STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 4, 2011

ON USING GENERICS FOR IMPLEMENTING ALGEBRAIC STRUCTURES

VIRGINIA NICULESCU

Abstract. Object oriented programming and design patterns form a very good framework for implementing a computational algebra system. Object oriented programming offers different kinds of mechanisms for obtaining high level of genericity; and these are also mechanisms for reusing.

The need to represent and work with abstract algebraic structures implies the need for genericity. This could be achieved by using parametric polymorphism or type polymorphism. We analyze in this article the suitability of using parametric polymorphism, in an object-oriented programming context, for implementation of algebraic structures, showing some advantages but also some problems. The implementation issues are discussed for the particular case of the square matrices of a certain order.

The analysis emphasizes also general advantages of using parametric types – such as type safety, but also some constrains forced by them.

1. Introduction

In modern programming, genericity and reusing are very important and much discussed issues, and so there are different mechanisms to achieve them. In object oriented programming we may distinct two main types of such mechanisms: one based on inheritance of type polymorphism (or polymorphic types), specific to pure object oriented languages, and another based on parametric types [1].

Computer algebra provides a compelling application of parametric polymorphism, where various algebraic constructions, such as polynomials, series, matrices, vector spaces, etc, are used over various different coefficient structures, which are typically rings or fields [2]. The work initiated by Jenks and

Received by the editors: March 25, 2011.

2010 *Mathematics Subject Classification.* 08A99.

1998 *CR Categories and Descriptors.* D.1.5 [**Software**]: Programming Techniques – *Object Oriented Programming*; I.1.4 [**Computing Methodologies**]: Symbolic and Algebric Manipulation – *Applications*.

Key words and phrases. genericity, OOP, templates, patterns, algebraic structures, abstractness.

Trager [11] led to the formulation of domain and category constructors in Axiom [24, 10], and higher-order domain and category producing functions in specialized language Aldor [3, 4].

Parametric genericity, initially represented in object oriented setting by source code reuse mechanism as C++ templates, became more and more popular and other object oriented languages as Java and C# enhanced their new versions with mechanisms that offer parametric types.

In C++ the template mechanism allows us not to create a single class, but to specify only once the pattern for creation of some classes that are different only by the type of some parameters. The template mechanism allows a very high degree of flexibility, but it is considered in some literature not a really parametric polymorphism mechanism since for each actual parameter a new class is created

Both the NTL library for number theory [22] and the Linbox library for symbolic linear algebra [6] use C++ templates to achieve genericity.

The mechanism which was included in Java since JDK 1.5 is considered more efficient since just one class is created for each parameterized class. Also, the mechanism of parameterized Java classes allows bounded polymorphism – the specification of a certain behavior of parameters by interface implementation.
A similar mechanism is implemented in C# too.

A comparison between C++ templates and the extensions for generics of the C# and Java languages based on their suitability for scientific computing was done in [8]. These measurements suggest that both Java(TM) and C# generics introduce only little run time overhead when compared with non-parameterized implementations. With respect to scientific application, C# generics have the advantage of allowing value types (including builtin types) as parameters of generic classes and methods. Also, in [5] there is study about the performance of generics for scientific computing in various programming languages, based on a standard numeric benchmarks. The conclusion was that in current implementations of generics must be improved before they are used for efficiency-critical scientific applications.

Design patterns [7] are also very important when we need genericity and reusing. They already showed there power, and for the case of a computer algebra system they bring important advantages. Creational design pattern are particularly important in this context, and the analysis in this paper is focused on the suitability of using them in the specified languages, too.

An efficient design for a computer algebra system with certain advantages has been presented in our previous papers [17, 18, 19]. This approach allows working not only with concrete algebraic structures, but also with abstract algebraic structures. The advantages of this approach result from the usage of:

- *creational design patterns* – which allow us to build not only a flexible numerical algebraic system, but also a general abstract algebraic system (these bring the most important advantages of implementing abstract algebraic structures);
- *reflection and dynamic loading* – which allow *automatic conversions* between compatible structures, and also allow dynamic creation of new classes that correspond to different algebras specified by the user;
- *representation independence* – which allows us to operate with algebraic elements independently of how their component values are represented.

