An Algebraic Programming Style for Numerical Software and its Optimization

T.B. Dinesh
Academic Systems Corporation
444 Castro Street, Mountain View, CA 94041, USA
T_Dinesh@academic.com

Magne Haveraaen
University of Bergen
Høyteknologisenteret, N-5020 Bergen, Norway
Magne.Haveraaen@ii.uib.no

Jan Heering
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Jan.Heering@cwi.nl

ABSTRACT
The abstract mathematical theory of partial differential equations (PDEs) is formulated in terms of manifolds, scalar fields, tensors, and the like, but these algebraic structures are hardly recognizable in actual PDE solvers. The general aim of the Sophus programming style is to bridge the gap between theory and practice in the domain of PDE solvers. Its main ingredients are a library of abstract datatypes corresponding to the algebraic structures used in the mathematical theory and an algebraic expression style similar to the expression style used in the mathematical theory. Because of its emphasis on abstract datatypes, Sophus is most naturally combined with object-oriented languages or other languages supporting abstract datatypes. The resulting source code patterns are beyond the scope of current compiler optimizations, but are sufficiently specific for a dedicated source-to-source optimizer. The limited, domain-specific, character of Sophus is the key to success here. This kind of optimization has been tested on computationally intensive Sophus style code with promising results. The general approach may be useful for other styles and in other application domains as well.

1991 Computing Reviews Classification System: D.1.5, D.2.2, J.2

Keywords and Phrases: coordinate-free numerics, object-oriented numerics, algebraic programming style, domain-specific programming style, optimization of numerical code

Note: Submitted to *Scientific Programming*, special issue on *Coordinate-Free Numerics*. This research was supported in part by the European Union under ESPRIT Project 21871 (SAGA—Scientific Computing and Algebraic Abstractions), the Netherlands Organisation for Scientific Research (NWO) under the *Generic Tools for Program Analysis and Optimization* project, and by a computing resources grant from the Norwegian Supercomputer Committee.
Most of the work reported here was done while the first author was at CWI, Amsterdam, The Netherlands, and during the second author's sabbatical stay at the University of Wales, Swansea, with financial support from the Norwegian Science Foundation (NFR).

1 Introduction

The purpose of the Sophus approach to writing partial differential equation (PDE) solvers originally proposed in [12] is to close the gap between the underlying coordinate-free mathematical theory and the way actual solvers are written. The main ingredients of Sophus are:

1. A library of abstract datatypes corresponding to manifolds, scalar fields, tensors, and the like, figuring in the abstract mathematical theory.

2. Expressions involving these datatypes written in a side-effect free algebraic style similar to the expressions in the underlying mathematical theory.

Because of the emphasis on abstract datatypes, Sophus is most naturally combined with object-oriented languages or other languages supporting abstract datatypes. Hence, we will be discussing high-performance computing (HPC) optimization issues within an object-oriented or abstract datatype context, using abstractions that are suitable for PDEs.

Sophus is not simply object-oriented scientific programming, but a much more structured approach dictated by the underlying mathematics. The *object-oriented numerics* paradigm proposed in [8, 23] is related to Sophus in that it uses abstractions corresponding to familiar mathematical constructs such as tensors and vectors, but these do not include continuous structures such as manifolds and scalar fields. The Sophus approach is more properly called *coordinate-free numerics* [13]. A fully worked example of conventional vs. coordinate-free programming of a computational fluid dynamics problem (wire coating for Newtonian and non-Newtonian flows) is given in [11].

Programs in a domain-specific programming style like Sophus may need additional optimization in view of their increased use of expensive constructs. On the other hand the restrictions imposed by the style may lead to new high-level optimization opportunities that can be exploited by dedicated tools. Automatic selection of high-level HPC transformations (especially loop transformations) has been incorporated in the IBM XL Fortran compiler, yielding a performance improvement for entire programs of typically less than 2× [14, p. 239]. We hope Sophus style programming will allow high-level transformations to become more effective than this.

In the context of Sophus and object-oriented programming this article focuses on the following example. Object-oriented languages encourage the use of *self-mutating* (*self-updating*, *mutative*) objects rather than a side-effect free algebraic expression style as advocated by Sophus. The benefits of the algebraic style are considerable. We obtained a reduction in source code size using algebraic notation vs. an object-oriented style of up to 30% in selected procedures of a seismic simulation code, with a correspondingly large increase in programmer productivity and maintainability of the code as measured by the Cocomo technique [4], for instance. On the negative side, the algebraic style requires lots of temporary data space for (often very large) intermediate results to be allocated and subsequently recovered. Using self-mutating objects, on the other hand, places some of the burden of variable management on the programmer and makes the source code much more difficult to write, read, and maintain. It may yield much better efficiency, however. Now, by including certain restrictions as part of the style, a precise relationship between self-mutating notation and algebraic notation may be achieved. Going one step further, we see that the natural way of building a program from high-level abstractions may be in direct conflict with the way current compilers optimize program code. We propose a source-to-source optimization tool, called CodeBoost, as a solution to many of these problems. Some further promising optimization opportunities we have experimented with but not yet included in CodeBoost are also mentioned. The general approach may be useful for other styles and other application domains as well.

This paper is organized as follows. After a brief overview of tensor based abstractions for numerical programming and their realization as a software library (Section 2), we discuss the relationship between algebraic and self-mutating expression notation, and how the former may be transformed into the latter (Section 3). We then discuss the implementation of the CodeBoost source-to-source optimization tool (Section 4), and give some further examples of how software construction using class abstractions may conflict with efficiency issues as well as lead to new opportunities for optimization (Section 5). Finally, we present conclusions and future work (Section 6).

