this could be achieved via code generation. We also knew that we needed to find "the right abstractions" to be able to be properly modular and achieve the desired parametricity.

The problem of finding a proper set of abstractions that allows expressing geometric computations independently of the underlying coordinate space and its dimensions is a thorny one, but study of the literature made it clear that this was key. If this can be achieved, then a *family* of geometric computations can be expressed using essentially the same abstract code. As a positive side-effect, this would also scale to higher dimensions. In other words, if we can find a proper interface for the lower-level implementation details, we could then provide different implementations (aka *variabilities*) for those details. It is important to remember that we have put ourselves in a generative context, so we need to make choices that allow us to specialize the generator, rather than directly specializing the program. However, the partial evaluation literature tells us

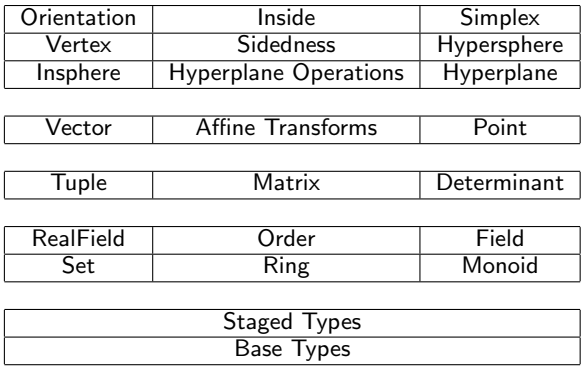

Figure 2. Details of the layers.

that, through staging, we can turn programs directly into their own *generating extensions* [22] by hand-writing a `cogen`.

We first sought to proceed as in [2], and "reverse engineer" the commonalities and variabilities from a sample set of geometric kernels, which embodied different design choices. This did not work. These kernels where much too highly optimized, thereby completely obscuring the abstract structures involved. The choice of Gaussian Elimination in [2] was *lucky* in that the mathematical structure was still visible in the code.

We thus turned away from the literature describing implementations of geometric algorithms, and started exploring abstract geometric computations instead. Mann et al. [26] propose using a geometric algebra as a basis for geometric abstractions. This approach results in generic higher-dimensional geometric kernels [28], and coordinate-free computations [20]. A comparison of the different types of geometric algebras can be found in [14, 19]. Among these different geometric algebras, affine geometry treats n-dimensional spaces through the same set of abstract concepts. In effect, geometric primitives that are based on affine geometry scale to higher dimensions. However, the affine geometry does not allow a straightforward extension to different coordinate systems other than the orthogonal coordinate systems [10]. A promising alternative geometric algebra can be found in [14] that can address this problem. For simplicity, we chose to follow the route of basing our geometric kernel on affine algebra.

Once we decided that we would not worry about non-Cartesian coordinate systems, the rest of the design became simpler. The final design (see figure 1) was completely motivated by top-down functionality requirements, in other words the implementation needs of the upper-layers drove the requirements for the lower layers.

Briefly, our geometric objects are encoded in terms of concepts from affine spaces, which are encoded using linear algebra (and containers), which itself is parametric over the underlying "number types" of the entries of matrices. And, at the bottom, the very representation of the programming language primitives (data-types and functions) are themselves abstracted into staged versions – which we will detail later.

Each layer is designed to be generic and parametric, however our current implementation only covers those parts that are necessary to implement variations on our geometric kernel.

Details of the Layers

The layers' contents are further detailed in Figure 2. The purpose and functionality of each of the layers is described as follows:

- The *geometric objects layer*: Geometric objects and computations are exposed in sufficient detail to allow writing geometric algorithms independently from the underlying geometric space

representations. Choices such as the coordinate system, the coordinate number types, or dimensions are not exposed. Examples of objects at this layer are hyperplanes, hyperspheres, and operations on them.

- The *affine space layer* provides the basic functionality for computing with geometric primitives of affine spaces. The concepts here are those for points, vectors, orientation, sidedness and general affine transforms.
- The *linear algebra layer*: A straightforward implementation of affine geometric primitives uses linear algebra. The scope of this layer is motivated by the needs of the affine geometry, not by the general scope of linear algebra. We only require a (representation of) vectors, which we call tuple, a (representation of) linear transformation – matrices, and the computation of determinants. However, obtaining "good" code for this was challenging, and drove a lot of our design for facilities in lower layers.
- The *number types layer*: Proper abstractions in geometric computation include identifying a "number type" for the underlying coordinate system. Traditionally this has been seen in a narrow fashion as being entirely determined by the carrier type – but here when we speak of "number type", we really mean an algebraic approach to these. So this layer exposes *algebraic structures* in their entirety, rather than leaving this implicit. This in turn allows us to place certain simplifications in the "right" place (i.e. in this layer) instead of being ad hoc; for example, constant folding can be done here, as can simplifications based on algebraic properties.
- The *code layer* provides code generation facilities. It provides abstractions for "staging" values and functions, as well as some code combinators.

In the next sections, we will describe, from the bottom up, each one of these layers in more detail.

