Category Theory and the Design of Parallel Numerical Algorithms

Manfred Liebmann
Max Planck Institute for Mathematics in the Sciences
manfred.liebmann@mis.mpg.de

November 30, 2006

Abstract

Concepts from category theory are used to guide the design process of numerical software. The main ideas of the proposed design techniques are presented and evaluated using the parallel algebraic multigrid algorithm as a test case.

1 Introduction

The article develops concepts and techniques for the design of parallel numerical algorithms that are inspired by category theory [1, 2]. A non-trivial example, the parallel algebraic multigrid (AMG) algorithm [6], acts as a test case for the proposed design process.
A modern approach to the design of a complex software system is the use of object oriented analysis and design (OOA/OOD) methods [4, 5] realized in the programming language C++ [3]. The critical point in this design process is the definition of the class hierarchy and the methods that implement the functionality of these classes.
Consider the special case of a numerical algorithm for solving a partial differential equation (PDE) using the finite element method (FEM). The abstract problem is formulated using the formal language of mathematics. Applying the concepts of OOA/OOD to this situation immediately leads to the requirement to model formal mathematical objects and represent them in data structures embedded in the class hierarchy. The transformation of pure mathematical concepts to data structures in a class hierarchy is highly non-trivial. For numerical algorithms to work efficiently it is necessary to have data structures that fit exactly the algorithmic requirements. Unfortunately algorithmic efficient data structures are often counter intuitive to a naive abstraction of the formal mathematical objects. This results in a tension between an easy to understand class hierarchy based on the abstraction of formal mathematical objects i.e. sets, vectors, matrices and a class hierarchy accessible only to experts that is specialized for certain key algorithms that are part of the computation.
Realizing these problems we propose a different approach to the design process of numerical software, that leaves the design process in the formal mathematical context using category theory as a bridge between the initial formal problem statement and the final implementation in the C++ programming language.
Looking at the formal mathematical problem setting and the associated numerical algorithm to solve it from the point of view of category theory gives more emphasis on the

transformations of formal objects, i.e. the flow of data through the numerical algorithm. Focusing the design of the numerical algorithm on *morphisms* that transform simple data structures, gives a direct and robust path to the implementation in C++.
The next sections on category theory and multicategories introduce some notation and are followed by implementation details of operads in C++. The last section deals with the construction of the parallel algebraic multigrid algorithm using the proposed design methodology.

2 Category Theory

In the design process of parallel numerical software it is important to create clean abstractions of the numerical components. For this purpose we explore the abstractions that category theory provides. To formulate these ideas we need some definitions.

Definition 1 (Category)**.** *Let $n \in \mathbb{N}$. A n-**category** consists of 0-**cells** (types) $a, b, c, \ldots$, 1-**cells** (morphisms) $f, g, h, \ldots$, 2-**cells** (morphisms between morphisms) $\alpha, \beta, \gamma, \ldots$ and so on, all the way up to n-**cells** together with composition operations.*

A 0-category is simply a set and a 1-category is an ordinary category.

Definition 2 (Multicategory)**.** *A **multicategory** consists of types $a, b, c, \ldots$, morphisms $f, g, h, \ldots$, a composition operation, and identities, like an ordinary category, the difference being that the domain of a morphism is not just a single type but a finite sequence of them.*

For example vector spaces and linear maps form a category; vector spaces and multilinear maps form a multicategory.

Definition 3 (Operad)**.** *An **operad** is a multicategory with only one type. Explicitly an operad consists of a set $C(k)$ for each $k \in \mathbb{N}$, whose elements are thought of as k-ary operations.*

For any vector space V there is an operad whose k-ary operations are the linear maps $V^{\otimes k} \to V$.

3 Multicategory

For numerical algorithms multicategories are the subject of interest. A multicategory $\mathcal{C}$ consists of