Implementation of this previous approach was based on type polymorphism. Polymorphic types based on inheritance offer several advantages (a greater flexibility, efficiency, conversions), but they have also important disadvantages: they cannot assure the homogeneity, and the possible errors are identified and eventually treated only at the run time. Because of this it has been advocated that the parametric polymorphism is more appropriate for the implementation of the algebraic structures.

The Java Algebra System (JAS)[14, 13, 12] provides a software library using generic types for algebraic computations implemented in the Java programming language. Our idea of using the creational design patterns has been used, but in the context of the Java generic types. The project emphasized nice advantages of using Java (object-orientation, type safety, multi-threaded), but also some inconveniences of using Java generics for a CAS related to type erasure, dependent types, and performance.

2. General requirements for the algebraic structures implementation

The main requirement is the possibility of working with abstract algebraic structures like groups, rings, fields, etc. The user has to be allowed to define concrete algebraic structures by using these abstractions.

This implies that we have to define abstract classes for elements of each abstract algebraic structure.

New abstract algebraic structures may be built over other algebraic structures; for example polynomials and matrices. Polynomials are built over a unitary commutative ring, and they also form a unitary commutative ring.

We may have polynomials or matrices that have different coefficient types: real, complex, etc. Also, a polynomial or a matrix can be built over any kind of algebraic unitary commutative ring $(R, +, *)$, and they also form rings.

The *Composite* design pattern may be used to implement these kinds of structures. Using the *Composite* pattern we may define polynomials over other polynomials. Similar examples may be given for matrices – we can define matrices over polynomials, or matrices over matrices. So, we may achieve the flexibility imposed by our goal.

We will analyze in what it follows the possible solutions for general algebraic structures implementation by considering two approaches: one based on the classic C++ templates, and one based on generic classes as they are implemented in Java.

First obvious conclusion is that the conversions will not be allowed any more, but very often the mathematicians' opinion was that it is enough if we are able to operate only inside one algebraic structure.

2.1. **Basic design.**

The basic algebraic structures are algebras with one or two binary operations such as groups, rings, and fields.

Following the definition of the basic algebraic structures we have interfaces that correspond to semigroups, monoids, groups, rings, unitary rings, division rings, and fields. These interfaces characterize the elements of the algebraic structures and their behavior. So, we have interfaces such as `GroupElem`, `RingElem`, etc.

The operations defined by these interfaces such as `plus()`, `minus()`, `opposite()`, create new elements.

For concrete groups, rings, etc. concrete classes are built by implementing the corresponding interfaces.

An algebraic structure is defined by the domain of its elements and by a set of operations. The operations could have different properties (i.e. commutativity, neutral element, etc). These properties are characteristics of the entire algebraic structure; they do not characterize each element of the domain. So, these properties are specified into separate classes for which we normally have to apply the *Singleton* pattern [7]; only one single instance would be not only enough, but also necessary in order to assure a safe representation. These classes can also be viewed as *Abstract Factories* [7], since they are able to create *products* which are the neutral elements for the algebraic operations.

A factory class associated to a ring structure will define operations such as:

```
-: isCommutative()
-: isUnitary()
-: createZero();
-: createOne()
```

An important issue is represented by the way in which is implemented the connection between the instances of the class that represents the algebraic elements, and the class that characterized the entire algebraic structure and which is also the factory of the neutral elements.

If the *Abstract Factory* and *Singleton* design patterns are used, then the connection between the elements and the singleton instance of their algebraic structure is done through the single global access point assured by the *Singleton* pattern. An elegant solution to implement this is to define the factories as static nested classes within the classes that define the algebraic elements.

If static definitions are not available, then a simple association relation between each element instance and a factory instance have to be implemented. In this case, problems may arise since we may have more factory instances for the same algebraic structure. We have to verify the correct association between each element and the corresponding factory, too.

Since we will focus more on implementation decisions we will restrain the analysis just to the case of square matrices implementation over unitary rings. So, we will consider the interface `RingElem` and the associated factory. The class `MatrixElem` implements also the interface `RingElem` since the set of all matrices over a ring forms also a ring. So, we have to be able to create matrices of matrices, etc.