2 A Tensor Based Library for Solving PDEs

Historically, the mathematics of PDEs has been approached in two different ways. The solution-oriented approach uses concrete representations of vectors and matrices, discretisation techniques, and numerical algorithms, while the abstract approach develops the theory in terms of manifolds, scalar fields, tensors, and the like, focusing more on the structure of the underlying concepts than on how to calculate with them (see [15] for a good introduction).

The former approach is the basis for most of the PDE software in existence today. The latter has very promising potential for the structuring of complicated PDE software when combined with template class based programming languages or other languages supporting abstract datatypes. As far as notation is concerned, the abstract mathematics makes heavy use of overloaded infix operators. Hence, user-definable operators and operator overloading are further desirable language features in this application domain. C++ [18] comes closest to meeting these desiderata, but, with modules and user-definable operators, Fortran 90/95 [1, 2] can also be used. In its current form Java [10] is less suitable. It has neither templates nor user-definable operators. Also, Java's automatic memory management is not necessarily an advantage in an HPC setting [16, Section 4]. Some of these problems may be alleviated in Generic Java [6]. The examples in this article use C++.

2.1 The Sophus Library

The Sophus library provides the abstract mathematical concepts from PDE theory as programming entities. Its components are based on the notions of manifold, scalar field and tensor field, while the implementations are based on the conventional numerical algorithms and discretisations. Sophus is currently structured around the following concepts:

- Basic n-dimensional mesh structures. These are like rank n arrays (i.e., with n independent indices), but with operations like $+$, $-$ and $*$ mapped over all elements (much like Fortran 90/95 array operators) as well as the ability to add, subtract or multiply all elements of the mesh by a scalar in a single operation. There are also operations for shifting meshes in one or more dimensions. Parallel and sequential implementations of mesh structures can be used interchangeably, allowing easy porting between architectures of any program built on top of the mesh abstraction.

- Manifolds. These represent the physical space where the problem to be solved takes place. Currently Sophus only implements subsets of R^n.

- Scalar fields. These may be treated formally as functions from manifolds to reals, or as arrays indexed by the points of the manifold with reals as data elements. Scalar fields describe the measurable quantities of the physical problem to be solved. As the basic layer of "continuous mathematics" in the library, they provide the partial derivation operations. Also, two scalar fields on the same manifold may be pointwise added, subtracted or multiplied. The different discretisation methods provide different designs for the implementation of scalar fields. A typical implementation would use an appropriate mesh as underlying discrete data structure, use interpolation techniques to give a continuous interface, and map the $+$, $-$, and $*$ operations directly to the corresponding mesh operations. In a finite difference implementation partial derivatives are implemented using shifts and arithmetic operations on the mesh.

- Tensors. These are generalizations of vectors and matrices and have scalar fields as components. Tensors define the general differentiation operations based on the partial derivatives of the scalar fields, and also provide operations such as componentwise addition, subtraction and multiplication, as well as tensor composition and application (matrix multiplication and matrix-vector multiplication). A special class are the metric tensors. These satisfy certain mathematical properties, but their greatest importance in this context is that they can be used to define properties of coordinate systems, whether Cartesian, axiosymmetric or curvilinear, allowing partial differential equations to be formulated in a coordinate-free way. The implementation of tensors relies heavily on the arithmetic operations of the scalar field classes.

A partial differential equation in general provides a relationship between spatial derivatives of tensor fields representing physical quantities in a system and their time derivatives. Given constraints in the form of the values of the tensor fields at a specific instance in time together with boundary conditions, the aim of a PDE solver is to show how the physical system will evolve over time, or what state it will converge to if left by itself. Using Sophus, the solvers are formulated on top of the coordinate-free layer, forming an abstract, high level program for the solution of the problem.

2.2 Sophus Style Examples

The algebraic style for function declarations can be seen in Figure 1, which shows specifications of some operations for multidimensional meshes, the lowest level in the Sophus library. The mesh class is parameterized by a class `T`, so all operations on meshes likewise are parameterized by `T`. Typical parameters would be a float or scalar field class. The operations declared are defined to behave like pure functions, i.e., they do not update any internal state or modify any of their arguments. Such operations are generally nice to work with and reason about, as their application will not cause any hidden interactions with the environment.

Selected parts of the implementation of a continuous scalar field class are shown in Figure 2. This scalar field represents a multidimensional torus, and is implemented using a mesh class as the main data structure. The operations of the class have been implemented

```
/** returns the mesh circularly shifted {i} positions in dimension {d} */
template<class T> Mesh<T> shift(const Mesh<T> & M, int d, int i);

/** returns the elementwise sum of {lhs} and {rhs}   */
template<class T> Mesh<T> operator+(const Mesh<T>& lhs, const Mesh<T>& rhs);

/** returns the elementwise difference of {lhs} and {rhs}   */
template<class T> Mesh<T> operator-(const Mesh<T>& lhs, const Mesh<T>& rhs);

/** returns the elementwise product of the {lhs} and {r} */
template<class T> Mesh<T> operator*(const Mesh<T>& lhs, const real& r);

...
```

Figure 1: Specification of algebraic style operators on a mesh template class.

as self-mutating operations (Section 3), but are used in an algebraic way for clarity. It is easy to see that the partial derivation operation is implemented by shifting the mesh longer and longer distances, and gradually scaling down the impact these shifts have on the derivative, yielding what is known as a four-point, finite difference, partial derivation algorithm. The addition and multiplication operations are implemented using the element-wise mapped operations of the mesh.