4. The Code Layer

The code layer offers abstractions for building and manipulating staged types, as well as code generation combinators.

We have made a very explicit design decision to encapsulate "code generation". Our goal is to provide the "right" primitives so that, at the higher levels, generic algorithms and generative algorithms are identical – and in fact resemble pseudo-code versions of mathematical algorithms. To achieve this, we need to abstract away all issues of "code generation", by localizing all uses of Escape and Bracket to a single module in the code layer.

4.1 Abstracting code

We could proceed by simply abstracting the code type and the operations on it. However, we noticed that a fair amount of information is available statically, as generation-time values. We want to be able to take advantage of this to simplify the generated code (e.g. by constant folding), yet still have the higher layers of our code be unaware of such issues. We now explain how we provide a unified view of code and values.

Code Expressions

As is by now well-understood [4, 24], it is important not to duplicate computations. To facilitate let insertion, we need to know if a code value can be duplicated or not; rather than encoding at the type-level like the `maybeValue` type of [24], we use the value-level at generation time. This is naturally isomorphic, but allows for more elegant code. We define the type `code_expr` as

```
type ('a,'b) code_expr =
  { c : ('a,'b) code; a : bool }
```

where the boolean represents whether the code expression can be duplicated or not (we call such expressions *atomic*). An expression `e` is atomic if and only if it is either an immediate value or (the code for) a variable.

Staged Expressions

Expressions can be classified in terms of evaluation time into two categories:

1. *Now expressions*. This category comprises immediate values and constants. The value of a Now expression is known at code generation time.
2. *Later expressions*. Values in this category are known at run-time. At generation time, the values in this category are represented by code expressions whose execution is delayed to a further stage, i.e., run-time.

As we want to abstract away from these considerations, we combine these into a single type of `staged` values:

```
type ('a,'b) staged =
  | Now of 'b
  | Later of ('a,'b) code_expr
```

4.2 Staging Functions

Once we have staged values, we need to build functions over these, which we will naturally call staged functions. Our mechanism is general, and only depends on the arity of the function.

4.2.1 Staging Unary Functions

A straightforward way to lift a function of type `'b -> 'c` to a staged one would be

```
let lift_unary f = function
  | Now x -> Now (f x)
  | Later x -> Later { a = false;
                       c = .< f .~(x.c) >. }
```

but this inlining could cause code duplication. This can be improved to

```
let lift_unary' f = function
  | Now x -> Now (f x)
  | Later x -> Later ({ a = false; c =
      if x.a then .< f .~(x.c) >.
      else .< let t = .~(x.c) in f t >. })
```

This solution is still not entirely satisfactory, as function composition now generates rather unnatural code. Furthermore, we are residualising a call to `f`, even though we might possess a version of `f` that does inlining. The problem is that we do not have enough contextual information at hand to deal with this case properly.

A better solution then is to *delegate* the handling of the `Later` case to the caller. That is, rather than asking for a single function f working on values as input, we rather ask for f_n working on values and f_l which works on code expressions. We bundle these up into a record, and a straightforward implementation of the application for these generalized unary functions is

```
type ('a,'b,'c) unary = {
  unow: 'b -> 'c ;
  ulater: ('a,'b) code_expr -> ('a, 'c) code_expr
}
let mk_unary f = function
  | Now x -> Now (f.unow x)
  | Later x -> Later (f.ulater x)
```

4.2.2 Staging Binary Functions

If we apply the above scheme to binary functions in a naïve manner, we would require 4 cases for a generalized binary functions, which does not seem quite "right". What we would prefer would be to give only 2 cases, and use lifting for the other cases.

To achieve this, we need a simple auxiliary function to lift atomic values (in this case, constants). Then we can define a type for generalized binary functions and its application function.

```
let lift_const x = {c = .<x>.; a = true}
type ('a,'b,'c,'d) binary = {
  bnow : 'b -> 'c -> 'd ;
  blater : ('a,'b) code_expr ->
             ('a, 'c) code_expr ->
               ('a,'d) code_expr
}
let mk_binary bop = fun x y -> match x, y with
 | (Now x), (Now y) -> Now (bop.bnow x y)
 | (Now x), (Later y) ->
     Later (bop.blater (lift_const x) y)
 | (Later x), (Now y) ->
     Later (bop.blater x (lift_const y))
 | (Later x), (Later y) -> Later (bop.blater x y)
```

It is important to note that the above is still sub-optimal: the knowledge of one of the static arguments could sometimes be useful for simplifications, but is not used. We will rectify this in section 5.1.

4.3 Generating Let Statements

Straightforward inlining can easily result in code duplication. The usual solution [3, 24] involves writing the generator in continuation passing style (CPS). However, in the cases we treat here, we do not need to use such high-powered tools. We can instead use a simpler `let` generator.

