Case study on algebraic software methodologies for scientific computing[1]

Magne Haveraaen
Department of Informatics, University of Bergen, P.O. Box 7800 N-5020 BERGEN, Norway

The use of domain specific languages and appropriate software architectures are currently seen as the way to enhance reusability and improve software productivity. Here we outline a use of algebraic software methodologies and advanced program constructors to improve the abstraction level of software for scientific computing. This leads us to the language of coordinate free numerics as an alternative to the traditional coordinate dependent array notation.

This provides the backdrop for the three accompanying papers: *Coordinate Free Programming of Computational Fluid Dynamics Problems*, centered around an example of using coordinate free numerics, *Machine and Collection Abstractions for User-Implemented Data-Parallel Programming*, exploiting the higher abstraction level when parallelising code, and *An Algebraic Programming Style for Numerical Software and its Optimization*, looking at high-level transformations enabled by the domain specific programming style.

1. Introduction

Scientific computing is the use of computers to solve and investigate problems from the sciences. It has all along been the driving force for high performance computing (HPC). The problems range from safety related problems, such as weather forecasting and bridge strength analysis, via exploration of the physical world around us, such as particle physics and seismics, to solving manufacturing problems, such as molding and coating of materials. The process that leads to a solution of such problems starts with a mathematical model, typically in the form of a partial differential equation (PDE), developed by a physicist, chemist or other domain specialist. The mathematical formulation is then studied and transformed by applied mathematicians or numerical analysts into data structures and algorithms. These are expressed as computer programs, whose computations are used to investigate the original problem. The use of computers, rather than experiments, to investigate problems has given rise to the term *computational science*, and is becoming a very important field in scientific computing. The production of software plays an important role in this. Quite early it was recognised that if the mathematical and the programming notations could be made as closely related as possible, then the process of writing computational software would be eased. This is expressed clearly by Kenneth E. Iverson in [35, preface page vii]:

> Applied mathematics is largely concerned with the design and analysis of explicit procedures for calculating the exact or approximate values of various functions. Such explicit procedures are called algorithms or *programs*. Because an effective notation for the description of programs exhibits considerable syntactic structure, it is called a *programming language*.
>
> Much of applied mathematics, particularly the more recent computer-related areas which cut across the older disciplines, suffers from the lack of an adequate programming language. It is a central thesis of this book that the descriptive and analytic power of an adequate programming language amply repays the considerable effort required for its mastery.

Since then the "toolbox" of both mathematics and computer science has been considerably expanded. Of special importance to us is the maturing of algebraic ideas in the form of the mathematical disciplines of *universal algebra*, the generalisation of algebra to any mathematical structure, and *category theory*, the study of structure and structural relationships. *Algebraic software methodologies* constitutes the conscious application of these in the software process.

Seeing software as a formal entity, we extend the observation of Iverson and emphasise the following notions in the formulation of software systems.

[1]Much of this work was performed during the author's sabbatical visit to the University of Wales Swansea, with financial support from the Research Council of Norway (NFR).

Scientific Programming 8 (2000) 261–273
ISSN 1058-9244 / $8.00

– Domain analysis and domain oriented languages: When we need to understand and discuss a problem domain, whether it is for requirements specification or the formulation of programs to solve problems, we need to have words for the concepts of the domain. Universal algebra and algebraic specifications are tools for precisely identifying such concepts. The domain oriented concepts form the basis for domain specific languages [34], giving us the vocabulary needed to discuss the domain.
– "Programming in the small": This is about how to build data structures and algorithms in order to form software. The basic program constructors for this are grouping, choice and repetition. Combining this with encapsulation of data structures and algorithms to form abstract data types and classes, provides means to produce software pieces that implement the domain oriented concepts and thus build domain specific programming languages.
– "Programming in the large" – software architectures: This concerns how our software pieces are to be organised in order to form the software we want. The focus is on what software pieces we need, and how they relate to each other in order to provide the appropriate domain oriented concepts. Choices here have a large influence on how to build software libraries and the flexibility and adaptability of the resulting software. Structure analysis using category theory related concepts provides valuable feedback on this.

Much information on structuring software is now systematised in the notions of *design frameworks* or *design patterns* [18,49]. These capture recurring design decisions at various levels of software design, but are based on practical experience rather than formal analysis – which we promote here.

When building a software system we want to analyse the problem in such a way as to reduce the cost of the software process, yet achieve optimal efficiency of the developed programs. Experience has shown, and this was formalised already in the COCOMO cost estimation model [9], that there is a positive correlation between software source code size and the development and maintenance costs. In fact, costs tend to increase more than linearly with software code size, i.e., the size of the code we need to "worry about" when developing or maintaining a software system. Bringing down the size of this code hence brings down overall costs. Software architecture and language concepts affect source code size more than many other considerations [19].

Algebraic software methodologies are useful as key players in both the selection of problem domain concepts and the selection of software architectures. Using these tend to raise the abstraction level of the concepts, allowing us to express ourselves more succinctly. The choice of domain specific language strongly influences how the original problem should be phrased and analysed. This is discussed in [23,29] for the domain of PDEs. As an alternative to linear algebra and coordinate based array notation, the traditional language of applied mathematics, there also exists a coordinate free language for PDEs [51]. We have developed a software library framework, Sophus,[2] embodying this. Coordinate free numerical software combines conceptual notions from several disciplines [45], but it turns out that the coordinate free language has many advantages when structuring numerical software. The Sophus framework can be implemented and used in any programming language with abstract data types or classes and objects. Our approach uses object-orientation, and we are thus doing object oriented numerics (OON) [3, 61]. Our focus on coordinate free numerics distinguishes us from most of the OON methods.

Here we present the background and rationale for the three accompanying papers, which all relate to the Sophus framework:

– [23] compares, using a case study from fluid dynamics, software properties for the traditional and the coordinate free domain specific languages.
– [28] presents an abstraction based data parallel programming approach.
– [15] discusses domain specific optimisation related to the higher abstraction level of the coordinate free language.

This paper is organised as follows. In section two we present universal algebra as domain analysis tools and apply them to the PDE problem domain. Section three discusses programming in the small. Then we focus on software architectures, and present the Sophus architecture for PDE software. In section five we discuss some new problems that arise from this more abstract approach to numerical software. Finally we summarise our basic approach.

[2] Named after the Norwegian mathematician Sophus Lie (1842–1899).