The meshes used in a scalar field tend to be very large. A `TorusScalarField` may typically contain between 0.2 and 2MB of data, perhaps even more, and a program may contain many such variables. The standard translation technique for a C++ compiler is to generate temporary variables containing intermediate results from subexpressions, adding a considerable run-time overhead to the algebraic style of programming. An implementation in terms of self-mutating operators might yield noticeable efficiency gains. For the addition, subtraction and multiplication algorithms of Figure 2 a self-mutating style is easily obtained. The derivation algorithm will require extensive modification, such as shown in Figure 5, with a marked deterioration in readability and maintainability as a result.

3 Algebraic Notation and Self-Mutating Implementation

3.1 Self-Mutating Operations

Let `a`, `b` and `c` be meshes with operators as defined in Figure 1. The assignment

```
c = a * 4.0 + b + a
```

is basically evaluated as

```
temp1 = a * 4.0;
temp2 = temp1 + b;
c     = temp2 + a.
```

This involves the creation of the meshes `temp1`, `temp2`, `c`, the first two of which are temporary. Obviously, since all three meshes have the same size and the operations in question are sufficiently simple, repeated use of a single mesh would have been possible in this case. In fact, for predefined types like integers and floats an

```
/** some operations on a scalar field implemented using the finite difference method
*/
class TorusScalarField {
private:
  Mesh<real> msf();   // data values for each grid point of the mesh
  real delta;         // resolution, distance between grid points

  :

public:

  :

/** 4 point derivation algorithm, computes partial derivative in dimension d */
void uderiv (int d)
{ Mesh<real> ans = (shift(msf,d,1) - shift(msf,d,-1)) * 0.85315148548241;
       ans = ans + (shift(msf,d,2) - shift(msf,d,-2)) * -0.25953977340489;
       ans = ans + (shift(msf,d,3) - shift(msf,d,-3)) * 0.06942058732686;
       ans = ans + (shift(msf,d,4) - shift(msf,d,-4)) * -0.01082798602277;
       msf = ans * (1/delta);
}

/** adding scalar field {rhs} to this TorusScalarField */
void operator+=(const TorusScalarField& rhs);
{ msf = msf + rhs;
}

/** subtracting scalar field {rhs} from this TorusScalarField */
void operator-=(const TorusScalarField& rhs);
{ msf = msf - rhs;
}

/** multiplying scalar {r} to this TorusScalarField */
void operator*=(const real& r);
{ msf = msf * r;
}

  :
}
```

Figure 2: A class `TorusScalarField` with self-mutating implementations of a partial derivation algorithm, a scalar field addition, and a scalar multiplication algorithm. The code itself is using algebraic notation for the mesh operations.

optimizing C or C++ compiler would translate the expression to a sequence of compound assignments[1]

```
c = a; c *= 4.0; c += b; c += a,
```

which repeatedly uses variable `c` to store intermediate results.

We would like to be able to do a similar optimization of the mesh expression above as well as other expressions involving n-ary operators or functions of a suitable nature for user-defined types as proposed in [9]. In an object-oriented language, it would be natural to define self-mutating methods (i.e., methods mutating *this*) for the mesh operations in the above expression. These would be closely similar to the compound assignments for predefined types in C and C++, which return a pointer to the modified data structure. Sophus demands a side-effect free expression style close to the underlying mathematics, however, and forbids *direct* use of self-mutating operations in expressions. Note that with a self-mutating `+=` operator returning the modified value of its first argument, the expression `(a += b) += a` would yield $2(a+b)$ rather than $(2a)+b$.

By allowing the user to define self-mutating operations and providing a way to use them in a purely functional manner, their direct use can be avoided. There are basically two ways to do this, namely, by means of wrapper functions or by program transformation. These will be discussed in the following sections.

3.2 Wrapper Functions

Self-mutating implementations can be made available to the programmer in non-self-mutating form by generating appropriate wrapper functions. We developed a C++ preprocessor SCC doing this. It scans the source text for declarations of a standard form and automatically creates wrapper functions for the self-mutating ones. This allows the use of an algebraic style in the program, and relieves the programmer of the burden of having to code the wrappers manually.

A self-mutating operator `op=` is related to its algebraic analog `op` by the basic rule

$$\texttt{x = y op z;} \equiv \texttt{x = copy(y); x op= z;} \quad (1)$$

or, if the second argument is the one being updated,[2] by the rule

$$\texttt{x = y op z;} \equiv \texttt{x = copy(z); y op= x;} \quad (2)$$

where $\equiv$ denotes equivalence of the left- and right-hand sides, `x`, `y`, `z` are C++ variables, and `copy` makes a copy of the entire data structure. Now, the Sophus style does not allow aliasing or sharing of objects, and the (overloaded) assignment operator `x = y` is always given the semantics of `x = copy(y)` as used in (1) and (2). Hence, in the context of Sophus (1) can be simplified to

$$\texttt{x = y op z;} \equiv \texttt{x = y; x op= z;} \quad (3)$$

and similarly for (2). We note the special case

$$\texttt{x = x op z;} \equiv \texttt{x op= z;} \quad (4)$$

and the obvious generalizations

$$\texttt{x = x op e;} \equiv \texttt{x op= e;} \quad (5)$$

$$\texttt{x = e1 op e2;} \equiv \texttt{x = e1; x op= e2;} \quad (6)$$

[1]Not to be confused with the C notion of compound statement, which is a sequence of statements enclosed by a pair of braces.