The principal usage is to express sharing in *non-linear expressions*, which are expressions where a value is used more than once (for example, in the expression `x+(x+y)`, `x` is used non-linearly). We want to rewrite this as `let v=x in v+(v+y)`. To achieve this, we need the "body" `v+(v+y)` to be expressed as a function (of `v`). This can be done in two ways, either as a staged function, or as a generalized unary function. In each case, we can write a let generator over staged values – which we call `let_` for the staged case and `letp` for the generalized unary function case.

```
let lift_atom x = {c=x;a=true}
let let_ ce exp = match ce with
  | Now _ -> exp ce
  | Later c when c.a -> exp ce
  | Later c ->
    of_comp .< let v = .~(c.c) in
      .~ (to_code (exp (lift_atom .<v>.))) >.
let letp ce exp = match ce with
  | Now v -> Now (exp.unow v)
  | Later c when c.a ->
    Later (exp.ulater (lift_atom c.c))
  | Later c -> Later
    { c = .< let v = .~(c.c) in
        .~((exp.ulater (lift_atom .<v>.)).c) >.;
      a = false }
```

In practice, we frequently use `let_` as it gives us the same behaviour as `letp` in simple situations.

4.4 Base Types and code combinators

We also provide some simple facilities for generating code involving boolean predicates (as well as for String, but it is not used). Some code combinators (like sequencing and choice) are likewise provided.

5. The Number Types Layer

The number types layer provides generic abstractions of the number types. It is probably misnamed: it should properly be called the "single-sorted universal algebra" layer. But as that is rather a mouthful, and our uses are largely for algebras of "numbers", we have decided to remain with our early name. Its purpose remains the same: encapsulating the right properties so that we may implement a *family* of algorithms with the number type as a *variability*. Thus we introduce a hierarchy of abstractions based on traditional algebraic structures, such as rings and fields. However, we do not strive for completeness, but rather only define those structures that naturally appear in our algorithms.

In retrospect, this layer really should be divided into two layers: we have a first part that directly uses the low-level code layer objects (like `unary` and `binary`) for building simplifiers for algebraic structures, encoded using records. The second part uses the `staged` types to create "staged" versions of algebraic structures – encoded using module types.

5.1 Building Simplifiers

Monoids Although monoids, as a mathematical structure, are rather uninteresting (they support very few theorems), they are a pervasive structure in computer science. We recall that a monoid consists of a carrier set M, an associative binary operation $\star$ over M, and a special element $u \in M$ such that u is a left and right unit for $\star$. We can take advantage of staging and build *better* staged operators that respect the identity laws of the base operators. That is, the resulting staged operator can generate code that does not contain unnecessary operations, for all "types" that support a monoidal structure. In particular, we can generically simplify $x + 0$ and $1 * x$ to x in the monoids $(\mathbb{B}, +, 0)$ and $(\mathbb{Q}, *, 1)$, respectively.

Concretely, we can create a type for monoids, and then create a special evaluator for the monoid's binary operation that takes advantage of the monoidal structure to perform simplification whenever possible. When no simplification applies, we just dispatch to the generic case of staged binary operations, covered in the previous section.

```
type ('a,'b) monoid = {
  bop : ('a, 'b, 'b, 'b) binary;
  uelem : 'b
}
let mk_monoid mon x y = match x, y with
  | (Now x), y when x = (mon.uelem) -> y
  | x, (Now y) when y = (mon.uelem) -> x
  | x,y -> mk_binary mon.bop x y
```

Rings Mathematically, a *ring* consists of a carrier set R, a commutative associative binary operator $+$ with two-sided unit $0 \in R$, and an associative binary operator $*$ with two-sided unit $1 \in R$, where $*$ distributes over $+$ and 0 acts as an annihilator for $*$. Seen another way, a ring consists of 2 monoidal structures over the same carrier set R which satisfy some compatibility laws. As with a monoid, we can make a type `ring` and a simplifier:

```
type ('a,'b) ring = {
  monp : ('a,'b) monoid;
  mont : ('a,'b) monoid;
}
let mult_ring rng x y = match x, y with
  | (Now x), (Later y) when
      x = (rng.monp.uelem) -> Now rng.monp.uelem
  | (Later x), (Now y) when
      y = (rng.monp.uelem) -> Now rng.monp.uelem
  | x, y -> mk_monoid rng.mont x y
```

Note how we take definite advantage of the multiplicative monoid structure when simplifying the multiplicative aspect of a ring. The additive aspect is exactly covered by the additive monoid structure.

5.2 Staged Structures

We use Ocaml's module types to encode concepts such as sets, orderings, monoid, normed set, ring, field and real field. Each interface (aka module type) specifies a minimal set of (typed) operations for each concept. To minimize duplication of code at this level as well, we use OCaml's module inclusion for extension among types. The following *types* are ordered by the inclusion relation ($A \sqsubseteq B$ if module B includes module A):

$$\texttt{SET} \sqsubseteq \texttt{RING} \sqsubseteq \texttt{FIELD} \sqsubseteq \texttt{REALFIELD}.$$

By writing generators that are independent of the choice of representation, we obtain genericity. However, since we are in a generative context, the code generated is specific to the choices made. The principal reason for using modules (rather than records) here is sub-typing.