3. C++ templates

In C++ we have the possibility to overload the operators and so we will use the classic operators for defining addition, difference, multiplication, etc. The interface `RingElem` can be specified based on operators.

```
template<typename T>
class RingElem{
public:
...
    virtual T operator+( const T&) = 0;
    virtual T operator-(const T&) = 0;
    virtual T operator*(const T&) = 0;
    virtual T& operator-() = 0;
/////////////////////////////////
    class Factory{
        public:
```

```
        virtual bool isCommutative()=0;
        virtual bool isUnitary()=0;
        virtual T createZero()=0;
        virtual T createOne()=0;
    };
////////////////////////////////
    virtual Factory* getFactory()=0;
};
```

The instance method `getFactory()` has to be defined in order to make the factory object accessible through each created element of the structure. This is particular important when the global access point to the factory object could not be implemented.

The parameter type T can be replaced by any type which may be considered as a unitary ring. An example is the class `Complex` for which we give a partial implementation.

```
class Complex:  public RingElem<Complex>{
    double re, im;
public:
    ...
    //the implementation of the operators
    ...
    class Factory:  public RingElem<Complex>::Factory{
        Factory(){}
        static Factory* instance;
    public:
        Complex createZero(){
            return Complex(0,0);
        }
        ...
        static Factory* getInstance(){
            if (instance==0) instance = new Factory();
            return instance;
        }
    };
    RingElem<Complex>::Factory* getFactory(){
        return Factory::getInstance();
    }
    static RingElem<Complex>::Factory* getFactoryStatic(){
        return Factory::getInstance();
    }
};
```

We may emphasize the fact that we were able to use the *Singleton* design pattern, and also to implement a static method `getFactoryStatic()` that returns the unique factory instance.

The class `MatrixElem` is built over a ring, so the parameter `T` has to implement the interface `RingElem`. But this constrain cannot be specified for a C++ template. This is a clear disadvantage.

```
template <typename T>
class MatrixElem:  public RingElem<MatrixElem>{
    Storage<T> mat;
    int n;
    RingElem<T>::Factory* factoryE;
public:
    ...
    MatrixElem (int n, RingElem<T>::Factory* f){
    //matrix Zero
        factoryE=f;
        ...
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                mat.add(i,j, factoryE->createZero());
        this->n=n;
    }

    class Factory:  public RingElem<MatrixElem>::Factory{
        int n;
        RingElem<T>::Factory* f;
        Factory(int n, RingElem<T>::Factory* f){
            this->n = n;
            this->f = f;
        }
        static vector<Factory*> instance;
    public:
        ...
        MatrixElem createZero(){return MatrixElem(n, f);}

        static Factory* getInstance(int n, RingElem<T>::Factory* f){
            for(int i=0;i<instance.size();i++)
                if ((instance)[i]->n==n)return instance[i];
            Factory * fnew=new Factory(n,f);
            instance.push_back(fnew);
            return fnew;
        }
    };
    RingElem<MatrixElem>::Factory* getFactory(){
        return Factory::getInstance(n,factoryE);
    }
};
```

A general storage is used for the elements because we want to preserve the advantage of representation independence. The class `MatrixElem` has to work with a factory (`factoryE` for managing the type of its elementary elements of type `T`). This factory is used, for example, when the matrix *zero* and the matrix *one* are created.

Also, the `MatrixElem` is itself a `RingElem`, so it has to be related to a factory class corresponding to this new structure of ring type.

A matrix algebraic structure is characterized by the order of the matrix, too. In the proposed implementation we have only one class for all the matrix structures with elements of the same type. So, the associated factory has to create the *zero* and the *one* matrices with the order given as parameter. The factory class associated to the template class `MatrixElem` uses a generalization of the *Singleton* design pattern, and stores a vector of instances, one for each order of the matrices in use.

Each matrix has to be associated to a ring factory. Because we could use *Singleton* design pattern for the factory implementation, the creation of the factory objects is very well controlled; we will have only one factory for each algebraic structure (in our case one for each order and element type). But since we have a different factory for a different order we cannot implement a static method `getFactoryStatic()` – as we did for complex numbers.

All the operations are correctly defined if they operate only on elements from the same algebraic structure – matrices of the same order and with elements of the same type. The template parameter imposes the same type for the elements type but the order is not constrained. Exceptions have to be thrown when the operations are called with wrong parameters.