[2]This does not apply to built-in compound assignments in C or C++, but user-defined compound assignments in C++ may behave in this way.

```
/** implements the basic mesh operations */
template<class T> class MeshCode1{
      ...
public:
      /** circularly shifts {this} mesh {i} positions in dimension {d} */
      void ushift(int d, int i){ ... }

      /** adds {rhs} elementwise to {this} mesh   */
      void operator+=(const MeshCode1<T> & rhs){ ... }

      /** subtracts {rhs} elementwise from {this} mesh   */
      void operator-=(const MeshCode1<T> & rhs){ ... }

      /** multiplies {this} mesh elementwise by {r} */
      void operator*=(real r){ ... }
      ...
      }
```

Figure 3: The use of self-mutating membership operations for a mesh class `MeshCode1`.

```
template<class T> MeshCode1<T> shift(const MeshCode1<T> & MD, int d, int i)
   { MeshCode1<T> C = MD; C.ushift(d,i); return C; }
template<class T> MeshCode1<T> operator+(const MeshCode1<T>& lhs, const MeshCode1<T>& rhs);
   { MeshCode1<T> C = lhs; C += rhs; return C; }
template<class T> MeshCode1<T> operator-(const MeshCode1<T>& lhs, const MeshCode1<T>& rhs);
   { MeshCode1<T> C = lhs; C -= rhs; return C; }
template<class T> MeshCode1<T> operator*(const MeshCode1<T>& lhs, const real& r);
   { MeshCode1<T> C = lhs; C *= r; return C; }
...
```

Figure 4: Wrapper functions implementing the specification of a mesh using `MeshCode1` operations generated by the SCC preprocessor.

where `e`, `e1`, and `e2` are expressions. SCC uses rules such as (6) to obtain purely functional behavior from the self-mutating definitions in a straightforward way. Figure 4 shows the wrappers created by SCC for the self-mutating mesh operations of Figure 3. The case of n-ary operators and functions is similar ($n \geq 1$). We note that, unlike C and C++ compound assignments, Sophus style self-mutating operators do not return a reference to the variable being updated and cannot be used in expressions. This simpler behavior facilitates their definition in Fortran 90/95 and other languages of interest to Sophus.

The wrapper approach is superficial in that it does not minimize the number of temporaries introduced for expression evaluation as illustrated in Section 3.1. We therefore turn to a more elaborate transformation scheme.

3.3 Program Transformation

Transformation of algebraic expressions to self-mutating form with simultaneous minimization of temporaries requires a parse of the program, the collection of declarations of self-mutating operators and functions, and matching them with the types of the operators and functions actually used after any overloading has been resolved. Also, declarations of temporaries have to be added with the proper type. Such a preprocessor would be in a good position to perform other source-to-source optimizations as well. In fact, this second approach is the one implemented in CodeBoost with promising results.

Figure 5 shows an optimized version of the partial derivation operator of class `TorusScalarField` (Figure 2) that might be obtained in this way. In addition to the transformation to self-mutating form, an obvious rule for `ushift` was used to incrementalize shifting of the mesh.

Assuming the first argument is the one being updated, some further rules for binary operators used in this stage are

```
x op1= e1 op2 e2; ≡
{T t = e1; t op2= e2; x op1= t;}      (7)

{T t1 = e1; s1;}{T t2 = e2; s2;} ≡
{T t = e1; s1; t = e2; s2;}.          (8)
```

Here `x`, `t`, `t1`, `t2` are variables of type `T`; `e1`, `e2` are expressions; and self-mutating operators `op=`, `op1=`, `op2=` correspond to operators `op`, `op1`, `op2`, respectively. Recall that Sophus does not allow aliasing. Rule (7) introduces a temporary variable `t` in a local environment and rule (8) reduces the number of temporary variables by merging two local environments declaring a temporary into a single one.

```
template<class T> void F (T & x)
{ x = x*x + x*2.0; }

template<class T> void P (T & x)
{ T temp1 = x;
  temp1 *= 2.0 ;
  x *= x;
  x += temp1;
}
```

Figure 6: Kernels `F` and `P`.

3.4 Benchmarks

3.4.1 Two Kernels

Consider C++ procedures `F` and `P` shown in Figure 6. `F` computes $x^2 + 2x$ using algebraic notation while `P` computes the same expression in self-mutating form using a single temporary variable `temp1`. Both were run with meshes of different sizes. The corresponding timing results are shown in Figures 7, 8, and 9.

```
/** some operations on a scalar field implemented using the finite difference
    method
*/
public class ScalarField {
  MeshCode1 msf();   // data values for each grid point of the mesh
  real delta;   // resolution, distance between grid points

  :

/** 4 point derivation algorithm, computes partial derivative in dimension d */
public void uderiv (int d)
{ MeshCode1 msa = msf;
  MeshCode1 msb = msf;
  MeshCode1 scratch();

  msa.ushift(d,1);
  msb.ushift(d,-1);
  scratch = msa; scratch.uminus(msb); scratch.umult(0.85315148548241);
  msf = scratch;

  msa.ushift(d,1);
  msb.ushift(d,-1);
  scratch = msa; scratch.uminus(msb); scratch.umult(-0.25953977340489);
  msf.uplus(scratch);

  msa.ushift(d,1);
  msb.ushift(d,-1);
  scratch = msa; scratch.uminus(msb); scratch.umult(0.06942058732686);
  msf.uplus(scratch);

  msa.ushift(d,1);
  msb.ushift(d,-1);
  scratch = msa; scratch.uminus(msb); scratch.umult(-0.01082798602277);
  msf.uplus(scratch);

  msf.umult(1/delta);
}
  :
}
```

Figure 5: Optimized partial derivation operator of class `TorusScalarField` (Figure 2).