6. The Linear Algebra Layer

An abstract implementation of affine geometry uses matrices. But since we will always be generating code for fixed, usually very small, dimensions, the overhead associated with matrices (both in terms of time and space) are prohibitive. But this is not a problem: our linear algebra layer uses matrices at *generation time* only. We have made sure that all linear algebra operations are always computed strictly at generation time.

On the other hand, linear algebra provides an extremely useful, and generic, formalism for expressing all the important operations in affine geometry. We thus use this formalism, in the generator, as a convenient abstraction between the affine geometry layer and the "algebra" layer.

The most important function in this layer is the computation of *determinants*. Most of the affine geometry operations which we are interested in, are defined in terms of special determinants. However, these determinants are performed on structured matrices, with many statically known entries, many of which are in fact 0 and 1. By doing a careful expansion of the determinant used in the definition of each concept at generation-time, we can produce "efficient" computation sequences for them.

7. The Affine Space Layer

This layer encapsulates the concepts from affine geometry, and drawn from both the literature on the mathematics of geometry as well as computational geometry [1, 9, 17, 34]. We selected concepts that were n-dimensional, abstract but specializable, and that were required for mesh generation computations [15, 18].

Our design follows the mathematical abstraction in the affine geometry domain. For example, an affine space is defined as a triple $(\mathcal{D}, \mathcal{V}, \mathcal{P})$, where $\mathcal{D}$ is a division ring, $\mathcal{V}$ is a set of free vectors over $\mathcal{D}$, and $\mathcal{P}$ is a set of points (also over $\mathcal{D}$). In a sense, an affine space is a vector space without a marked "origin". This abstract definition says little about the dimension of the space or the coordinate systems. However, certain concepts can be extracted from the definition such as number types, vectors, and points. At this level, the interfaces corresponding to those objects should make no assumptions about dimensions or coordinate system. In other terms, the API for these module types should be dimensionless and coordinate-free.

Matrix Representation of Affine Transforms

An affine transform T on a vector x can be represented as $Tx = Ax + b$ where A is an n by n matrix, and b and x are both n-dimensional vectors. A is called the *linear transformation matrix*. x is the column vector representing the affine coordinates of the object subject to transformation. b is the additive part of the transformation, normally called the *translation* part. If we denote by a bold font symbol (such as $\mathbf{a}$) an n-dimensional column vector, then $T\mathbf{x} = \mathbf{y} = A\mathbf{x} + b$ can equivalently be written as

$$\begin{pmatrix} \mathbf{y} \\ 1 \end{pmatrix} = \begin{pmatrix} A & \mathbf{b} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix}$$

The expressive power that we get from the linear algebra layer allows us to implement the affine transforms following the formulation above.

8. The Geometric Objects Layer

Here we finally get geometric computations, with our selection driven by the prevalence of the objects and computations in mesh generation systems. There are challenges in designing such a layer: in typical texts on mesh generation, as well as on computational geometry, most primitive operations are given in special forms for 2 and 3 dimensions only. For example, a 2D in-circle test is usually defined as testing whether a point x is inside the circle circumscribing a particular triangle t, while in 3D the circle becomes a sphere and the triangle, a tetrahedron. All too frequently, 2 separate expressions are given for each of these computations, with no formal link between them. More abstract presentations define the in-circle test as testing whether a point is inside a hypersphere circumscribing a simplex – and then proceed to give a general constructive definition via the computation of the determinant of a particular matrix built in terms of the coordinates of the point x and simplex t. At this point, all our work on the lower layers pays off immediately: this abstract definition can directly be used in our generator, and furthermore the code it generates in special cases is identical to the special form found in textbooks. Note that we do not have to specialize the dimension – our generator can also produce dimension-generic forms of the in-circle test.

We give a quick overview of the fundamental geometric objects we have implemented.

Hyperplanes A hyperplane is an n-dimensional generalization of a point in 1D, a line in 2D, and a plane 3D. More specifically, it is a codimension 1 linear subspace of an n-dimensional space. It is frequently defined by giving $n - 1$ non-colinear points that lie on the hyperplane. The implementation of a hyperplane is free to choose whatever "container" representation it wishes for storing those points. The interface for hyperplanes leaves the representation completely abstract. Hyperplanes are parametrized by (representations for) a normed set, a vector and a point and some compatibility constraints. All of these are dimensionless and coordinate-free.

The interface to hyperplanes exposes functions for their construction, extracting the dimension, the normal vector and offset, as well as operations for 'frames' and local coordinates. Above this, one can then generically build functions for computing the distance between a point and hyperplane, computing whether a point is "above" or "below" (or on) a hyperplane, as well as projecting a point onto a hyperplane.