In order to solve this problem related to dependent types [21], a better solution defines the matrix order as a template parameter, too. This leads to the creation of one class for any particular algebraic structure. From the programming point of view this is not very efficient (a bunch of classes will be created), but assures a high degree of rigor from the mathematical point of view, and also a static verification of the operands.

```
template <typename T, int order>
class MatrixOrdElem:  public RingElem<MatrixOrdElem>{
    Storage<T> mat;
    static RingElem<T>::Factory* factoryE;
public:
    MatrixOrdElem (){// matrix Zero
        factoryE=T::getFactoryStatic();
        for(int i=0; i<order;i++)
            for (int j=0; j<order;i++)
                mat.add(i,j,factoryE->createZero());
    }

    class Factory:  public RingElem<MatrixOrdElem>::Factory{
        Factory(){ }
        static Factory* instance;
    public:
        MatrixOrdElem createZero(){return MatrixOrdElem();}
        ...
        static Factory* getInstance(){
            if (instance==0) instance = new Factory();
            return instance;
        }
    };
    static RingElem<MatrixOrdElem>::Factory* getFactoryStatic(){
        return Factory::getInstance();
    }
};
```

With this variant we eliminate the necessity to give the associated factory instance (for the type T) as a constructor argument, and so, the necessity to create/access a factory instance each time a new matrix instance is created. The presented solution uses a static method `getFactoryStatic` that returns the singleton instance. This method is called inside the constructors of the class `MatrixOrdElem`.

For creating factories, this solution is safe because the responsibility of correct initialization of the factory is not let to the user. But this brings some other disadvantages: C++ predefined types (int, double, etc.) could not be used anymore as values of the parameter `T`. The possibility of using predefined type as a template parameter is one of the nicest characteristic of genericity based on templates. One solution to preserve this possibility (different from that with factory parameter of the constructor) is to set the static member `factoryE` only from outside (the user has to do this setting properly before instantiation – which is an important disadvantage). Also, corresponding factories for the predefined types have to be defined.

```
class IntFactory: public RingElem<int>::Factory{
    IntFactory(){}
    static IntFactory* instance;
public:
    int createZero(){return 0;}
    static IntFactory* getInstance(){
                if (instance==0) instance = new IntFactory();
        return instance;
    }
    ...
};
```

4. Java Generics Approach

Using Java generics is similar in many aspects to C++ templates, but there are important differences imposed by the different mechanism for achieving parametric polymorphism.

In this case the implementation of the `RingElem` interface uses recursive bounded polymorphism –which is a clear advantage of this solution. Factory interfaces could also be defined as nested interface.

```
public interface RingElem <T extends RingElem<T>> {
    public interface Factory <T >{
        T createZero();
        ...
    }
    public T plus(T e );
    public boolean isZero();
    ...
    public Factory<T> getFactory();
}
```

For a simple implementation of a concrete type (such as `Int` for integer numbers, or `Complex` for complex numbers) the implementation of the factory could be based on *Singleton* pattern and on static members, in a similar way as for C++ templates. The predefined types are used based on autoboxing properties.

The class `MatrixElem` is defined as a generic class with a type parameter bounded to `RingElem<T>`. All the matrices of a certain order and with a certain type of its elements type also form a ring, and so the class `MatrixElem` implements the interface `RingElem<MatrixElem<T>>`.

```
public class MatrixElem <T extends RingElem<T>> implements RingElem<MatrixElem<T>>{
    private Storage<T> a;
    private int n;
    private RingElem.Factory<T> factory;

    public MatrixElem(int n,RingElem.Factory<T> f){
    //create a zero matrix
        this.n=n;
        factory =f;
        a=...
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++)
                a.add(i,j, factory.createZero());
    }
    ...
    public MatrixElem<T> plus(MatrixElem<T> x) throws Exception{
        if (n!=x.getOrder())
            throw new Exception("illegal arguments");
        ...
    }

    public Ring.Factory<T> getFactory(){
        return new MatrixFactory<T>(n,factory);
    }
}
```

If we intend to follow the design based on which factories classes are defined inside elements classes, then for this kind of objects the factory is defined as an inner class. Since we work with generics we cannot define static members and so the nested class cannot be static. If we let the factory class to be an inner class, then each factory instance of type `MatrixElem<T>.Factory` is connected as a member with an instance of the outer class `MatrixElem`. This could lead to the creation of one factory instance for each matrix element – which is abnormal. Because of this, the general design decision to define the associated factory class as nested class has to be eliminated, and we have to define the factory class outside the class `MatrixElem`. In any cases, the *Singleton* pattern cannot be used.

```
class MatrixFactory<T extends RingElem<T>> implements RingElem.Factory<MatrixElem<T>>{
    private RingElem.Factory<T> factory;
    private int n;
    public MatrixFactory(int n,RingElem.Factory<T> f){
        this.n=n;
        factory = f;
    }
    public MatrixElem<T> createZero(){
        return new MatrixElem<T>(n,f);
    }
    ...
}
```

