STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume L, Number 2, 2005

INTEGRATING CONVERSIONS INTO A COMPUTATIONAL ALGEBRAIC SYSTEM

VIRGINIA NICULESCU AND GRIGORETA SOFIA MOLDOVAN

Abstract. Conversions play an important role in any computational algebraic system. This article analyzes two approaches for integrating conversions. The first is based on template method design pattern and the other is based on aspect-oriented programming. The advantages and disadvantages of these approaches are emphasized.

1. Introduction

Object oriented programming and design patterns introduce a high level of abstraction that allows us to implement and work with mathematical abstractions. Classic algebraic libraries and systems, based on imperative programming, contain subalgorithms for working with polynomials, matrices, vectors, etc. Their main inconvenience is the dependency on types.

In [4] we have analyzed the design of the kernel for an object oriented computational algebra system based on design patterns. This approach allows us to work not only with concrete algebraic structures, but also with abstract algebraic structures. The advantages are mainly given by the creational design patterns, by reflection and dynamic loading, and by representation independence. These introduce significant flexibility and abstraction.

Conversions play an important role in a computational algebraic system, and we present here two solutions for integrating them.

2. The Basic Design of the Algebraic System

The main requirement for an alegraic system 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. We restrict the discussion to basic algebraic structures, and to polynomials and vector spaces.

Received by the editors: November 21, 2005.

2000 *Mathematics Subject Classification.* 68R01, 68U99.

1998 *CR Categories and Descriptors.* J.2 [**Computer Applications**]: Physical Sciences and engineering – *Mathematics and statistics* ;

Abstract classes are defined for elements of each abstract algebraic structure. Their hierarchy is shown in Figure 1.

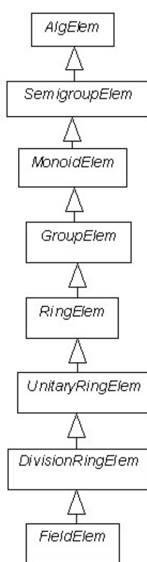

Figure 1. The class diagram for basic abstract algebraic structures.

New abstract algebraic structures may be built over the existing algebraic structures; for example polynomials and vector spaces. Polynomials are built over a unitary commutative ring, and they also form a unitary commutative ring. In order to define a vector space we need a group, a field, and an external operation.

The *Composite* design pattern [3] may be used to implement this kind 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, etc.

More details about the system design can be found in [4].

3. Conversions

Operations between different types of numbers are an important issue for an algebraic system, and the design of the algebraic structures must take into consideration the design of the conversions.

If we add a real number to a complex number, we know that the result is a complex, because a real number is also a complex number, which has the imaginary part equal to zero.

If we generalize this, we arrive to a situation where between two algebraic structures an inclusion relation may be defined. We may have subgroups, subrings, etc. Let us consider that we have a group $(G, +)$, and a subgroup (or submonoid) $(SG, +)$, $SG \subset G$. Corresponding to these, we will have the classes `GElem` and `SGElem`. If we have an element g of type G and an element sg of type SG, we may consider that sg is also an element of type G, and we may use it in operations of class `GElem`. For this we have to allow automatic conversions from SG type to G type.

For example, if we consider the group $(\mathbb{Z}, +)$ and the monoid $(\mathbb{N}, +)$, we have to allow conversions from natural to integer numbers.

One solution to allow this is to use inheritance for defining `SGElem` class (to be derived from `GElem`). Then the operation `g.plus(sg)` would be possible. But the operation `plus` is a commutative one, so we would like to also allow the operation `sg.plus(g)`, but using this solution this is not possible.

But the main disadvantage of the solution based on inheritance can be understood from the following example. We define the class `IntElem` derived from `GroupElem`, corresponding to the group $(\mathbb{Z}, +)$, and the class `NaturalElem` derived from `MonoidElem`, corresponding to the monoid $(\mathbb{N}, +)$. If we choose the solution based on inheritance for conversions, we have to derive `NaturalElem` from class `IntElem`, as well. So, we arrive to a situation when `NaturalElem` is a group, too — which is completely wrong (the corresponding UML diagram is presented in Figure 2).

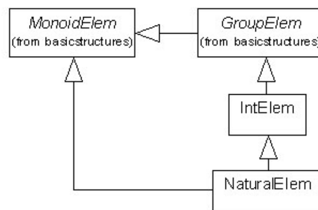

FIGURE 2. A wrong solution for implementing conversions.

A compatibility between two classes is defined when an instance of one class can be converted into an instance of the other.

As a general rule we allow defining conversion if an inclusion type relation can be defined between structures.

Still, we should allow some special cases:

- To add a simple real to a polynomial over reals.
- Consider the group of continuous functions $(\mathbb{R}^A, +)$, where $\mathbb{R}^A = \{f | f : A \to \mathbb{R}, f \text{ continuous}, A \subset \mathbb{R}\}$. If we have $g \in R^A$, we can add it to an $r \in \mathbb{R}$, and the result will be of type $\mathbb{R}^A$. We can add g to r, because r can be seen as $h(x) = r$, for any $x \in A$, hence we actually apply the add operation to g and h.
- New structures may be defined based on the existing ones. For example, the group $(M_n, +)$, where $M_n = \{a + b\sqrt{n} | a, b \in \mathbb{Z}\}$ and $n \in \mathbb{N}$, n is a prime number. The structure $(M_n, +)$ is a group, and $\mathbb{Z} \subset M_n$ (because if $b = 0$, $a + b\sqrt{n} \in \mathbb{Z}$, $\forall a \in \mathbb{Z}$).