For example, given a hyperplane implementation `H`, we can compute the distance between a point p and a hyperplane h, as $\hat{n} \cdot (p - o)$, where $\hat{n}$ is a vector normal to h and o is an arbitrary point on h. In code, this is

```
let dist h p =
  H.V.dot (H.normal h) (H.P.sub p (H.orig h))

val dist : 'a H.h -> 'a H.P.point -> 'a H.V.N.n = <fun>
```

We can see the delegation of duties – we ask the underlying vector space representation for the computation of a dot product, and the point presentation for the computation of "subtraction" of 2 points (which returns a vector).

Hyperspheres A hypersphere is a generalization of circle and sphere to any dimension. The implementation of a hypersphere is quite similar to that of a hyperplane – both are defined in terms of points, as we are interested in circumscribing spheres defined by points on the circumference, rather than general spheres defined by radius-center. Currently, the interface for hyperspheres is less extensive than that for a hyperplane, but they have a number of commonalities, as they are both orientable surface that can be closed (hypersphere) or open (hyperplane). The interface exposes functions for construction, and getting an arbitrary point on the hypersphere. From this one can provide generic implementations of getting the center point, the content (volume), the "surface", and defining an in-sphere predicate. The formulas are much more complicated than the formula for distance seen above, but nevertheless all reduce to operations from layers below this one.

Simplex A simplex is a generalization of the concept of a triangle to any dimension. The interface `SIMPLEX` provides functionality for construction of simplices, retrieval of vertices and faces, and accessing neighbourhood information. Then one can generically build an is-inside predicate for testing whether a point is inside a simplex.

9. Methodology

We first demonstrate the bottom-up part of our methodology for building code generators through a simple example. Although we were sorely tempted to use the `power` function for this purpose (as it actually demonstrates many of our ideas rather well), we will use ℓ^1 vector norm instead.

9.1 Example: ℓ^1 norm

The simplest implementation is for a `list` of `floats`:

```
let rec norm = function
    | [] -> 0.
    | x::xs -> abs_float x +. norm xs
val norm : float list -> float = <fun>
```

We achieve the task of building a generator for the above function via the following six steps:

(1) Generalize the type of numbers. What we really need is a 'normed set'. Our "numbers" can come from an arbitrary set as long as we have a function from that set into a commutative additive monoid as the 'target' of the norm function. Let us use `NS` for a normed set, over a normed commutative monoid `R`. Then we can write

```
let rec norm = function
    | [] -> NS.R.zero
    | x::xs -> NS.R.plus (NS.norm x) (norm xs)
val norm : NS.n list -> NS.R.n = <fun>
```

(2) Staging `norm`. We now lift from the normed set type `NS` to a staged version

```
let rec norm = function
    | [] -> Staged.of_immediate NS.R.zero
    | x::xs -> NS.R.plus_s (NS.norm_s x) (norm xs)
val norm : 'a NS.ns list -> 'a NS.R.ns = <fun>
```

where we use the staged version of `NS.R.plus` and `NS.norm`.

(3) Parametrizing `norm`. The function `norm` is parametric in `R`. By using a Functor, we can better express this parametricity. So we can lift this to use a Functor:

```
module GenericNorm (NS : NORMED_SET) =
struct
  let rec norm = function
    | [] -> Staged.of_immediate NS.R.zero
    | x::xs -> NS.R.plus_s (NS.norm_s x) (norm xs)
end
```

(4) Abstracting the container. Vectors do not have to be represented by lists - any container which implements a fold will do. We can repeat steps (1)-(3) above, but for the type of lists generalized to any type with a fold. Since it is frequent to first do a 'map' then a 'fold', an optimized version is provided. The most general code would then be

```
module GenericNorm(NS : NORMED_SET)
                  (C: FOLDABLE with t = R.n) =
struct
  let norm = C.mapfold(NS.norm_s)(NS.R.plus_s)
                (Staged.of_immediate NS.R.zero)
end
```

(5) Collecting variabilities. The generator `GenericNorm` has two variabilities: the normed set, and the container. These variabilities can be collected in the (module) type:

```
module type NORM_VAR =
sig
  module NS : NORMED_SET
  module C : FOLDABLE with t = NS.n
end
```

which becomes the input for the generator.

(6) Building the generator. The generator, `GenNorm`, is defined as:

```
module GenNorm (NV : NORM_VAR) =
struct
 let gen_norm () =
  let module GP = GenericNorm(NV.NS)(NV.C) in
   .< fun x -> .~(Staged.to_code
       (GP.norm (Staged.of_atom .<x>.))) >.
end

module GenNorm :
 functor (NV : NORM_VAR) ->
  sig
   val gen_norm :
    unit -> ('a, NV.NS.n NV.C.t -> NV.NS.R.n) code
  end
```

The generator `gen_norm` is put in a functor which takes as input the module `NV` of variabilities. In other words, `NV` is the configuration for the generator.

In fact, all of our containers are fixed-length. This is another variability which can be exposed (but we have chosen not to do so in this example to try to keep things relatively simple).