The connection between elements and factories is done through the constructor arguments. (This solution has been also chosen in JAS.) In order to obtain a factory for elements which are matrices of a certain order and a certain elements type, we may use directly the factory constructor or the instance method of the `MatrixElem` class `getFactory()`. This solution is not a very safe solution since the creation of the factory instances is not controlled. Also, the responsibility for correct association between the elements and factories is let to the user and could become very complicated when matrices of matrices are created.

Another solution to assure an implicitly correct done association is to use reflection, but there are several drawbacks too. Implicit constructors (with no arguments) have to be defined for all elements, and also the problem related to dependent types (different structures for different orders of matrices) cannot be solved efficiently. The following code shows a possible definition based on reflection of a constructor for `MatrixElem`.

```
public MatrixElem( Class<T> c, int n){
    try{
        this.n=n;
        java.lang.reflect.Method meth=null;
        Object retobj=null;
        try{
            meth = c.getMethod("getFactoryStatic");
            retobj = meth.invoke(null);
        }
        catch(java.lang.NoSuchMethodException e){
            try{
                meth = c.getMethod("getFactory");
                retobj = meth.invoke(c.newInstance());
            }
            catch(java.lang.NoSuchMethodException e1){
                ...
            }
        }

    factory = (Ring.Factory<T>)retobj;

    ...
}
```

First a possible invocation of the method `getFactoryStatic()` is done; if this fails then the instance method `getFactory()` is invoked. This means that for the classes for which we may define the static method `getFactoryStatic()`, this method will be used, and for the others the instance method `getFactory()` will be used.

An example of using such a constructor is:

```
Matrix<Int> a = new MatrixElem<Int>(Int.class ,2);
```

It can be noticed that no factory instance creation is necessary.

5. Conclusions

The problem of implementing a general matrix over any kind of commutative and unitary ring emphasizes the fact that we need more than specifying a parameter with a certain behavior; we need to refer and work with some specific instances from the domain of the type parameter before knowing their exact concrete type.

The main emphasized problem related to the C++ templates was the fact that we cannot specify explicitly any restriction related to the parameters, to bound them. A step forward in this direction has been done for generic Java classes, where for a parameter we can specify that it implements a certain interface. But we may ask if restraining the behavior is powerful enough. The C++ templates mechanism is considered for implementing parametric polymorphism based on an "heterogeneous" approach. The "heterogeneous" approach constructs a special class for each different use of the type parameters. The compiled code is fast, but the object code could become bulky since we have many different versions of each class.

Still, the analysis emphasized the fact that we have certain advantages using the old C++ templates over the new Java or C# generics, for this case. Java generics implement "homogenous" approach of the parametric polymorphism. Since is based on "type erasing" we have strong restrictions, and the most important one, for the analyzed case, was the impossibility of specifying static members for the generics, and so the impossibility of specifying properties related to an entire algebraic structure.

For creation and usage of the neutral elements at the abstraction level imposed by the generic type definition, there are solutions based on the creational design patterns: *Factory Method*, *Prototype*, or *Abstract Factory. Abstract Factory* is the best solution since all these special values can be considered products from the same family. *Abstract Factory* is usually used together with *Singleton* pattern that assures a global access point the factory instance. Still, we have to define a concrete factory for each concrete type in order to be used as an actual parameter.

Some problems could be observed also from the Java implementation of JAS library. For example, for the simple coefficients (elements) rings such as integer or rational numbers, there are classes that implement the both interfaces: one defined for the elements of the structure, and the second defined for the associated factory. These kinds of solutions may locally simplify things but they are not based on appropriate object-oriented design decisions. Factories have to be related to the type (class) and not to each element of an algebraic structure.

References

[1] Cardelli, L., Wegner, P. *On understanding types, data abstraction, and polymorphism* ACM COMPUTING SURVEYS, (1985).