Number of elements	Type	SUN Ultra-2							
		No options			Option `-fast`			Optim. speedup	
		NC	NS	NS/NC	OC	OS	OS/OC	NC/OC	NS/OS
$8^3 = 2$kB	F	6.4s	28.4s	4.4	2.8s	4.7s	1.7	2.3	6.0
	P	7.8s	12.2s	1.6	2.8s	2.0s	0.7	1.8	6.1
	F/P	0.8	2.3		1.0	2.4			
$64^3 = 1$MB	F	6.8s	31.7s	4.7	3.2s	8.3s	2.6	2.1	3.8
	P	8.2s	13.4s	1.6	3.2s	3.4s	1.1	2.6	3.9
	F/P	0.8	2.4		1.0	2.4			
$256^3 = 67$MB	F	7.1s	238.5s	33.6	3.5s	199.3s	56.9	2.0	1.2
	P	8.5s	15.6s	1.8	3.5s	18.5s	5.3	2.4	0.8
	F/P	0.8	15.3		1.0	10.8			

Figure 7: Speed of conventional vs. Sophus style on SUN sparc Ultra-2 workstation. More specifically, a *SunOS 5.6 Generic_105181-06 sun4u sparc SUNW, Ultra-2* hardware platform with 512MB internal memory and the SunSoft C++ compiler *CC: WorkShop Compilers 4.2 30 Oct 1996 C++ 4.2* were used.

Number of elements	Type	Silicon Graphics/Cray Origin 2000							
		No options			Option `-Ofast`			Optim. speedup	
		NC	NS	NS/NC	OC	OS	OS/OC	NC/OC	NS/OS
$8^3 = 2$kB	F	3.3s	10.5s	3.2	1.0s	2.3s	2.3	3.3	4.6
	P	4.4s	6.3s	1.4	1.0s	1.3s	1.3	4.4	4.8
	F/P	0.8	1.6		1.0	1.8			
$64^3 = 1$MB	F	3.4s	12.3s	3.6	1.3s	4.1s	3.2	2.6	3.0
	P	4.5s	6.9s	1.5	1.2s	2.3s	1.9	3.8	3.0
	F/P	0.8	1.8		1.1	1.8			
$256^3 = 67$MB	F	4.0s	25.0s	6.3	1.8s	15.6s	8.7	2.2	1.6
	P	5.2s	10.3s	2.0	1.7s	7.0s	4.1	3.1	1.5
	F/P	0.8	2.4		1.1	2.2			

Figure 8: Speed of conventional vs. Sophus style on Silicon Graphics/Cray Origin 2000. More specifically, the Origin 2000 had hardware version *IRIX64 ask 6.5SE 03250013 IP27* with a total of 24GB memory distributed among 128 processors. The C++ compiler used was *MIPSpro Compilers: Version 7.2.1.1m.*

Number of elements	Type	SUN Ultra-2 No options NC	NS	NS/NC	Option -fast OC	OS	OS/OC	Optim. speedup NC/OC	NS/OS
$8^3 = 2$kB	F	6.4s	28.4s	4.4	2.9s	4.7s	1.6	2.2	6.0
	P	7.8s	12.2s	1.6	2.8s	2.0s	0.7	2.8	6.1
	F/P	0.8	2.3		1.0	2.4			
$16^3 = 16$kB	F	6.5s	28.9s	4.4	3.2s	5.2s	1.6	2.0	5.6
	P	7.9s	12.6s	1.6	3.2s	2.5s	0.8	2.5	5.0
	F/P	0.8	2.3		1.0	2.1			
$32^3 = 128$kB	F	6.8s	29.5s	4.3	3.2s	6.2s	1.9	2.1	4.8
	P	8.1s	12.8s	1.6	3.1s	2.7s	0.9	2.6	4.7
	F/P	0.8	2.3		1.0	2.3			
$64^3 = 1$MB	F	6.8s	31.7s	4.7	3.2s	8.3s	2.6	2.1	3.8
	P	8.2s	13.4s	1.6	3.2s	3.4s	1.1	2.6	3.9
	F/P	0.8	2.4		1.0	2.4			

Figure 9: Speed of conventional vs. Sophus style on SUN sparc Ultra-2 workstation for small meshes.

The mesh size is given in the leftmost column. Mesh elements are single precision reals of 4B each. The second column indicates the benchmark procedure (`F` or `P`) or the ratio of the corresponding timings (`F/P`). The columns NC, NS, OC, and OS give the time in seconds of several iterations over each mesh so that a total of 16 777 216 elements were updated in each case. This corresponds to 32 768 iterations for mesh size 8^3, 64 iterations for mesh size 64^3, 1 iteration for mesh size 256^3, and so forth. In columns _C (conventional style) the procedure calls are performed for each element of the mesh, while in columns _S (Sophus style) they are performed as operations on the entire mesh variables.

Columns N_ give the time for unoptimized code (no compiler options), while columns O_ give the time for code optimized for speed (compiler option `-fast` for the SUN CC compiler and `-Ofast` for the Silicon Graphics/Cray CC compiler). The timings represent the median of 5 test runs. These turned out to be relatively stable measurements, except in columns NS and OS, rows 256^3 `F` and `P` of Figure 7, where the time for an experiment could vary by more than 100%. This is probably due to paging activity on disk dominating the actual CPU time. Also note that the transformations done by the optimizer are counterproductive in the `P` case, yielding an NS/OS ratio of 0.8.

When run on the SUN the tests where the only really active processes, while the Cray was run in its normal multi-user mode but at a relatively quiet time of the day (Figure 10). As can be seen the load was moderate (around 58) and although fully utilized, resources where not overloaded.