We can instantiate our norm function with different number type implementations as well as different containers. If we choose say `Float_Ring` as our instantiation of a normed set (where floats are elements of the set and floats are also used as the commutative monoid), and a 2-tuple for the container, the result is:

```
.<fun x_1 -> abs_float(fst x_1) +.
             (abs_float(snd x_1))>.
```

Choosing `IntegerRing` and a 1 dimensional container represented using a record (with single entry `x`), we get:

```
  .<fun x_1 -> abs (x_1.x) >.
```

The code has no traces of `norm` or `mapfold`. Furthermore, there are no traces of the "zero" of the monoid, since it always occurs statically and operations with it can always be optimized out at generation time.

9.2 Aspects of the Methodology

The bottom-up aspects of our methodology are well-known in the generic programming [30, 37] world. From pieces of working code, one finds the minimal assumptions necessary for ensuring the operational semantics and the correctness properties, and then abstracts out every other aspect. As shown above, this can frequently be done incrementally. There are two difficult aspects of this task: recognizing the minimal assumptions, and abstracting out "the rest". Recognizing minimal assumptions is a very mathematical endeavour, which requires a thorough knowledge of abstract algebra, geometry, topology, etc. The abstracting step on the other hand is difficult because one must use the facilities available for abstraction in the host programming language. Finding appropriate encodings of structures to enable parametrization can be quite challenging in a typed environment.

The top-down aspects were somewhat counter-intuitive to us, when we first encountered them:

1. The *code generation* aspects need to be abstracted out *first*, and integrated at the very base of the hierarchy.
2. Very abstract mathematical formulations help produce the best code. Our reasoning here is that the abstract formulation captures the core knowledge of each concept. Once this is captured, then it becomes quite clear where to put in particular optimizations (like algebraic simplifications), which are otherwise done in an ad hoc manner.
3. More layers produces better code than fewer layers. In normal code, more layers produces slower programs – although it does improve modularity. In the generative setting, more layers helps expose more opportunities for attaching meta-knowledge (in other words, static invariants), and they are then used for generating better code.
4. Separating out operational concepts (like arrays) from mathematical concepts (like matrices) is crucial. An array is really a memory arrangement, while a matrix is a representation of a linear operator with respect to a certain basis. The "natural" operations on matrices have completely different interfaces than those of arrays.
5. Separating out concepts (like points and vectors) that are usually stored the same way, but whose semantics is quite different.

There is a real tension between the bottom-up and the top-down aspects of the methodology. For example, when we first attempted what amounts to a pure bottom-up approach to "reverse engineer" abstract mathematical structures from a collection of existing geometry kernels, we failed. This is because we did not, at that point, know about *geometric algebra*. This was the key abstraction we were missing to proceed. From that point on, we actively looked for more such abstractions, and used them as appropriate.

10. Discussion

Here we discuss a few more items we learned while performing this work.

10.1 Infrastructure

At first, the additional infrastructure seems rather heavyweight. However the most complex parts are actually the bottom 3 layers, and the upper layers end up actually being much simpler than "normal". This is because the code in the upper layers is essentially identical to the abstract mathematical formulation of the concepts. Furthermore, as more variabilities are added, the payoff becomes increasingly clear, as the amount of extra effort necessary is very small. And since these lower layers are quite generic, they should be shared amongst any program families of scientific computation software, thus lowering the overall development costs.

10.2 Linear algebra

We want to remark again that our linear algebra layer performs all of its computations at generation-time. The computations performed here are the generation of *simplified closed-form expressions* for determinant expansions. This hints that other computations that are essentially symbolic *computer algebra* may become an important part of code generators.

10.3 Eliminating Code Duplication

We already covered the generation of let bindings in section 4.3. But it is easy to get the impression that this is a feature only used in the lower layers, which is not at all the case. For example, consider the function `length` for computing the Euclidean length of a vector. For `N` a `REALFIELD`, a naive implementation of `length` is:

```
let length v = N.sqrt_s (dot v v)
```

However, if `v` is an non-atomic expression, the resulting code will contain duplications. We can do this via

```
let length v = N.sqrt_s(let_ v (fun x->dot x x))
```

which avoids the duplication.

10.4 Optimizations

By leveraging techniques originally developed for partial evaluation, we are able to write a code generator that performs both *constant folding* and *constant propagation*. By tracking which (code) values are non-atomic, we can use known let-insertion techniques to make sure that these computations are not duplicated. We do this not only for "base" values, but also for compound data-structures (like vectors), where the gain is even more noticeable[4].

While many have implemented algebraic simplifications before us, we believe that we have exploited the underlying mathematical *structures* in a novel manner. In other words, rather than using the mathematical structures only in (generally paper) proofs of the correctness of our algebraic simplifications, we use the structures directly in the code. This also serves to structure our own code, as well as making these simplifications generic.

10.5 Future Work

While we achieved our goals with respect to what we set out to learn, based on that knowledge, there are a number of items we would like to do.

1. *Expand the staging layer.* Many refinements can be done in the staging layer.
   (a) Modifying the type `staged` to accommodate more complex types such as staged pairs and records. In particular, apply the techniques from [11].
   (b) Employing the monadic techniques from [4], to improve the generation of control structures, and let-bindings.
   (c) Implementing more code optimizations such as loop unrolling and other code transformations from [5].