2. Domain analysis and domain concepts

Discovering the right concepts needed for a domain specific language is a difficult task. It requires insight into the domain, an ability to pinpoint the crucial concepts, and deciding how to express them in programming language terms. In Section 2.1 we claim that domain concepts may be presented as sorts with operations, which allow them to be embedded in general programming languages. Universal algebra provides means for such a presentation. Algebraic specifications lets us investigate concepts and their meaning before committing ourselves to them. We show this for the PDE domain in Section 2.2 where we present the coordinate based and the coordinate free languages. Validation, i.e., a check that the concepts are suited to precisely formulate concrete problems is discussed in Section 2.3.

2.1. Domain analysis tools: Universal algebra and algebraic specifications

Universal algebra introduces a distinction between syntax (signature) and semantics (models) akin to the distinction between declaration and implementation in a programming language. A plain, many-sorted *signature* Σ declares a set of *sort names* s_j and function symbols $f_i : s_{i_1}, \ldots, s_{i_m} \rightarrow s_{i_{m+1}}$, where $s_{i_1}, s_{i_2}, \ldots, s_{i_m}$ for $m \geqslant 0$ are the argument sorts and $s_{i_{m+1}}$ is the result sort (including the case $m = 0$ for a constant). We may think of a sort as a type name or a class name in programming language terms. A function symbol corresponds to a side-effect free function or typed method. A procedure or method that changes the program state (has side-effects) can be decomposed into one or more side-effect free functions. The application of the procedure can then be formulated as side-effect free function calls with explicit assignments to program variables. A signature corresponds to an idealised declaration section of a program.

Mathematically a *model* or *algebra* A for a signature Σ defines for each sort s_j of Σ a mathematical set $A(s_j)$, called the carrier, and for each function symbol $f_i : s_{i_1}, \ldots, s_{i_m} \rightarrow s_{i_{m+1}}$, a total mathematical function $A(f_i) : A(s_{i_1}) \times \cdots \times A(s_{i_m}) \rightarrow A(s_{i_{m+1}})$. In a programming context we may let an $A(s_j)$ be a data structure. Likewise, $A(f_i)$ may denote an algorithm which defines a computable function from its argument data values to its result data values.

A *specification* restricts the class of allowable models for a signature. *Algebraic specification* is a technique which has gained some momentum. It only focuses on the properties that we want satisfied, rather than devising specific model constructions. Thus it is a rather abstract approach, but it permits both mathematical models and programming language oriented models. The former are important in the mathematical PDE domain, the latter in the software domain. A specification may be extended (often referred to as *interface inheritance*) and combined with other specifications in various ways to form a new specification [50]. Given a signature Σ and a specification, we may ask whether an implementation satisfies the specification. Already in [42] this problem was addressed in a clear way, and a software development technique taking this into account was proposed. This technique is currently known as *programming by contract* [40] and object-oriented programming languages provide sufficient support to utilise this approach.

Many algebraic specification languages have been developed. Some of the more evolved ones are LSL (the Larch Shared Language [24,25]) and ASF+SDF (Algebraic Specification Formalism and Syntax Definition Formalism [6]). At present there is a pan-European effort in creating a standard algebraic specification language CASL [44]. A powerful toolbase supporting CASL is expected to appear over the next few years.

2.2. Scientific computing domain concepts

Applied mathematics is concerned with studying the models needed for scientific computing. The language of applied mathematics has traditionally been concerned with operations on real or complex numbers. Linear algebra is one of its important subdisciplines, where the notions of vectors and linear mappings in the form of (multidimensional) matrices are prominent in a coordinate dependent language for PDEs. Early attempts at domain specific programming languages for this application area, such as APL [35] and Fortran [17], naturally took these as a starting point. They defined abstractions for real and complex numbers, and provided the array construct[3] as basis for implementations of vectors and matrices. Arrays were also used to store the grid points of discretisations such as the finite difference or finite element methods.

[3] There is an interesting difference though: while APL treated arrays as collection-oriented abstractions, providing operations on the whole collection, Fortran only provided operations on the individual array elements. This did not change until matrix operations finally where introduced in Fortran-90 [1].

If we start investigating the problem domain concepts using algebraic methods, we will of course rediscover the basic structures of algebra:[4] monoid, group, ring, field, vector space, linear mappings (matrices), tensors (which generalise rings, fields, vectors, matrices and (multi)linear mappings), etc. Algebraic specifications of many of these concepts for a problem domain analysis is in [30]. As an example, a ring R has binary operations $+$ (addition), $-$ (subtraction) and $*$ (multiplication), and constants 0 (zero) and 1 (one). These form the *ring signature*,

$$
\begin{aligned}
&+ : R, R \to R,\\
&- : R, R \to R,\\
&* : R, R \to R,\\
&0 : \to R,\\
&1 : \to R.
\end{aligned}
$$

An *algebraic specification* of a ring could be the following, where a, b, c range over all ring elements R.

$$(a + b) + c = a + (b + c), \tag{1}$$

$$a + b = b + a, \tag{2}$$

$$(a * b) * c = a * (b * c), \tag{3}$$

$$(a + b) - b = a, \tag{4}$$

$$0 + a = a, \tag{5}$$

$$1 * a = a, \tag{6}$$

$$a * 1 = a, \tag{7}$$

$$(a + b) * c = (a * c) + (b * c), \tag{8}$$

$$a * (b + c) = (a * b) + (a * c). \tag{9}$$

Here we see that any *model* for a ring must obey the laws that addition is associative Eq. (1) and commutative Eq. (2), subtraction is the inverse of addition Eq. (4), and multiplication is associative Eq. (3) and distributes over addition in the familiar way Eqs (8)–(9). Further, the neutral element with respect to addition is denoted by 0 Eq. (5) and with respect to multiplication by 1 Eqs (6)–(7).

Continuing such an analysis for the PDE domain we note that a basic assumption is that every spatial point in the physical world can be represented by an element of a set $\mathcal{M}$ called a *manifold*. The physical properties are then ascribed to each point in the form of a *value field*, akin to a function from the manifold to some value domain. If the values at each point are reals (one of the many models for a ring) they are said to form a scalar field. If they are vectors they are said to be vector fields, tensors give rise to tensor fields, etc. Now a value field lifts properties of the value elements to the field. All equationally defined property are retained, so a scalar field is a ring, a vector field is a vector, a tensor field is a tensor, etc. If the manifold in addition has sufficient structure, at least a notion of proximity and direction, we may define partial differentiation operators on the value fields, provided the value fields are smooth enough.