So, we may also admit conversions when we have a structure defined over another structure, and when a simple element of the basic structure may form an element of the complex structure.

Another special case when conversions have to be used is related to special representations and precision. If we want to represent structures that are defined over infinite sets, we cannot represent all the elements. So, we may consider only the elements of some subsets. These subsets may be included one into another. Corresponding to these we will have different classes. Integers may be represented using primitive types like `int` or `long`, but also using another representation that could be based on a bigger base. Because they are different representations of the same algebraic structures, they have to be compatible. (The situation is similar to that of the usual conversions that appear in any programming language.) When it is the case, we may base our conversions on the conversions in the implementation language.

3.1. **The Basic Design.** The solution that we suggest is based on *reflection* and *dynamic loading*. These are used for defining new compatibilities between the existing and the new created structures, and for dynamic loading of these compatibilities. The compatibilities are implemented as distinct classes, and their names are stored in a specific file, which can be updated. We will be able to choose whether conversions are accepted or not, or to choose a subset of the set of all defined conversions.

We define a `Conversions` class, which will store all the conversions available in the system. This class will be a singleton [3], because we do not need more than one instance of it. We also define a `Conversion` interface, which will handle the actual conversion, and which has three methods (Figure 3). The methods `getFirstClass()` and `getSecondClass()` are used to determine what type of conversions the concrete class deals with. The `convert(...)` method converts one of the two parameters to the class of the other parameter. Because we do not need to know which parameter has been converted, the result will be an array with two elements.

When the constructor of the `Conversions` class is called, it will dynamically load from the file all the classes that implement the `Conversion` interface, and store

an instance of each class in a list. The method `existConversion(...)` verifies whether there exists a conversion between the two classes given as parameters. If there is one, the `convert(...)` method of the `Conversions` class will be used to make the actual conversion, using the corresponding `Conversion` class.

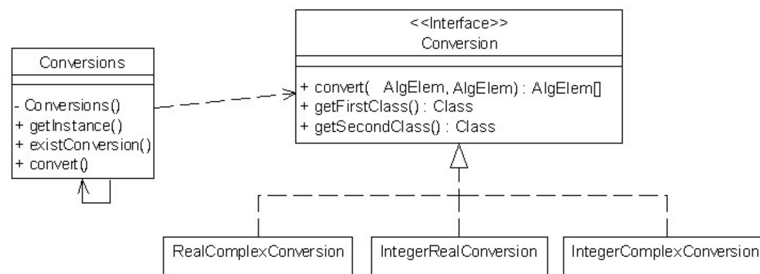

FIGURE 3. Integer - Real - Complex conversions.

The inconvenience of this solution is that a different conversion class must be defined for each possible compatibility. We must define a class for converting an integer to a real, one for converting a real to a complex, and also one for converting an integer to a complex. This inconvenience can be solved using a *graph of types* and then finding the smallest path from one type to another. The vertices of this graph represent the types, and an edge between two vertices represents a compatibility between the two corresponding types.

In the following, we present two approaches to integrate this design into the algebraic system.

3.2. **The *Template Method* Approach.** For each operation `equals()`, `plus()`, `minus()`, etc. the same steps are followed every time, so one approach to integrate the conversions could be based on *Template Method* design pattern [3].

As we have said before, two algebraic elements are considered compatible if they have the same class or there is a conversion between them (Section 3). Any algebraic element can be compared for equality with another algebraic element if they are compatible. So, `AlgElem` has an `equals()` method that compares two `AlgElem`s for equality.

It is desirable to build an extendible system, to which the user can add new types. For each new type, the developer will have to override the `equals()` method, where he/she has to verify first if those elements are compatible. The `equals()` method is implemented as a template method. It verifies whether the elements have the same type, and if they do, it calls the method `equalsS()` which compares two elements of the same type. If the elements have different types, but they are compatible, it first calls the `convert()` method and then the

`equalsS()` method to compare them. If the types are incompatible, an exception is thrown. Therefore, the `AlgElem` defines not only the method `equals()` (the template method), but also an abstract method `equalsS()`.

```
public abstract class AlgElem {
 public final boolean equals(AlgElem e)
            throws IncompatibleClassesException{
  if (this.getClass()!=e.getClass()){
   if (Conversions.getInstance().existConversion(
       this.getClass(),e.getClass())){
      AlgElem[] conver=Conversions.getInstance().convert(this, e);
      return conver[0].equalsS(conver[1]);
   }else
      throw new IncompatibleClassesException("There is not defined
       any conversion between "+this.getClass()+" "+e.getClass());
  }else
   return this.equalsS(e);
 }
 protected abstract boolean equalsS(AlgElem o);
}
```

FIGURE 4. Java implementation of the class `AlgElem`.