[2] Chicha, Y.; Lloyd, M. Oancea C.; Watt, S.M. *Parametric Polymorphism for Computer Algebra Software Components.*

[3] Chicha,Y.; Defaix, F. ; Watt, S. TR537 - *The Aldor/C++ Interface: User's Guide.* Technical report, Computer Science Department - The University of Western Ontario, 1999.

[4] Chicha,Y.; Defaix, F. ; Watt, S. TR538 - The Aldor/C++ Interface: Technical Reference. Technical report, Computer Science Department - The University of Western Ontario, 1999.

[5] Dragan, L.; Watt, S.M. *Performance Analysis of Generics in Scientific Computing*, Proceedings of Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05), 2005, pp.93-100.

[6] J. G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LinBox: *A Generic Library for Exact Linear Algebra.* In Proc. of ICMS, pages 40-50. A. Miola ed. Academic Press, 2002.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

[8] Gerlach, J.; Kneis, J. *Generic programming for scientific computing in C++, Java, and C#.* Lecture Notes in Computer Science. Proceedings of International Workshop on Advanced Parallel Processing Technologies (APPT) ¡5, 2003, Xiamen¿, pp.301-310

[9] Gilbert, J., *Elements of Modern Algebra*, PWS-Kent, Boston, 1992.

[10] Jenks, R. D.; Sutor, R. S. *AXIOM, The Scientific Computation System.* Springer-Verlag, 1992.

[11] R. D. Jenks and B. M. Trager. *A Language for Computational Algebra.* In Proc. SYMSAC, pages 6-13. ACM, 1981.

[12] Kredel, H.; Jolly, R. *Generic, Type-Safe and Object Oriented Computer Algebra Software* Computer Algebra in Scientific Computing, Lecture Notes in Computer Science, 2010, Volume 6244/2010, 162-177,

[13] Kredel, H., *Evaluation of a Java Computer Algebra System*, in Lecture Notes in Computer Science, volume 5081/2008, pp. 121-138, Springer Berlin / Heidelberg.

[14] Kredel, H.: *On the Design of a Java Computer Algebra System.* In Gitzel, R., Aleksy, M., Schader, M., Krintz, C., eds.: Proc. PPPJ 2006. ACM International Conference Proceedings Series, Mannheim University Press (2006) pp. 143-152.

[15] Lujn, M., Freeman, T. L., Gurd, J. R., *OoLaLa: an Object Oriented Analysis and Design of Numerical Linear Algebra*, In the Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA 2000.

[16] Musser, D.R., Scine A., *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley, 1995.

[17] Niculescu, V.,*A Design Proposal for an Object Oriented Algebraic Library*, Studia Univ. Babes-Bolyai, Informatica, Volume XLVIII, No. 1, 2003, pg. 89-100.

[18] Niculescu, V., Moldovan, G.S. *OOLACA: an object oriented library for abstract and computational algebra*, Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications OOPSLA 2004, Vancouver, BC, CANADA, ACM Press New York, NY, USA, pp. 160-162.

[19] Niculescu, V., Moldovan, G.S. *Building an Object Oriented Computational Algebra System Based on Design Patterns.* Proceedings of International Symposium on Symbolic and Numeric Algorithms for Scientific Computing SYNASC'05, Timisoara, IEEE Computer Society Press, Romania, Sept. 2005, pp. 101-108.

[20] Niculescu, V.,*Teaching about Creational Design Patterns – General Implementations of an Algebraic Structure*, In the Proceedings of ECOOP 2003 Workshop on Pedagogies and Tools for Learning OOP, Darmstadt, Germany.

[21] Poll, E.; Thomson, S.: *The type system of Aldor.* Technical report, Computing Science Institute Nijmegen (1999)

[22] V. Shoup. *NTL: A Library for doing Number Theory*, 2004, http://www.shoup.net/ntl/doc/tour.html.

[23] B. Stroustrup, M. Ellis, *The Annotated Reference C++ Manual*, Addison-Wesley, 1994.

[24] Watt,S. M.; Jenks,R. D. ; Sutor,R. S.; Trager, B. M.. *The Scratchpad II Type System: Domains and Subdomains*. In Proc. of Computing Tools For Scientific Problem Solving, pages 63-82. A. Miola ed. Academic Press, 1990.

[25] *Generic Java*,http://download.oracle.com/javase/1.5.0/docs/guide/language/generics.html

Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca
E-mail address: `vniculescu@cs.ubbcluj.ro`