This analysis gives us a domain specific language which is coordinate free [51] with concepts close to those of pure mathematics. This language is equally valid as a domain specific language as a language based on the traditional concepts of applied mathematics.

2.3. Validating the domain concepts

To validate a domain specific language we need to check that it is useful, i.e., that we may use it to express problems and solutions in our domain. A time dependent PDE provides a relationship between spatial derivatives of tensor fields (which represent physical quantities) 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. Solvers for time independent PDEs are similar.

The coordinate free language we sketched allows a PDE to be formulated abstractly and simply, independently of coordinate systems and number of dimensions of the problem. For example, the elastic wave equation may be written as

$$
\begin{aligned}
&\rho \frac{\partial^2 \vec{u}}{\partial t^2} = \nabla \cdot \sigma + \vec{f}(t),\\
&\sigma = \Lambda(e),\\
&e = \mathcal{L}_{\vec{u}}(g).
\end{aligned}
\tag{10}
$$

This formulation is valid for 1, 2 and 3 dimensions, independently of coordinate system (cartesian, cylindrical, curvilinear, ...). Here ρ is the density scalar field and Λ the stiffness tensor field, both are given data for the physical domain, $\vec{u}$ is the time varying particle displacement vector field which represents the propagation of the elastic wave, the tensor field g defines the

[4] It was the investigation and generalisation of these concepts that led to the discovery of universal algebra [58].

coordinate system used, the tensor fields σ (stress) and e (strain) are computed intermediate values, and $\vec{f}(t)$ is a time-varying vector field representing the forces which initiate the wave. The $\nabla\cdot$ and $\mathcal{L}_{\vec{u}}$ are derivation operators, the latter dependent on the displacement $\vec{u}$. It follows that the coordinate free language allows us to express interesting problems from the PDE domain, as required for validation. See [29] for a discussion of the software properties of this formulation,

In comparison the traditional component based formulation of the elastic wave equation may look like

$$
\begin{aligned}
\rho\frac{\partial^2 u_1}{\partial t^2} &= \frac{\partial \sigma_{11}}{\partial z} + \frac{\partial \sigma_{12}}{\partial x} + f_1(t),\\
\rho\frac{\partial^2 u_2}{\partial t^2} &= \frac{\partial \sigma_{12}}{\partial z} + \frac{\partial \sigma_{22}}{\partial x} + f_2(t),\\
\sigma_{11} &= (\lambda_1 + 2\lambda_2)e_{11} + \lambda_1 e_{22},\\
\sigma_{22} &= (\lambda_1 + 2\lambda_2)e_{22} + \lambda_1 e_{11},\\
\sigma_{12} &= 2\lambda_2 e_{12},\\
e_{11} &= \frac{\partial u_1}{\partial z},\\
e_{22} &= \frac{\partial u_2}{\partial x},\\
e_{12} &= \frac{1}{2}\left(\frac{\partial u_1}{\partial x} + \frac{\partial u_2}{\partial z}\right).
\end{aligned}
$$

This is formulated for two dimensions (denoted z- and x-directions), cartesian coordinate system, assuming an isotropic medium. Here u_1 and u_2 are the components of $\vec{u}$, λ_1 and λ_2 are the two distinct components of Λ for an isotropic medium, σ_{11}, σ_{12} and σ_{22} are the distinct components of σ (which is symmetric), e_{11}, e_{12} and e_{22} are the distinct components of e (which is also symmetric), and f_1 and f_2 are the components of the function $\vec{f}$. The derivation operators have been given directly in terms of their partial derivatives.

Further examples with a discussion of the difference between these two PDE domain languages is in the accompanying paper [23].

3. Programming in the small

To make a domain specific language into a tool for programming a computer, its concepts need to be realised in a programming language. Here we will first look at program constructors to build data structures and algorithms, allowing us to express all computable functions. Then we will look at the abstraction mechanisms needed to provide the domain oriented concepts in a programming language. In Section 3.3 we discuss parallel programming and relate it to the use of abstractions. Finally we discuss the implementation of the PDE problem domain concepts.

3.1. Program construction and reasoning

In order to build programs we need algorithm and data constructors, confer the slogan *Algorithms + Data Structures = Programs* [59]. Most of the early programming languages were strictly following the von Neumann machine model, i.e., sequential imperative programming languages. Imperative languages place the burden of organising instruction sequencing (choice, loops) and storage use (variable updates, arrays and pointers) on the programmer. It was soon argued that this was too restricting, and that semantical notions, and hence reasoning tools, for imperative languages were overly complex. Backus proclaimed his FP (functional programming) language as a reaction to this [4].

Functional languages provide the programmer with expressions and recursion as the primary tool for writing algorithms, tuples and recursive types (lists and trees) for organising data. These are less error prone than their imperative counterparts. Unfortunately, in spite of this simplicity, recursively formulated algorithms often have an exponential growth in space and time complexity, while their imperative counterparts may be linear in time and constant in space. Memoisation (caching of computed function values) may help in avoiding unnecessary recomputation, but does not guarantee good usage of storage, and premature purging of the cache may be needed [16]. The lack of execution time efficiency, coupled with lack of storage control, has hitherto proven detrimental to the use of functional programming for high performance computing, in spite of several attempts such as those in [22,54]. The constructive recursive approach with programmer defined explicit data dependency may be a way out of this [14,27,31].

A general belief is that functional languages have a simpler semantics than imperative languages, thus that they are more akin to reasoning and manipulation. However, it was shown already in [43], that with simple syntactic restrictions, imperative languages can be made just as semantically simple. This allows program reasoning and transformation to become just as easy in an imperative context as in a functional context.