[4] At the source code level. We encourage the reader to run the tests included (directory `ctest`) in our distribution, and to inspect the generated code.

2. *Expand the abstract algebra layer.* For example, the current abstractions do not differentiate between concepts such as division rings and commutative rings. Such a finer set of abstractions will allow expressing the geometric algorithms in terms as close as possible to the domain concepts and hence, reduce the gap between program code and domain concepts. This will also be reusable for many other projects.
3. *Experiment with different geometric algebras.* Affine algebra is not the only choice of "geometric algebra." Other algebras should also be considered in future experiments. For example, the algebra in [14] looks promising for handling different coordinate systems.
4. Rework our implementation of mesh generation (not presented here) to use our geometric kernel.
5. Determine how to best encode domain-specific constraints. Some of these are implemented via types, while others are done via generation-time values. The advantage of encoding this information in the type system is that the compiler will automatically track this for us; the disadvantage is that the resulting error messages can be generously described as cryptic. Using generation-time values requires more initial effort from the generator-writer, but the benefit to the user of the generator is that we can output precise, domain-specific error messages.

11. Related Work

Previous work on the geometric core of mesh generation and computational geometry has focused on obtaining genericity, flexibility, or performance, but rarely attempting all three simultaneously.

Simpson [38] presents an attempt to decouple mesh generators from the underlying geometry by using object-oriented programming techniques to dynamically bind computations to different local coordinate representations. However, object-oriented programming can introduce run-time overhead because of dynamic dispatch. XYZ GeoBench [35] offers a programming environment for implementing geometric algorithms by relying on object-oriented programming and virtual functions to implement genericity – for example their geometric algorithms are implemented in an arithmetic-independent manner. However, here again, the use of dynamic binding for achieving flexibility can result in performance penalties. By avoiding dynamic dispatch, we believe we can achieve the same level of genericity as these libraries, but better performance.

LEDA [28] is a comprehensive library of data types and algorithms focused on geometry. LEDA makes use of C++ templates to achieve genericity. The library has a layered design that decouples geometric algorithms from number types. The independence from coordinate systems is achieved by having two sets of geometric kernels: one for Cartesian coordinates, and another for the homogeneous coordinates. This duplication is a challenge to the extensibility and maintainability of the overall library. Our current work covers but a small fraction of their extensive library; however, we believe our design will scale better than theirs. We also have the advantage of the stronger abstraction mechanisms available in OCaml over those of C++.

Another large C++ library for geometry, also using templates is CGAL [13]. They achieve extensibility by parametrizing each geometric object by the geometric kernel type and the number type. For vector mathematics, Blitz++ [46] pioneered many of the C++ template meta-programming technologies used in LEDA and CGAL. As with LEDA, we believe that our design and use of OCaml's abstraction mechanisms will allow us to scale further.

The work on using MetaOCaml for scientific computation has already been cited several times throughout this paper.

Generative programming in the context of numeric computation has a very long history: the 1953 work of Kahrimanian [23] on what is now called automatic differentiation is a personal favourite. By 1972 (see for example the beautiful work of A. Norman [31] on code generation for the solution of ODEs), the techniques were already quite advanced. Outside the C++ libraries already mentioned, modern work would include FFTW [16] and Spiral [33].

12. Conclusion

Our eventual goal is to have a rich program family of mesh generators, implemented as a generator. Here we report on the foundations of this, which is the implementation and design of a program family of geometric kernels, implemented generatively, using typed meta-programming. By cleanly separating run-time from generation-time, we are able to create a much more modular program family (for geometry) than has previously been done, without paying the abstraction cost associated with traditional modularity mechanisms. More precisely, the source code which we obtain from our generator is comparable to the human-written source code for the same specialized situation.

We believe that typed generative programming is relatively successful for the implementation of program families because it allows extremely modular code as well as the use of advanced abstraction techniques (in the generator) without sacrificing efficiency of the generated code. It does impose an extra burden on the generator writer: being conversant with a level of abstract mathematics that is not commonly seen in computer science education. There is also an extra design effort consisting in capturing some invariants (like being either "atomic" or a computation) which are only obvious post-facto. But, in the end, since the generator's higher layers are based on abstractions that are natural to the domain itself, we believe this makes correctness much easier to verify, as well as improving maintainability.

Furthermore, for well-understood domains, we have developed a methodology for writing (typed) generators based on leveraging the most abstract mathematical formalization of that domain for structuring modules. As this methodology is compatible with the successive generalization of existing code, it can be applied to existing (OCaml) codes.

References

[1] Michele Audin. *Geometry*. Springer-Verlag, Inc., New York, NY, USA, 2003.

[2] Jacques Carette. Gaussian elimination: a case study in efficient genericity with MetaOCaml. *Sci. Comput. Program.*, 62(1):3–24, 2006.

[3] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In Robert Glück and Michael Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 256–274. Springer Berlin / Heidelberg, 2005.