- a set C_0, whose elements are called types of $\mathcal{C}$
- for each $n \in \mathbb{N}$ and $X_1, \ldots, X_n, X \in C_0$ a set $C(X_1, \ldots, X_n; X)$, whose elements f are called **arrows** or **maps**

$$f : X_1, \ldots, X_n \to X \tag{1}$$

- for each $n, k_1, \ldots, k_n \in \mathbb{N}$ and $X, X_i, X_i^j \in C_0$, a function

$$C(X_1, \ldots, X_n; X) \times C(X_1^1, \ldots, X_1^{k_1}; X_1) \times \cdots \times C(X_n^1, \ldots, X_n^{k_n}; X_n) \tag{2}$$

$$\rightarrow C(X_1^1, \ldots, X_1^{k_1}, \ldots, X_n^1, \ldots, X_n^{k_n}; X) \tag{3}$$

  called **composition** and written

$$(f, f_1, \ldots, f_n) \mapsto f \circ (f_1, \ldots, f_n) \tag{4}$$

- for each $X \in C_0$, an element $1_X \in C(X; X)$, called the **identity** on X satisfying
- associativity:

$$f \circ \Big(f_1 \circ (f_1^1, \ldots, f_1^{k_1}), \ldots, f_n \circ (f_n^1, \ldots, f_n^{k_n}) \Big) \tag{5}$$

$$= (f \circ (f_1, \ldots, f_n)) \circ (f_1^1, \ldots, f_1^{k_1}, \ldots, f_n^1, \ldots, f_n^{k_n}) \tag{6}$$

  whenever f, f_i, f_i^j are arrows for which these composites make sense
- identity:

$$f \circ (1_{X_1}, \ldots, 1_{X_n}) = f = 1_X \circ f \tag{7}$$

  whenever $f : X_1, \ldots, X_n \rightarrow X$ is an arrow.

4 C++ Implementation of Operads

After the abstract definitions of category theory are in place we propose a C++ implementation of multicategories and operads. The structure of a n-category implies that morphisms should not be implemented as methods in a class definition but as *classes*, since we are also interested in morphisms of morphisms and higher order structures. For numerical algorithms the *vector* data type of the C++ Standard Template Library (STL) is a natural choice to represent a typed block of data elements, typically *double* or *int* values.

```
vector<double> x;
vector<int> c;
```

Now consider the special case of a multicategory an operad $\mathcal{C}$ defined over the type *vector*. As indicated above we want morphisms in the multicategory that are represented by classes. This can be realized by overloading the function call operator of the class. To represent the set $C(k),\ k \in \mathbb{N}$ of k-ary operations we define an abstract class acting as interface. The C++ code for an unary operation is listed below.

```
template<class T, class S> class unary
{
public:
    virtual void operator()(const vector<S> &_u, vector<S> &_v) const = 0;
};
```

A class that represents the action of a linear operator can be implemented conforming to the *unary* interface. See the class definition below.

```
template<class T, class S> class linear_operator : public unary<T, S>
{
private:
    const vector<T> &_acnt;
    const vector<T> &_acol;
    const vector<S> &_aele;
public:
    linear_operator(const vector<T> &_acnt, const vector<T> &_acol,
        const vector<S> &_aele) : _acnt(_acnt), _acol(_acol), _aele(_aele)
    {
    }
    void operator()(const vector<S> &_u, vector<S> &_v) const
    {
    ...
    }
};
```

The whole code for the internal implementation of the *linear_operator* is completely isolated. This isolation of functionality enables easy to integrate plug-and-play components. The overloading of the function call operator makes the coding syntax clean and easy to use. The code below shows the construction and use of the object *laplace*. The function call operator implements a sparse matrix multiplication using the compressed row storage (CRS) data format defined by the arrays *cnt*, *col*, and *ele* in the constructor. After the call the vector y contains the matrix-vector product with x.

```
vector<int> cnt;
vector<int> col;
vector<double> ele;
vector<double> x, y;
...
linear_operator<int, double> laplace(cnt, col, ele);
laplace(x, y);
```

The power of this approach lies in the fact, that in the example above *laplace* is an object and not a function and thus can be an argument of another morphism, implementing the notion of morphisms of morphisms.

5 Example: Algebraic Multigrid Solver

To show how the design concepts based on category theory work out in a non-trivial example we consider a preconditioned algebraic multigrid solver. The concept of a solver is implemented as an abstract class, providing an interface.

```
template<class T, class S> class solver
{
public:
    virtual void operator()(const vector<S> &_f, vector<S> &_u) = 0;
};
```

The AMG solver follows the same design strategy as outlined above. The two step process for the user of the code is to first construct the solver object and second to use the functionality according to the solver interface.

```
vector<int> cnt;
vector<int> col;
vector<double> ele;
vector<double> f, u;
...
amg_solver<int, double> amg(cnt, col, ele, 16, 1);
amg(f, u);
```