3.2. Abstraction mechanisms

Program construction becomes more flexible if it is possible to abstract over algorithms and types. Algorithmic abstraction, i.e., naming algorithms by function symbols, is often called procedural abstraction, or `SUBROUTINE` in Fortran. Its use has been a great success for scientific computing in the form of large numerical libraries. Type abstraction, i.e., naming data structures as sorts, is likewise useful by itself. Algorithmic and type abstraction taken together gives the notion of data abstraction – *abstract data types* or, in object-oriented terminology, *classes*. This couples the constructed models (data structures and algorithms) to the signature (sorts and function symbols) of the domain concepts. With encapsulation [47] data abstraction becomes a means of realising domain concepts as atomic elements in a general purpose programming language, thus tailoring it as a domain specific programming language. How this can be achieved and shown to be correct was demonstrated already in [33,42]. The use of *data invariants*, properties that are to be invariantly true on the data, is very important for this. The data invariant reduces the state space of the data abstraction, making the operations easier to implement, and making the relationship with the mathematical models of a specification clearer.

In some cases hardware may provide an interesting feature, such as parallelism, while a general purpose programming language does not acknowledge it. Then it is normally possible for the user to define a programming language interface to the feature, providing it as an abstract data type. The hardware feature will normally not be generally available, but an implementation of the abstract data type in the programming language can emulate its overall functionality, providing access to the abstraction irrespectively of the hardware platform used. Thus abstractions make it feasible to utilise advanced hardware much quicker than waiting for improvements in compilers or programming languages.

Abstract data types lets us abstract hardware features or software constructions, relating them to the sorts and function symbols resulting from of our algebraic analysis of the problem domain. We thus have the ability to get implementations of the domain specific languages we find useful. With such abstractions, a program may be expressed at an arbitrarily high abstraction level.

Often the use of (many layers of) abstractions reduce the run-time efficiency of code. Partly because current compilers introduce extra instructions when calling procedures due to an overly complex procedure call semantics, partly because certain short-cuts that are possible on low-level code would cut through non-related high-level abstractions. With syntactic restrictions the semantical notions of a programming langauge may be simplified, see Section 3.1, enabling an algebraic program transformation tool like CodeBoost, described in the accompanying paper [15], to optimise code by taking advantage of high-level properties of source code as well as cutting through abstraction layers to introduce low-level short-cuts in the code and avoid procedure call overhead.

3.3. Data dependencies and parallelisation

In order to achieve higher performance for computational modeling, machines with multiple processors have been taken into use. Their utilisation apparently requires either a move away from the von Neumann programming approach by the use of functional languages or by explicit parallel constructs in the code, or compilers that analyse and parallelise sequential imperative code. Functional programming has had some success for advanced parallel programming, see [54] for a collection of approaches and [38] for a recent overview. The use of explicit parallel constructs turns out to be very difficult to program, and is in general discouraged. Letting the compiler analyse the code for dependencies and then generate the parallelism sounds ideal, but turns out to be an NP-complete problem in general [39], although quite a lot may be achieved in practice. Compiler analysis may look at the flow of control (control dependence analysis) giving a coarse grain parallelisation. Or it may look at data dependencies, i.e., how data at one point during program execution depends on data at another point of the program execution, giving rise to fine grain parallelism. Good automatic analysis is only possible for programs with a simple semantical structure. This is typical of Fortran programs, where high performance often is essential, but this information is also clearly visible in functional programming. Many of the more advanced compiler optimisation strategies were discovered for functional programming languages, such as Crystal [13] and later adapted for imperative languages [60].

One step further is to make the data dependencies explicit abstractions in the programs. This approach has been shown in a functional programming context [14, 26,31] and in an imperative context in [12,14]. In general there will be a gap between the data dependencies in a program and the communication patterns of the hardware. This gap has to be filled either by the

compiler, a problem which again is NP-complete in general [39], or closed explicitly by the programmer, as in [14]. Making data dependencies available as an explicit structure has the added benefit that space and time considerations may be fully controlled by the programmer, both for sequential and parallel compilation, breaking the exponential execution time barrier often associated with functional programming.

Another important observation is that when using collection oriented abstractions, such as arrays, in sequential code, these collections may be distributed in parallel on processors [8]. This is the basis for data parallel programming [48], which has been coupled with abstraction oriented programming in the accompanying paper [28], giving direct access to this hardware feature as a data abstraction.

3.4. Implementing the coordinate free language

Showing that the domain oriented concepts can be implemented is a final assurance of their usefulness. It guarantees that the concepts actually are computable, and thus will be useful in the solution of the problems. To check that our coordinate free language is realisable we will sketch the implementation of two of the key abstractions, the scalar fields and the tensors.

Numerical discretisation methods (finite difference, finite element, etc.) make it possible to represent the scalar fields over an infinite set $\mathcal{M}$, the manifold, by a finite approximation. Typically we store data values for carefully selected grid points in large array data structures. The discretisation has to provide ring operations and partial differentiation operations by performing computations on the stored data.

The tensor is where coordinate systems are handled and the more advanced differentiation operations are implemented. Tensors are typically represented as multidimensional arrays, with appropriate, well-known algorithms for the tensor operations. These algorithms only require that the array elements are ring structures.

These models can easily be built using the standard program constructors. This is well-known from the use of the traditional, coordinate based language for the PDE domain, which only uses the basic types and constructors of programming languages. With data abstraction we may couple this together to create the coordinate free language.

4. Programming in the large: software architectures

In our short overview we have sketched an analysis of the PDE domain that provided us with PDE domain specific languages, the traditional coordinate based language and the coordinate free language, with algebraic specifications of the concepts. The languages were shown adequate to formulate interesting problems from the domain (Section 2.3). We have also sketched that the concepts of these domain languages may be implemented using standard data structures and algorithms. Here we will study how to organise the concepts of the more abstract coordinate free language as a software library, i.e., study software architectures for such a library. In this analysis we also need to consider the issue of developing both sequential and parallel versions of the software.

A good software architecture is achieved if we minimise the number of distinct library components (*packages*), and the software complexity[5] of each. How we combine the packages to achieve the problem domain specific concepts will be a blue-print for *configuring* (i.e., putting together) the application programs from the packages. Good choices here will greatly reduce the software development effort. Both by directly reducing our coding effort, and, more importantly, by identifying reusable components for other tasks within the same problem domain. In the algebraic specification language CASL software architecture can be defined explicitly [7].

First we will introduce the notions of categories and functors, precise mathematical notions for the study of structure. Then we will use these tools to investigate an architecture for the coordinate free language. Lastly we present the Sophus library framework which builds on this insight.