In the current context, only columns NS and OS are relevant, the other ones are explained in Section 5.1. As expected, the self-mutative form `P` is a better performer than the algebraic form `F` when the Sophus style is used. Disregarding the cases with disk paging mentioned above, we see that the self-mutating mesh operations are 1.8–2.4 times faster than their algebraic counterparts, i.e., the CodeBoost transformation roughly doubles the speed of these benchmarks.

```
IRIX64 ask 6.5SE IP27
load averages: 58.37 57.74 58.30     06:46:21
385 processes: 323 sleeping, 3 stopped,
               1 ready, 58 running
128 CPUs:  0.0% idle, 0.0% usr,  0.0% ker,
           0.0% wait, 0.0% xbrk, 0.0% intr
Memory: 24G max,  23G avail, 709M free,
        25G swap, 17G free swap
```

Figure 10: Random load information for test run on Silicon Graphics/Cray Origin 2000.

3.4.2 Full Application: SeisMod

We also obtained preliminary results on the Silicon Graphics/Cray Origin 2000 for a full application, the seismic simulation code SeisMod, which is written in C++ using the Sophus style. It is a collection of applications using the finite difference method for seismic simulation. Specific versions of SeisMod have been tailored to handle simulations with simple or very complex geophysical properties.[3] We compared a version of SeisMod implemented using SCC generated wrapper functions and a self-mutating version produced by the CodeBoost source-to-source optimizer:

- The algebraic expression style version turned out to give a 10–30% reduction in source code size and greatly enhanced readability for complicated parts of the code. This implies a significant programmer productivity gain as well as a significant reduction in maintenance cost as measured by the Cocomo technique [4], for instance

- A 30% speed increase was obtained after 10 selected procedures out of 150 procedures with speedup potential had been brought in self-mutating form. This speedup turned out to be independent of C++ compiler optimization flag settings.

This shows that although a more user-friendly style may give a performance penalty compared to a conventional style, it is possible to regain much of the efficiency loss by using appropriate optimization tools. Also, a more abstract style may yield more cost-effective software, even without these optimizations, if the resulting development and maintenance productivity improvement is taken into account.

4 Implementation of CodeBoost

CodeBoost is a dedicated C++ source-to-source transformation tool for Sophus style programs. It has been implemented using the ASF+SDF language prototyping system [20]. ASF+SDF allows the required transformations to be entered directly as conditional rewrite rules whose right- and left-hand sides consist of language (in our case C++) patterns with variables and auxiliary transformation functions. The required language specific parsing, rewriting, and prettyprinting machinery is generated automatically by the system from the high-level specification. Program transformation tools for Prolog and the functional language Clean implemented in ASF+SDF are described in [7, 19].

An alternative implementation tool would have been the TAMPR program transformation system [5], which has been used successfully in various HPC applications. We preferred ASF+SDF mainly because of its strong syntactic capabilities enabling us to generate a C++ environment fairly quickly given the complexity of the language.

Another alternative would have been the use of template metaprogramming and/or expres-

[3]SeisMod is used and licensed by the geophysical modelling company UniGEO A.S. (Bergen, Norway).

sion templates [21, 22]. This approach is highly C++ specific, however, and cannot be adapted to Fortran 90/95.

Basically, the ASF+SDF implementation of CodeBoost involves the following two steps:

1. Specify the C++ syntax in SDF, the syntax definition formalism of the system.

2. Specify the required transformation rules as conditional rewrite rules using the C++ syntax, variables, and auxiliary transformation functions.

As far as the first step is concerned, specification of the large C++ syntax in SDF would involve a considerable effort, but fortunately a BNF-like version is available from the ANSI C++ standards committee. We obtained a machine-readable preliminary version [3] and translated it largely automatically into SDF format. The ASF+SDF language prototyping system then generated a C++ parser from it. The fact that the system accepts general context-free syntax rather then only LALR or other restricted forms of syntax also saved a lot of work in this phase even though the size of the C++ syntax taxed its capabilities.

With the C++ parser in place, the required program transformation rules were entered as conditional rewrite rules. In general, a program transformer has to traverse the syntax tree of the program to collect the context-specific information used by the actual transformations. In our case, the transformer needs to collect the declaration information indicating which of the operations have a self-mutating implementation. Also, in Sophus the self-mutating implementation of an operator (if any) need not update *this* but can indicate which of the arguments is updated using the `upd` flag. The transformer therefore needs to collect not only which of the operations have a self-mutating implementation but also which argument is being mutated in each case. As a consequence, CodeBoost has to traverse the program twice: once to collect the declaration information and a second time to perform the actual transformations. Two other issues have to be taken into account:

- C++ programs cannot be parsed before their macros are expanded. Some Sophus-specific language elements are implemented as macros, but are more easily recognized before expansion than after. An example is the `upd` flag indicating which argument of an operator or function is the one to be updated.

- Compared to the total number of constructs in C++, there are relatively few constructs of interest. Since all constructs have to be traversed, this leads to a plethora of trivial tree traversal rules. As a result, the specification gets cluttered up by traversal rules, making it a lot of work to write as well as hard to understand. One would like to minimize or automatically generate the part of the specification concerned with straightforward program traversal.