[4] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, In Press, Corrected Proof, 2008.

[5] Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.*, 62(1):25–46, 2006.

[6] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[7] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Gluck, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on*

Generic Programming, pages 25–39, London, UK, 2000. Springer-Verlag.

[8] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Lengauer et al. [25], pages 51–72.

[9] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1st edition, 1997.

[10] T. D. DeRose. A coordinate-free approach to geometric programming. *Theory and practice of geometric modeling*, pages 291–305, 1989.

[11] Iavor Diatchki and Tim Sheard. Staging algebraic datatypes. `http://web.cecs.pdx.edu/~sheard/papers/stagedData.ps`.

[12] A. H. ElSheikh, S. Smith, and S. E. Chidiac. Semi-formal design of reliable mesh generation systems. *Adv. Eng. Softw.*, 35(12):827–841, 2004.

[13] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. Pract. Exper.*, 30(11):1167–1202, 2000.

[14] Daniel Fontijne and Leo Dorst. Modeling 3D Euclidean geometry. *IEEE Comput. Graph. Appl.*, 23(2):68–78, 2003.

[15] Pascal Jean Frey and Paul-Louis George. *Mesh Generation: Application to Finite Elements*. ISTE, 2007.

[16] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[17] Jean Gallier. *Geometric methods and applications: for computer science and engineering*. Springer-Verlag, London, UK, 2000.

[18] P.L. George and H. Borouchaki. *Delaunay Triangulation and Meshing: Application to Finite-Elements*. Hermes, New York, NY, 1998.

[19] Ron Goldman. On the algebraic and geometric foundations of computer graphics. *ACM Trans. Graph.*, 21(1):52–86, 2002.

[20] Philip W. Grant, Magne Haveraaen, and Michael F. Webster. Coordinate free programming of computational fluid dynamics problems. *Sci. Program.*, 8(4):211–230, 2000.

[21] Objective Caml home page. `http://caml.inria.fr/ocaml/`.

[22] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[23] H. G. Kahrimanian. Analytical differentiation by a digital computer. Master's thesis, Temple University, May 1953.

[24] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 249–258, New York, NY, USA, 2004. ACM.

[25] Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors. *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*. Springer, 2004.

[26] Stephen Mann, Nathan Litke, and Tony Derose. A Coordinate Free Geometry ADT. Technical Report CS-97-15, Computer Science Dept., University of Waterloo, 1997.

[27] Kurt Mehlhorn, Michael Müller, Stefan Näher, Stefan Schirra, Michael Seel, Christian Uhrig, and Joachim Ziegler. A computational basis for higher-dimensional computational geometry and applications. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 254–263, New York, NY, USA, 1997. ACM.

[28] Kurt Mehlhorn and Stefan Näher. *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, New York, NY, USA, 1999.

[29] MetaOCaml Home Page. `http://www.metaocaml.org/`.

[30] D. R. Musser and A. A. Stepanov. Generic programming. In P. (Patrizia) Gianni, editor, *Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer Verlag, 1989.

[31] Arthur C. Norman. A system for the solution of initial and two-point boundary value problems. In *Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 826–834, New York, NY, USA, 1972. ACM.

[32] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[33] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[34] Philip J. Schneider and David Eberly. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[35] Peter Schorn. Implementing the XYZ GeoBench: A programming environment for geometric algorithms. In *CG '91: Proceedings of the International Workshop on Computational Geometry – Methods, Algorithms and Applications*, pages 187–202, London, UK, 1991. Springer-Verlag.

[36] Tim Sheard. Accomplishments and research challenges in metaprogramming. In *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[37] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 59–70, London, UK, 1998. Springer-Verlag.

[38] R. Bruce Simpson. Isolating geometry in mesh programming. In *Proc. of the 8th Int'l Meshing Roundtable*, pages 45–54, South Lake Tahoe, California, October 1999.

[39] S. Smith and C. H. Chen. Commonality analysis for mesh generating systems. Technical Report CAS-04-10-SS, McMaster University, 2004.

[40] S. Smith, J. McCutchan, and F. Cao. Program families in scientific computing. In J. Sprinkle, J. Gray, M. Rossi, and J.-P. Tolvanen, editors, *7th OOPSLA Workshop on Domain Specific Modelling*, pages 39–47, Montreal, Quebec, 2007.

[41] Walid Taha. *Multi-stage programming: its theory and applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[42] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004.

[43] Walid Taha and Michael Florentin Nielsen. Environment classifiers. *SIGPLAN Not.*, 38(1):26–37, 2003.

[44] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[45] Walid Mohamed Taha. *Multistage programming: its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Supervisor-Sheard, Tim.

[46] Todd L. Veldhuizen. Arrays in Blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag.

[47] Todd L. Veldhuizen. Guaranteed optimization for domain-specific programming. In Lengauer et al. [25], pages 307–324.

[48] David M. Weiss. Commonality analysis: A systematic process for defining families. In *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, pages 214–222, London, UK, 1998. Springer-Verlag.