4.1. Algebraic structuring concepts

A collection of related mathematical structures, such as the data structures of a programming language, typically form a category [20]. A *category* $\mathbf{C}$ is a collection of objects $A, B, \ldots$, and morphisms $f : X \to Y$ from objects X to Y, with an associated associative composition rule $\circ$ on morphisms and a neutral morphism

[5] Software complexity is a measurement of the complexity of the program text, as opposed to space and time complexity which refer to run-time properties of the software. Software complexity correlates to the cost of developing and maintaining software.

(with respect to $\circ$) for each object. We will use the categories **Prog** and **Set** as our examples. The category **Set** is from mathematics. It has sets as objects and total functions between sets as morphisms. In **Prog** the objects are data structures, and the morphisms are all side-effect free algorithms from a data structure to a data structure. The identity morphism and composition rule for morphisms in these categories should be obvious. Functions of more than one argument are defined from special objects called *product objects* in the category. A comprehensive introduction to category theory may be found in [32], while [56] is a lighter, more intuitive introduction.

Categories are related by *functors*, functions between categories. A functor $F : \mathbf{C} \to \mathbf{D}$, from category $\mathbf{C}$ to category $\mathbf{D}$, maps objects to objects and morphisms to morphisms such that identities and compositions are preserved. It is not to hard to find a functor from **Prog** to **Set** that takes the data structures and algorithms (for some specific computer) to the sets of values and mathematical functions being computed.

Functors (on objects) are in many ways like C++ template classes [53] or Ada generic packages [5]. These mechanisms will take a data type as argument and define a new data type based on it. We may for instance define a generic `list` package with a type parameter, such that whenever we instantiate the package with a data structure D, we get a data structure `list of` D. The functor version of a data type constructor has some additional properties. A list constructing functor $L : \mathbf{Prog} \to \mathbf{Prog}$ takes a data structure D and returns a `list of` D data structure $L(D)$. But in addition to defining the list data type, it will take any function $f : D \to E$ and define an *iterator function* $L(f) : L(D) \to L(E)$. When $L(f)$ is given a `list of` D as argument it will perform f on every element of that list, returning a `list of` E of the results. Likewise we may treat array data structure constructors as functors. For every index type I we have an array constructing functor $A_I : \mathbf{Prog} \to \mathbf{Prog}$ which takes an object E and defines `array` I `of` E, the array structure with elements of type E. But we also get the iterator functions. Given for instance a binary operation $+ : E \times E \to E$ we have $A_I(+) : A_I(E) \times A_I(E) \to A_I(E)$ which adds, componentwise, i.e., for each index $i \in I$, the elements of the two argument arrays, yielding a new array with the summed values. The functor mechanism preserves equational properties of the argument E for $A_I(E)$. If E is a ring, such as the reals $\mathcal{R}$, then $A_I(E)$ will also be a ring. This is very convenient, but may only be simulated by explicit programming of these functions in current programming languages. [6]

Code inheritance, in spite of its popularity in the object-oriented community, is not an important feature in this structuring of software, and should only be considered incidental in the construction of a software system. While interface inheritance for specifications, and other methods combining and relating specifications [50], behave very nicely when properties are added and modified, this is not the case for code inheritance. Code inheritance is a form of code reuse, where the data structure is modified in a restricted way and some operations are replaced and additional operations may be added. But this is only sound if the data invariants between the original and the new module are compatible. Also note that different models of the same abstraction may require unrelated implementations. Consider the ring abstraction. Both real numbers and scalar fields are rings, but they are implemented by very different and unrelated data structures and algorithms.

4.2. Software architecture for PDE problems

When designing the software architecture for a domain like the PDE domain, we need to investigate how various constructors, such as the array functors, can be (re)used. A nice example is in the construction of vector fields. We start with the observation that the array functor can be used to generate the value fields over a manifold $\mathcal{M}$. Simply apply $A_{\mathcal{M}}$ to the appropriate value domains, such as the reals $\mathbf{R}$, vectors or matrices. We can also use the array construction to define finite vector spaces by the expression $A_{\{1,\ldots,n\}}(\mathbf{R})$, for appropriate natural numbers n. Actually the vector space implementation algorithms are independent of the ring itself, and will work for any ring R when given the data structure $A_{\{1,\ldots,n\}}(R)$.

We have earlier noted that a scalar field has ring properties. As a consequence, n-dimensional vector fields over a manifold $\mathcal{M}$ may be constructed by either of the two approaches:

1. applying the value domain construction to vectors, $A_{\mathcal{M}}(A_{\{1,\ldots,n\}}(\mathbf{R}))$, or
2. applying the vector construction to scalar fields, $A_{\{1,\ldots,n\}}(A_{\mathcal{M}}(\mathbf{R}))$.

[6] Unfortunately, this is not fully sufficient, as the generic package mechanisms do not allow enough genericity power to let us do this once and for all. We will omit a discussion of the technicalities of these deficiencies.

There does not seem to be any reason to prefer one over the other, and conventional numerical software, as well as many object oriented numerics approaches, uses the first construction. If we study the problem domain further, we see that PDEs contain many distinct differentiation operators. Further, these operators may all be expressed from the partial derivatives on the scalar fields. A more fruitful approach is then to use the second approach as starting point. Instead of building many different constructors for the value domains (vectors, matrices, multi-linear mappings, etc.), we also note that it suffices to build a tensor constructor, which, given certain assumptions, encompass all these. Tensors also give us the building blocks needed to define coordinate free operators.

To implement the full scalar field abstraction we expand the construction $A_{\mathcal{M}}$ such that it also includes the definition of partial differential operators, to get a functor $S_{\mathcal{M}} : \mathbf{Prog} \to \mathbf{Prog}$ for the construction of scalar fields. The tensor constructor $T_{\{1,\ldots,n\}} : \mathbf{Prog} \to \mathbf{Prog}$ extends $A_{\{1,\ldots,n\}}$ with the derivation operations and other tensor operators. The tensor field construction, including all derivation operations, for a manifold $\mathcal{M}$ then becomes $T_{\{1,\ldots,n\}}(S_{\mathcal{M}}(\mathbf{R}))$. As noted, we should expect to reuse the array functors in the construction of both scalar fields and tensors.

Using the array constructor to implement both the numerical discretisation and the tensor construction allows for a reuse of the array module. But more importantly it allows a separation of concerns when implementing these modules: the array constructor may focus on the data layout pattern, while the numerical modules may focus on the numerical aspects, using the array construction for the storage aspects. The software architecture also implies that we only need to relate to, and thus implement, the discretisation method when we implement the scalar field, and that the vector and tensor field implementations are independent of this choice. If we need to change discretisation method, this will be localised to one module, and not being spread out all over the code, which is the normal case with traditional numerical software.