Our approach to the above problems is to give the specification a two-phase structure as shown in Figure 11. Under the reasonable assumption that the declarations are not spoiled by macros, the first phase processes the declarations of interest prior to macro expansion using a stripped version of the C++ grammar that captures the declaration syntax only. We actually used a Perl script for this, but it could have been done in ASF+SDF as well. It yields an ASF+SDF specification that is added to the specification of the second phase. The effect of this is that the second phase is specialized for the program at hand in the sense that the transformation rules in the second phase can assume the availability of the dec-

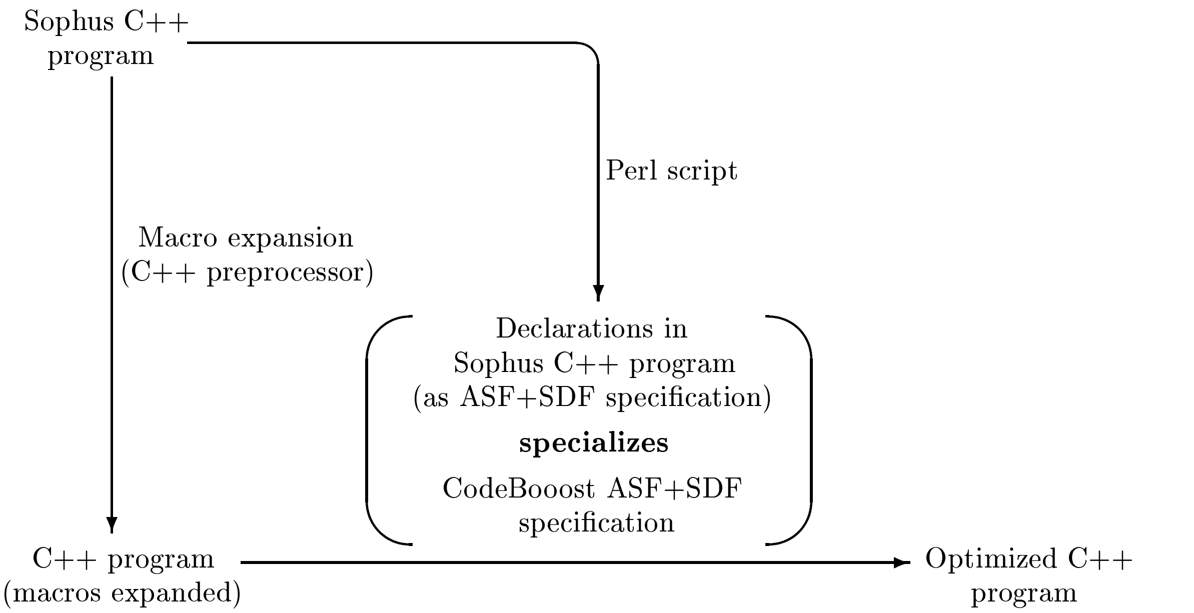

Figure 11: Two-phase specification of CodeBoost.

laration information and thus can be specified in a more algebraic, i.e., context independent manner. As a consequence, they are easy to read, consisting simply of the rules for the constructs that may need transformation and using the ASF+SDF system's built-in innermost tree traversal mechanism. In this way, we circumvented the last-mentioned problem.

As CodeBoost is developed further, it will have to duplicate more and more functions already performed by any C++ preprocessor/compiler. Not only will it have to do parsing (which it is already doing now), but also template expansion, overloading resolution, and dependence analysis. It would be helpful if CodeBoost could tap into an existing compiler at appropriate points rather than redo everything itself. One of the candidates we are considering is the IBM Montana C++ compiler/programming environment [17], which provides an open architecture with APIs giving access to various compiler intermediate representations with pointers back to the source text.

5 Software Structure vs. Efficiency

As noted in Section 1, programs in a domain-specific programming style like Sophus may need additional optimization in view of their increased use of expensive constructs. On the other hand, the restrictions imposed by the style may lead to new high-level optimization opportunities that can be exploited by a CodeBoost-like optimization tool. We give some further examples of both phenomena.

5.1 Inefficiencies Caused by the Use of an Abstract Style

We consider an example. As explained in Section 2.1, scalar field operations like $+$ and $*$ are implemented on top of mesh operations $+$ and $*$. The latter will typically be implemented as iterations over all array elements, performing the appropriate operations pairwise on the elements. For scalar fields, expressions like
$X_1 = A_{1,1} * V_1 + A_{1,2} * V_2$,
$X_2 = A_{2,1} * V_1 + A_{2,2} * V_2$
will force 8 traversals over the mesh data structure. If the underlying meshes are large, this may cause many cache misses for each traversal. Now each of the scalar fields $A_{i,j}$, V_j, and X_j are actually implemented using a mesh, i.e., an array of n elements, and are represented in the machine by `A[i,j,k]`, `V[j,k]` and `X[j,k]` for `k` $= 1, \ldots,$ `K`, where `K` is the number of mesh points of the discretisation. In a conventional implementation this would be explicit in the code more or less as follows:

```
for k := 1,K
    for j := 1,2
        X[j,k] := 0
        for i := 1,2
            X[j,k] += A[i,j,k]*V[j,k]
        endfor
    endfor
endfor
```

It would be easy for an optimizer to partition the loops in such a way that the number of cache misses is reduced by a factor of 8.

In the abstract case aggressive in-lining is necessary to expose the actual loop nesting to the optimizer. Even though most existing C++ compilers do in-lining of abstractions, this would be non-trivial since many abstraction layers are involved from the programmer's notation on top of the library of abstractions down to the actual traversals being performed.

Consider once again the timing results shown in Figure 7, Figure 8, and Figure 9. As was explained in Section 3.4, the procedure calls in columns _C (conventional style) are performed for each element of the mesh, while they are performed as operations on the entire mesh variables in columns _S (Sophus style). Columns OS/OC for row `P` give the relevant figures for the performance loss of optimized Sophus style code relative to optimized conventional style code as a result of Sophus operating at the mesh level rather than at the element level. The benchmarks show a penalty of 1.1–5.3, except for data structures of less than 128kB on the SUN, where a speedup of up to 1.4 (penalty of 0.7) can be seen in Figure 9. As is to be expected, for large data structures the procedure calls in column OC are more efficient than those in column OS, as the optimizer is geared towards improving the conventional kind of code consisting of large loops with procedure calls on small components of data structures. Also, cache and memory misses become very costly when large data structures have to be traversed many times.