To show how the concept of objects providing functionality simplifies complex code passages the core of the multigrid code is listed below. The whole complexity of the multigrid algorithm is captured by unary operations. The different components can be easily replaced without interfering with the logic structure of the core algorithm.

```
template<class T, class S> class amg_solver : public solver<T, S>
{
private:
    ...
public:
    amg_solver(const vector<T> &_acnt0, const vector<T> &_acol0,
        const vector<S> &_aele0, const int max_level, const int max_size)
    {
    ...
            lu_decomposition<int, double> _C(_acnt[l], _acol[l], _aele[l]);
    }

    void operator()(const vector<S> &_f0, vector<S> &_u0)
    {
        memcpy(&_f[0][0], &_f0[0], _f0.size() * sizeof(S));
        multigrid(0);
        memcpy(&_u0[0], &_u[0][0], _u[0].size() * sizeof(S));
    }

    void multigrid(int _k)
    {
        int l = _k++;

        if(l < _level)
        {
            linear_operator<int, double> _K(_acnt[l], _acol[l], _aele[l]);
            gauss_seidel_f<int, double> _S(_acnt[l], _acol[l], _aele[l], _adia[l]);
            gauss_seidel_b<int, double> _T(_acnt[l], _acol[l], _aele[l], _adia[l]);
            simple_prolongation<int, double> _P(_rcnt[l], _rcol[l]);
            simple_restriction<int, double> _R(_rcnt[l], _rcol[l]);

            memset(&_u[l][0], 0, _u[l].size() * sizeof(S));
            _S(_f[l], _u[l]);

            _K(_u[l], _r[l]);
            rsb(_f[l], _r[l]);
            _R(_r[l], _f[l + 1]);

            multigrid(_k);

            _P(_u[l + 1], _s[l]);
            add(_s[l], _u[l]);

            _T(_f[l], _u[l]);
        }
        else
        {
            _C(_f[l], _u[l]);
        }
    }
};
```

The structure of the complete parallel CG-AMG solver is given in the code sequence below. First the parallel communication object *com* is set up, then the operator *laplace* and the preconditioner *amg* are constructed and passed as arguments to the constructor of the *pcg_solver*. The preconditioner is passed as an object implementing the *solver* interface and this is all the conjugate gradient (CG) algorithm knows about the preconditioner thus implementing true plug-and-play capabilities.

```
vector<int> con;
vector<int> cnt;
vector<int> col;
vector<double> ele;
vector<double> f, u;
...
mpi_communicator<int, double> com(con);
linear_operator<int, double> laplace(cnt, col, ele);
amg_solver<int, double> amg(cnt, col, ele, 16, 1, com);
pcg_solver<int, double> pcg(laplace, amg, 1.0e-9, 256, com);
pcg(f, u);
```

6 Conclusions

The use of concepts from category theory in the design process of numerical algorithms gives another point of view of a complex software system, that is more focused on the data flow of the algorithm, than more traditional approaches to OOA/OOD. This alternative way of capturing the complexity of a numerical algorithm using the concepts of multicategories and operads leads to the implementation of plug-and-play components, that are robust and easy to use.

7 Future Work

The construction of numerical software with concepts from category theory leads also to the possibility to construct *visual representations* of the algorithm. Every morphism can be thought of as a little piece of a circuit, that connects input lines with output lines. This visual representation could then be even used for the construction of algorithms, linking together simple algorithmic building blocks to more complex structures.
The point of view of category theory also provides an approach to software optimization that is not realized in todays compilers. Due to the fact that the numerical algorithm is captured by a category, this potentially allows the *structural* manipulation of the algorithm by a compiler. It is possible to establish *algebraic* relationships between the algorithmic building blocks the morphisms and thus allow high level simplification algorithms comparable with the *Simplify* command in the *Mathematica* [7] software package.

References

[1] T. LEINSTER: Higher Operads, Higher Categories, Cambridge 2004

[2] S. AWODEY: Category Theory, Oxford 2006

[3] B. STROUSTRUP: The C++ Programming Language, Addison-Wesley 1997

[4] G. BOOCH: Object-Oriented Analysis and Design with Applications, Addison-Wesley 2004

[5] E. GAMMA: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995

[6] C.C. DOUGLAS, G. HAASE, U. LANGER: A Tutorial on Elliptic Pde Solvers and Their Parallelization, Society for Industrial and Applied Mathematics 2003

[7] Internet: http://www.wolfram.com