This also provides a route to parallelisation. We will, at the scalar field level at least, have a large collection of data values that may be distributed in a data parallel fashion [10]. Actually, it suffices to provide a parallel implementation of the array constructor to get a parallel version of the whole program. See the accompanying paper [28] for a discussion of this.

4.3. The Sophus library framework

The software architecture discussed in the previous section is the basis for the Sophus software library framework. It provides the abstract mathematical concepts from PDE theory as programming entities. Its concepts are based on the notions of manifold, scalar field and tensor field, while implementations are based on the conventional numerical algorithms and discretisations. Sophus is structured around the following concepts:

- Basic n-dimensional mesh functor $M_n : \mathbf{Prog} \to \mathbf{Prog}$, for any natural number n, taking a ring R as argument. A mesh structure $M_n(R)$ is like an array $A_{\{1,\ldots,k_1\}\times\ldots\times\{1,\ldots,k_n\}}(R)$ with element type R, and includes the general iterator operations. Specifically, operations like $+$, $-$ and $*$ are iterated over all elements (like collection oriented array operators), likewise operations to add, subtract and multiply all elements of the mesh by a scalar. There are also operations for shifting meshes in one or more dimensions. Operations like multidimensional matrix multiplication and equation solvers may easily be implemented for the meshes. Sparse meshes, i.e., meshes where most of the elements are 0 or have some other fixed value, may also be provided. Parallel and sequential implementations of mesh structures can be used interchangeably, allowing easy porting between computer architectures of any program built on top of the mesh abstraction.
- Manifolds $\mathcal{M}$. These define sets with a notion of proximity and direction. A manifold is the index set for a value field. It represents the physical space where the problem to be solved takes place.
- Scalar fields $S_{\mathcal{M}}$. They 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 and integration operations. Also, two scalar fields on the same manifold may be pointwise added, subtracted and multiplied. The different discretisation methods, such as finite difference, finite element and finite volume methods, provide different designs for the implementation of scalar fields. Scalar fields are typically implemented using the basic mesh structures for the data.
- Tensors $T_{\{1,\ldots,n\}}$. These provide coordinate free mathematics based on knowledge of the coordinate system, whether it is cartesian, axisymmetric or general curvilinear. The tensor module provides

the general differentiation and integration operations, based on the partial derivatives and integrals of the scalar fields. Tensors also provide operations such as componentwise addition, subtraction and multiplication, as well as tensor product, composition and application. The implementation uses the basic mesh structures, with scalar fields as the ring parameter.
- Equation administrators. These are abstractions containing collections of scalar and tensor fields with the purpose of building the matrices and vectors used to describe sets of linear equations, such as those needed for implicit time stepping schemes. These matrices and vectors do not represent coordinate free properties of a physical system, but abstract the important properties of linear equations. Equation administrators are also implemented using mesh structures with tensor fields or reals as the ring, as appropriate.

Further abstraction levels, such as time integrators [2], may be added to this framework to raise the abstraction level further as even more aspects of the problem domain are investigated. 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.

The Sophus library framework can be implemented using any object-oriented programming language or any programming language with abstract data types. Ideally the language should have template classes. This includes languages like Ada [5], Clu [36,37], C++ [53], Generic Java [11] and Standard ML [41]. Languages which have the abstraction mechanism but lacks the template mechanism, such as Fortran-90 [1] and Java [21], will be much harder to use. We have chosen C++ for our implementations, since it is widespread, has reasonably good compilers, and has gained some acceptance in the numerical community.

5. Specification, certification and proofs of modules

We have developed our presentation as if we could satisfy our specifications, such as that of the ring, when realising them on a computer, e.g., in the form of floating point numbers or scalar field approximations. This is not the case, which is well-known in numerical analysis. The problem is not just that we cannot exactly represent the abstractions on our finite computers, but that the available representations break the fundamental laws of our abstractions. For instance, the machine's floating point numbers do not satisfy the associative laws Eqs (1) and (3) of rings, laws which are fundamental for the development of linear algebra. This can be remedied if we choose to use computable reals as our abstraction [57]. But this would only touch the tip of the iceberg, as the discretisations we work with only provide coarse approximations to the mathematical concepts, and this seems to be fundamental for numerics [52].

Accepting this situation there is a need to supply the approximate implementations with some kind of information about their inaccuracy. This is routinely done at the level of the implementation, such as in [46]. With abstractions a certificate of the approximation's characteristics at the level of the specification is more appropriate. The certificate should also include information about storage space requirements and execution time properties. How this can be done remains open, but [55] represents a start.

Our proof methods also fall short when moving into this terrain: how do we prove that our implementation satisfies our specification sufficiently well, when it obviously has to break the most fundamental properties? What notions of satisfaction we will need is open and needs research.

6. Summary

In this paper we have sketched algebraic software methodologies and shown how they can be applied in the area of scientific computing, especially in the area of partial differential equations (PDEs). We presented the notion of a domain specific language, and universal algebra and algebraic specifications as tools to analyse a problem domain in order to find suitable domain specific languages. Then we discussed how to construct models of a specification as data abstractions. The choice of imperative or functional programming styles is independent of this analysis. Both styles may accommodate the program constructors and abstraction mechanisms needed, and both kinds of programs may be subject to the same kinds reasoning of and transformations, if necessary by enforcing simple syntactic restrictions to simplify the underlying semantic notions.

Studying software architectures was shown to be beneficial for the structuring of software libraries implementing a domain specific language. Special needs of the PDE domain, such as approximate accuracy of

an abstraction in satisfying a specification, has been discussed.

Presenting the analysis and software architecture investigation as a *software process* we see the following steps:

1. Investigate the problem domain by defining its main concepts using algebraic specifications.
2. Check the appropriateness of the concepts found by formulating the problem and sketching its solution using the problem domain specific language.
3. Check the computability of the problem domain concepts by sketching their implementation using a general purpose programming language.
4. Investigate software architectures for the problem domain concepts using algebraic and categorical methods to yield an optimal architecture for the domain, taking into account side conditions like the need for parallel implementations.