The figures for `P` in column OS of Figure 9 are somewhat unexpected. In these cases OS is the fastest alternative up to a mesh size somewhere between 32^3 and 64^3. This may be due to the smaller number of procedure calls in the OS case than in the OC case. In the latter case `F` and `P` are called once per element, i.e., 16 777 216 times, while in the OS case they are called only once and the self-mutating operations are called only 4 times.

Another interesting phenomenon can be seen in column NC of Figure 7 and Figure 8. Here the self-mutating version takes longer than the algebraic version, probably because the compiler automatically puts small temporaries in registers for algebraic expressions, but cannot do so for self-mutating forms. The OC column shows that the optimizer eliminates the difference.

5.2 New Opportunities for Optimization

The same abstractions that were a source of worry in the previous section at the same time provide the specificity and typing making the use of high-level optimizations possible. Before they are removed by inlining, the information the abstractions provide can be used to select and apply appropriate datatype specific optimization rules. Sophus allows application of such rules at very high levels of abstraction. Apart from the expression transformation rules (1)–(8) (Section 3), which are applicable to a wide range of operators and functions, further examples at various levels of abstraction are:

- The laws of tensor algebra. In Sophus the tensors contain the continuous scalar fields as elements (Section 2.1), thus making the abstract tensor operations explicit in appropriate modules.
- Specialization of general tensor code for specific coordinate systems. A Cartesian coordinate system gives excellent simplification and axiosymmetric ones also give good simplification compared to general curvilinear code.
- Optimization of operations on matrices with many symmetries. Such symmetries offer opportunities for optimization in many cases, including the seismic modelling application mentioned in Section 3.4.2.

6 Conclusions and Future Work

- The Sophus class library in conjunction with the CodeBoost expression transformation tool shows the feasibility of a style of programming PDE solvers that attempts to stay close to the abstract mathematical theory in terms of both the datatypes and the algebraic style of expressions used.
- Our preliminary findings for a full application, the Sophus style seismic simulation code SeisMod, indicate significant programmer productivity gains as a result of adopting the Sophus style.
- There are numerous further opportunities for optimization by CodeBoost in addition to replacement of appropriate operators and functions by their self-mutating versions. Sophus allows datatype specific rules to be applied at very high levels of abstraction.

Acknowledgments

Hans Munthe-Kaas, André Friis, Kristin Frøysa, Steinar Søreide, and Helge Gunnarsli have contributed to Sophus in various ways.

References

[1] J. C. Adams, W. S. Brainerd, and J. T. Martin. *Fortran 90 Handbook: Complete ANSI/ISO Reference.* Intertext Publications, 1992.

[2] J. C. Adams, W. S. Brainerd, J. T. Martin, and B. T. Smith. *Fortran 95 Handbook: Complete ISO/ANSI Reference.* MIT Press, 1997.

[3] AT&T Research. *C++ Syntax—RFC Version*, 1996.

[4] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[5] J. M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In T. J. Biggerstaff and

A. J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.

[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, 1998.

[7] J. J. Brunekreef. A transformation tool for pure Prolog programs. In J. P. Gallagher, editor, *Logic Program Synthesis and Transformation (LOPSTR '96)*, volume 1207 of *Lecture Notes in Computer Science*, pages 130–145. Springer-Verlag, 1996.

[8] K. G. Budge, J. S. Peery, and A. C. Robinson. High-performance scientific computing using C++. In *USENIX C++ Technical Conference Proceedings*, pages 131–150. USENIX Association, August 1992.

[9] T. B. Dinesh. Extending compound assignments for C++. *OOPS Messenger*, 3(1):45–49, 1992.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[11] P. W. Grant, M. Haveraaen, and M. F. Webster. Tensor abstraction programming of computational fluid dynamics problems. Technical Report CSR 3–98, Department of Computer Science, University of Wales, Swansea, 1998.

[12] M. Haveraaen, V. Madsen, and H. Munthe-Kaas. Algebraic programming technology for partial differential equations. In A. Maus et al., editors, *Proceedings Norsk Informatikk Konferanse (NIK '92)*, pages 55–68, 1992.

[13] H. Munthe-Kaas and M. Haveraaen. Coordinate free numerics—closing the gap between 'pure' and 'applied' mathematics? *ZAMM Z. angew. Math. Mech.*, 76(S1):487–488, 1996. (Proceedings ICIAM/GAMM '95).

[14] V. Sarkar. Automatic selection of high-order transformations in the IBM XL Fortran compiler. *IBM J. Res. Develop.*, 41:233–264, 1997.

[15] B. Schutz. *Geometrical Methods of Mathematical Physics.* Cambridge University Press, 1980.

[16] S. K. Singhal et al. Building high-performance applications and servers in Java: An experiential study. Technical report, IBM, 1997. URL http://-www.ibm.com/java/education/javahipr.-html.

[17] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in Montana. Technical Report RC 20770, IBM Research Division, March 1997.

[18] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3d edition, 1997.

[19] M. G. J. van den Brand, S. M. Eijkelkamp, D. K. A. Geluk, Meijer, H. R. Osborne, and M. J. F. Polling. Program transformations using ASF+SDF. In *Proceedings of ASF+SDF '95*, Technical Report P9504, pages 29–52. Programming Research Group, University of Amsterdam, 1995.

[20] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

[21] T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[22] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In Y. Ishikawa et al., editors, *Scientific Computing in Object-Oriented Parallel Environments (ISCOPE '97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 49–56. Springer-Verlag, 1997.

[23] M. K. W. Wong, K. G. Budge, J. S. Peery, and A. C. Robinson. Object-oriented numerics: A paradigm for numerical object-oriented programming. *Computers in Physics*, 7(5):655–663, 1993.