In the hierarchy of algebraic structures (Figure 1) there are other operations that need to use conversions: `plus(), minus(), times(), etc`. These operations take parameters(operands) which should have the same type. But there are cases when these operations could be called with parameters of different types, because the types are compatible.

The first time such an operation appears in the hierarchy, we define a template method for it. This template method calls another method that does the actual work on the parameters of the same type. The implementation is similar to that for `equals`.

This solution is quite good but it spans over many classes (`AlgElem`, `SemigroupElem`, `GroupElem`, `RingElem` and `DivisionRingElem`) and any modifications/changes to the conversions module will also have to be done in all these classes. This solution also adds a number of new methods to the algebraic structures interfaces, which increases the complexity.

If we decide not to allow conversions, or to only allow a subset of the defined conversions, we have to replace the file that contains them. No recompilation of the program is needed.

3.3. The *Aspect Oriented* Approach.

The second approach uses Aspect Oriented Programming(AOP) [1].

AOP is a new methodology that provides separation of crosscutting concerns (as logging, authorization, etc.) by introducing a new unit of modularization, the

aspect that crosscuts other modules. With AOP it is possible to implement crosscutting concerns in aspects instead of fusing them in the core modules. An aspect weaver, which is a compiler-like entity, composes the final system by combining the core and crosscutting modules through a process called weaving. The result is that AOP modularizes the crosscutting concerns in a clear-cut fashion, yielding a system architecture that is easier to design, implement, and maintain. Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing core concerns.

Conversions are concerns that are separated from operations such as `equals(), plus(), etc`. In order to integrate conversions in the system using this approach, all we have to do is define an aspect: `ConversionsAspect` that contains the calls to the conversions. For each method that might use conversions we define a pointcut and an advice. The pointcut gathers the context and other necessary information, and the advice tells what it must be done when the pointcut is reached. The code below presents the pointcut for `equals()` when AspectJ [2] is used:

```
equals(AlgElem e1,AlgElem e2): target(e1) && args(e2) &&
 execution(* AlgElem+.equals(AlgElem) throws IncompatibleClassesException);
```

We consider here as join point the execution of the method `equals()` from `AlgElem` or any of its subclasses. After that, we define an advice for it. Because this method might throw an exception when there is no conversions between the two algebraic elements, and the body of `equals()` is no longer executed, we need to use the `around()` advice:

```
Object around(AlgElem e1,AlgElem e2)
           throws IncompatibleClassesException : equals(e1,e2){
 if (e1.getClass()!=e2.getClass()){
     if (Conversions.getInstance().existConversion(
            e1.getClass(),e2.getClass())){
AlgElem[] conver=Conversions.getInstance().convert(e1,e2);
return new Boolean(conver[0].equals(conver[1]));
    }else
     throw new IncompatibleClassesException("There is not defined
         any conversion between "+e1.getClass()+" "+e2.getClass());
 }else
    return proceed(e1,e2);
}
```

The pointcuts and advices are very similar for all operations, but after a call to conversions the context might change, so we must take into consideration this situation, as well. For example, if we try to compare a real to a complex, the `equals()` method called belongs to the first parameter (which belongs to the `Real` class), but after the conversion we need to call the `equals()` method from the `Complex class`, and this is not possible using AspectJ `proceed()` statement, which remembers the states of each parameters from the advice.

If later we decide not to use conversions, we have to recompile the system without the `ConversionsAspect`. We could replace this aspect with another one to check the compatibility of the parameters type, but it is not mandatory.

Using AOP there is no need to add more methods in the algebraic structure classes, the conversion crosscutting concern is kept in one place, and if in the future more operations that need conversions are added, the only place that must be changed/modified is the aspect. If the Conversion module is changed, modifications must also be done in only one place, the `ConversionsAspect` aspect.

4. Conclusion

We have analyzed how conversions could be added to an algebraic system. The design is based on reflection and dynamic loading, and can be integrated using two approaches. The first one uses *Template Method* design pattern. This approach is easy to understand and allows adding new types and operations that might use convertions without recompilation. But it also has some disadvantages. For each operation a new template method has to be built and this increases the complexity of maintenance and extendibility. If the conversions are removed, the execution is still slown down because of the verifications done by the template methods. The second approach uses Aspect Oriented Programming. Using this approach there is no need to add new methods in the algebraic structures classes, the conversion crosscutting concern is kept in one place, and if in the future more operations that need conversions are added, the only place that must be changed is the aspect. But any addition of new types needs recompilation of the whole system. If the conversions are removed, we need to recompile the whole system, but the execution is not slown down by verifications.

References

[1] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. *Aspect-Oriented Programming*, Proceedings European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, 1997, pages 220–242.
[2] The AspectJ web site: http://eclipse.org/aspectj.
[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.
[4] Niculescu V, Moldovan G.S.,*Building an Object Oriented Computational Algebra System Based on Design Patterns*, Proceedings of 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05), IEEE Computer Press, 2005.

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

Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Cluj-Napoca, Romania
E-mail address: `grigo@cs.ubbcluj.ro`