Once the infrastructure has been established, implementation of the library is evolutionary. A library component may be implemented on demand, e.g., as needed by an application, or as part of a library upgrade. Each implementation should be validated as much as reasonable and certificates provided.

We started this software process for PDEs in [30] where problem domain concepts were investigated and a possible software architecture was discussed. This led us in the direction of coordinate free numerics, and formed the basis for the Sophus PDE library framework. In [45] we presented a high level view, and proposed using concepts from pure mathematics as our domain specific language, providing them as data abstractions in a programming language, using the knowledge from applied mathematics to implement the concepts. This brings together three separate disciplines, pure and applied mathematics and computer science, in a synergetic effect for the efficient development of PDE solver software. An account of the full software process, with examples of application software based around the elastic wave Eq. (10), is presented in [29]. A very coarse picture of a scientific computing application software process which utilises a domain specific language infrastructure is:

1. Investigate the specific physical problem and develop an appropriate mathematical model, e.g., in the form of a PDE expressed in the appropriate domain specific language.
2. Analyse and reformulate the equations defining the physical model as an abstract algorithm for solving the problem.
3. Study and modify the properties of the solver to increase its accuracy and speed, for instance by improving the convergence rate.
4. Code the solver as a program in the domain specific language.
5. Carefully choose a configuration of library components to achieve optimal numerical and algorithmic properties.
6. Do domain specific optimisations to the solver program to increase speed and reduce resource usage.

These software processes for computational modelling is used in the SAGA (Scientific computing and algebraic abstractions) project. It encompasses case studies and tool development, some of which are reported here.

In the accompanying paper [23] we discuss the benefits of the coordinate free domain language compared to more traditional numerical domain languages. In [29] we showed that using this methodology it is possible to reduce program code size by up to 90%, with a corresponding gain in software development productivity.

The abstractions developed for coordinate free programming naturally are collection oriented and thus allow data parallelism in a natural way, see the accompanying paper [28].

General compiler optimisation technology is unaware of the abstractions of new domain specific languages, such as the coordinate free language. Therefore we cannot expect programs in these languages to be optimised as well as those written in a more traditional (numerical) domain oriented language. We propose a user-controlled code modification and optimisation tool, CodeBoost, see the accompanying paper [15], to take into account properties of new domain oriented languages.

Our hope is that exploring technologies like the above may aid in broadening the approaches taken to write numerical software. Further, that algebraic software methodologies and coordinate free numerics, within the broader field of object oriented numerics (OON) [3,61], may be accepted as an important approach to the development and understanding of numerical software.

Acknowledgements

Thanks to Hans Munthe-Kaas for initiating this area of research with me and providing many useful discus-

sions, and to Helmer André Friis for many good discussions and useful input. Also special thanks to the partners of the ESPRIT-IV SAGA project, Jan Heering and Phil Grant, to Victor Aarre, Dinesh, Kristin G. Frøysa, Helge Gunnarsli, John Simmons, Kristian J. Stewart, Steinar Søreide, John V. Tucker, Eric G. Wagner, Mike Webster and others involved in SAGA, Sapphire and Sophus or who otherwise have contributed to the theme of the project. In addition, Jan Heering, Krister Åhlander and the referees provided valuable feedbacks on various versions of these papers. This work has in part been financed by the European Union through the ESPRIT-IV LTR project SAGA, by The Research Council of Norway (NFR), by the Netherlands Organisation for Scientific Research (NWO), and by grants of computing resources from the Norwegian Supercomputer Committee.

References

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

[2] K. Åhlander, An extendable PDE solver with reusable components, in: *ASME 2nd International Symposium on Computational Technologies for Fluid/Thermal/Structural/ Chemical Systems with Industrial Applications*, volume 397-1 of *PVP* (*Pressure Vessels and Piping*), V.V. Kudriavtsev, C.R. Kleijn and S. Kawano, eds, American Society of Mechanical Engineers, New York, NY, 1999, pp. 39–46.

[3] E. Arge, A.M. Bruaset and H.P. Langtangen, Object-oriented numerics, in: *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito, eds, Birkhäuser, Boston, 1997, pp. 7–26.

[4] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Communications of the ACM* **21**(8) (1978), 613–641.

[5] D. Barstow, ed., *The Programming Language Ada – Reference Manual*, Springer LNCS 155, 1983.

[6] J. Bergstra, J. Heering and P. Klint, eds, *Algebraic Specification*, ACM Press Frontier Series, Addison Wesley, 1989.

[7] M. Bidoit, D. Sannella and A. Tarlecki, Architectural specifications in CASL, in: *Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, A.M. Haeberer, ed., Springer, 1999, pp. 341–357.

[8] G.E. Blelloch, *Vector models for data-parallel computing*, MIT Press, Cambridge, Mass., 1990.

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

[10] L. Bougé, The data parallel model: A semantic perspective, in: *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, G.-R. Perrin and A. Darte, eds, Springer, 1996, pp. 4–26.

[11] 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.

[12] T. Bräunl, Structured SIMD programming in Parallaxis, *Journal on Structured Programming* **10**(2) (July 1989), 121–132.

[13] M.C. Chen, Y.-I. Choo and J. Li, Crystal: Theory and pragmatics of generating efficient parallel code, in: *Parallel Functional Languages and Compilers*, B. Szymanski, ed., ACM Press, New York / Addison Wesley, Reading, Mass., 1991, pp. 255–308.

[14] V. Čyras and M. Haveraaen, Modular programming of recurrencies: a comparison of two approaches, *Informatica* **6**(4) (1995), 397–444.

[15] T. Dinesh, M. Haveraaen and J. Heering, An algebraic programming style for numerical software and its optimisation, *Scientific Programming* **8**(4) (2000), 247–259.

[16] A. Field and P. Harrison, *Functional Programming*, Addison Wesley, Wokingham UK, 1988.

[17] *USA standard FORTRAN: approved March 7, 1966*, Published by United States of America Standards Institute, 1966.

[18] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley professional computing series, Addison-Wesley, Reading, Mass, 1995.

[19] D. Garlan and D.E. Perry, eds, Special issue on software architecture, *IEEE Transactions on Software Engineering* **21**(4) (April 1995), 269–386.

[20] J.A. Goguen, A categorical manifesto, *Mathematical Structures in Computer Science* **1** (1991), 49–67.

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

[22] P. Grant, J. Sharp, M. Webster and X. Zhang, Experiences of parallelising finite element problems in a functional style, *Software – Practice and Experience* **25**(9) (Sep. 1995), 947–974.

[23] P.W. Grant, M. Haveraaen and M.F. Webster, Coordinate free programming of computational fluid dynamics problems, *Scientific Programming* **8**(4) (2000), 211–230.

[24] J.V. Guttag and J.J. Horning, Report on the Larch shared language, *Sci. Comput. Programming* **6**(2) (1986), 103–134.

[25] J.V. Guttag and J.J. Horning, *Larch: Languages and tools for formal specification*, Texts and monographs in computer science, Springer, 1993.

[26] M. Haveraaen, Distributing programs on different parallel architectures, in: *Proceedings of the 1990 International Conference on Parallel Processing*, volume II Software, D.A. Padua, ed., The Pennsylvania State University Press, University Park and London, 1990, pp. 288–289.

[27] M. Haveraaen, Efficient parallelisation of recursive problems using constructive recursion, in: *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, A. Bode, T. Ludwig and R. Wismüller, eds, Springer Verlag, 2000, pp. 758–761.

[28] M. Haveraaen, Machine and collection abstractions for user-implemented data-parallel programming, *Scientific Programming* **8**(4) (2000), 231–246.

[29] M. Haveraaen, H.A. Friis and T.A. Johansen, Formal software engineering for computational modeling, *Nordic Journal of Computing* **6**(3) (1999), 241–270.

[30] M. Haveraaen, V. Madsen and H. Munthe-Kaas, Algebraic programming technology for partial differential equations, in: *Norsk Informatikk Konferanse – NIK'92*, A. Maus and F. Eliassen et al., eds, Tapir, Norway, 1992, pp. 55–68.

[31] M. Haveraaen and S. Søreide, Solving recursive problems in linear time using constructive recursion, in: *Norsk Informatikk Konferanse – NIK'98*, S. Storøy and S. Hadjerrouit et al., eds, Tapir, Trondheim, Norway, 1998, pp. 310–321.

[32] H. Herrlich and G.E. Strecker, *Category Theory. An Introduction*, Allyn and Bacon, 1973.
[33] C.A.R. Hoare, Proofs of correctness of data representations, *Acta Informatica* **1**(4) (1972), 271–281.
[34] P. Hudak, Building domain-specific embedded languages, *Computing Surveys*, 28(4es): only available via http://www.acm.org/pubs/articles/journals/surveys/1996-28-4es/a196-hudak/a196-hudak.html, December 1996.
[35] K.E. Iverson, *A Programming Language*, John Wiley and Sons, Inc., 1962.
[36] B. Liskov, *CLU Reference Manual*, Springer LNCS 114, 1981.
[37] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, Abstraction mechanisms in CLU, *Communications of the ACM* **20**(8) (1977).
[38] B. Lisper, Data parallelism and functional programming, in: *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, G.-R. Perrin and A. Darte, eds, Springer, 1996, pp. 220–251.
[39] M.E. Mace, *Memory storage patterns in parallel processing*, volume 30 of *The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic Publishers, Boston, MA, 1987.
[40] B. Meyer, Design by contract, in: *Advances in object-oriented software engineering*, D. Mandrioli and B. Meyer, eds, Prentice Hall, Englewood Cliff N.J., 1991, pp. 1–50.
[41] R. Milner et al., *The Definition of Standard ML* (*revised*), MIT Press, Cambridge, Mass, 1997.
[42] J. Morris, Types are not sets, in: *Proceedings of the ACM Symposium on the Principles of Programming Lanugages*, October 1973, pp. 120–124.
[43] J.H. Morris, Real programming in functional languages, in: *Functional Programming and its Applications*, J. Darlington, P. Henderson and D. Turner, eds, Crest advanced courses from Cambridge University Press, Cambridge University Press, 1982, pp. 129–176.
[44] P.D. Mosses, CoFI: The common framework initiative for algebraic specification and development, in: *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, M. Bidoit and M. Dauchet, eds, Springer-Verlag, 1997, pp. 115–137.
[45] H. Munthe-Kaas and M. Haveraaen, Coordinate free numerics – closing the gap between 'pure' and 'applied' mathematics? *Zeitschrift fur Angewandte Mathematik und Mechanik* **76**(1) (1996), pp. 487–488.
[46] *The NAG library of numerical algorithms: MARK 10*, 1984.
[47] D.C. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM* **15**(12) (1972), 1053–1058.
[48] G.-R. Perrin and A. Darte, eds, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, Springer, 1996.
[49] W. Pree, *Design patterns for object-oriented software development*, ACM Press books, ACM Press, New York, 1994.
[50] D. Sannella, S. Sokolowski and A. Tarlecki, Toward formal development of programs from algebraic specifications: parameterisation revisited, *Acta Informatica* **29**(8) (1992), 689–736.
[51] B. Schutz, *Geometrical Methods of Mathematical Physics*, Cambridge University Press, 1980.
[52] S. Smale, Some remarks on the foundations of numerical analysis, *SIAM Rev.* **32**(2) (June 1990).
[53] B. Stroustrup, *The C++ Programming Language*, (3rd ed.), Addison-Wesley, 1997.
[54] B. Szymanski, ed., *Parallel Functional Languages and Compilers*, ACM Press, New York / Addison Wesley, Reading, Mass., 1991.
[55] A. van Wijngaarden, Numerical analysis as an independent science, *BIT* **6** (1966), 66–81.
[56] R. Walters, *Categories and Computer Science*, Number 28 in Cambridge computer science texts, Cambridge University Press, Cambridge, GB, 1991.
[57] K. Weihrauch, *Computability*, volume 9 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, Berlin, 1987.
[58] A.N. Whitehead, *A Treatise on Universal Algebra, with Applications I*, Cambridge University Press, 1898. Reprinted by Hafner Publishing Company, New York, 1960.
[59] N. Wirth, *Algorithms + data structures = programs*, Prentice-Hall series in automatic computation, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[60] M. Wolfe, ed., *High Performance Compilers for Parallel Computing*, Addison Wesley, Reading, Mass., 1996.
[61] M. Wong, K. Budge, J. Peery and A. Robinson, Object-oriented numerics: A paradigm for numerical object-oriented programming, *Computers in Physics* **7**(6) (Nov/Dec 1993), 655–663.