# A Case Study in Object Oriented Programming. Algebraic Structures in Eiffel

**Peter Sanders**
Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler
Universität Karlsruhe, Postfach 6980
76128 Karlsruhe, Germany
Email: sanders@ira.uka.de

**Abstract**

This paper explores the idea of using the well established theory of abstract algebra as a testing ground for building reusable class hierarchies in a modern object-oriented programming language such as Eiffel. The paper discusses design experiences in trying to find the balance between a too-simple and a too-complex class hierarchy.

Abstract algebra yields particularly clear examples for problems with instances-creation involving type parameters and performing automatic type conversion.

Another set of problems appears when it is tried to take formal specification and subcontracting serious. It may be difficult to keep preconditions of subclass methods weaker than their specification in a superclass due to range limitations of data types, limitations of the type system and peculiarities of the application itself. There are also some problems with the use of assertions if quantification or seemingly circular specifications are necessary.

## 1 Introduction

The paradigm of object-oriented programming is increasingly replacing the conventional notion of structured programming as the state of the art in programming methodology. This paper is concerned with one particularly important aspect of object-oriented programming: How to build a class hierarchy for very fundamental data types? These data types are understood here as being application independent and specializeable for many different uses. Also the same abstract class can be implemented by a variety of concrete data structures and internal representations. These data types have a very fine-grained usage pattern in which the reused methods are called very often and where most methods will only replace a few lines of traditional code. Therefore there is a need for a very flexible, efficient and convenient interface.

Coming up with a good example for this setting is not easy. Many applications are either too simple to show interesting phenomena concerning object-oriented programming or they involve a prohibitively large amount of code.

A very important application is fundamental abstract data structures like lists, arrays or trees. They have the aforementioned properties and some basic classes are relatively easy to implement. But fundamental data structures also pose problems: Class hierarchy design is not easy because there are a lot of more or less arbitrary choices (see [Uhl 90]) that make it hard to decide if problems are due to a bad hierarchy design or due to more fundamental problems with the language or the methodology. In addition there are a number of practical constraints due to existing classes and programming conventions that might interfere with the goal of a clean and clear hierarchy design.

The mathematical structures investigated by abstract algebra (like groups, rings or vector spaces) model even more fundamental concepts than abstract data structures and therefore their implementations are also characterized by a very fine-grained code-reuse pattern and have a lot of code that is representation independent. Hierarchy design is less arbitrary because more than a century of mathematical research has provided us with a mesh of interrelated concepts whose overall structure is widely acknowledged. A closer look shows that these relations often resemble class/subclass relationships.

Abstract algebra classes are also useful. Computer algebra is becoming more and more important for a wide area of applications from cryptography to theoretical physics. Although modern computer algebra knows very complex algorithms, it is possible to implement basic operations and some interesting applications with a relatively small amount of code. Finally many data structures that are not math-specific (e.g. strings, matrices) have an underlying algebraic structure. Therefore algebraic types might also be viewed as a starting point for a systematic hierarchy of reusable data structures.

The programming language Eiffel was chosen because it has a number of features that make it interesting. It combines the discipline of strong typing with a clean design, the power of multiple inheritance and genericity. None of the other readily available languages (like Smalltalk, C++, Simula or CLOS) offer this combination of capabilities. In addition to strong typing, Eiffel offers an even more rigorous approach to high-quality software by providing assertion statements that make it possible to incorporate a formal specification into the sources. This is particularly useful in the context of computer algebra because many assertions can be taken from algebra textbooks.

### Overview

Section 2 gives a short introduction to the implemented[1] classes. The main part of the paper discusses class-hierarchy design (section 3), class-interface design (section 4) and issues involving formal design methods (section 5). It is not the purpose of this case study to work out every aspect of the design of reusable software. Instead, certain aspects have been selected, for which the algebra example appears to be particularly helpful in understanding important issues.

## 2 The Computer Algebra Example

This section introduces the mathematical background for the algebra hierarchy and the classes used to implement the mathematical concepts. Rather than describing every detail of the actual design, a simplified, less fine-grained structure is described. The resulting hierarchy is at the same time interesting enough to describe the experiences made and simple enough to present it in a compact and understandable way.

[1]All the classes discussed here have been implemented in ISE-Eiffel 2.3. Due to time constraints and problems with getting a working Eiffel 3.0 compiler the classes have not been ported to Eiffel 3.0 yet. Nevertheless the discussion in the paper is based on Eiffel 3.0.

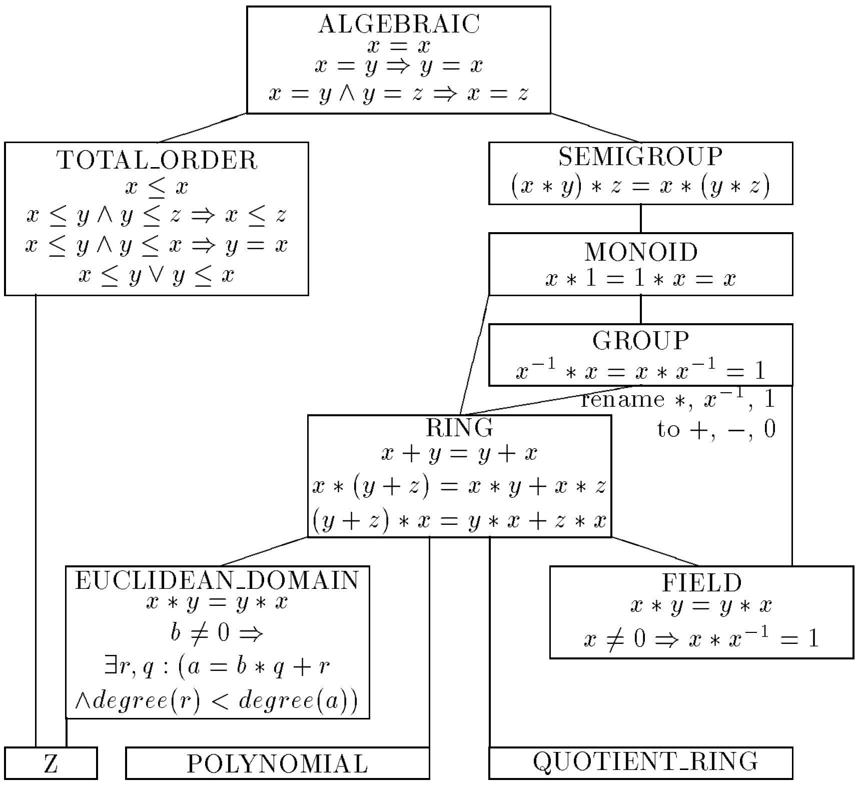
## 2.1 The Structure Suggested by Mathematics

The basic idea of doing abstract algebra is to postulate a simple set of rules (axioms) that are supposed to hold for the members of a set and to investigate the implications of these rules. This abstract approach has two very useful properties:

- Everything that is known about a structure can be applied to any specific set, given that the axioms are true for this set.
- When investigating a new structure, all theorems that have been proved for a more general structure will also hold for the new structure.

These mathematical properties of abstract algebra correspond to well known properties of the object-oriented programming paradigm. An abstract algebraic structure that is defined by a set of axioms corresponds to a deferred class having the axioms as invariants and introducing the basic operators as deferred features. All code that can be defined in terms of the basic operations will be reusable in all subclasses. Subset relations[2] between axiom sets can be mapped to subclass relations between the corresponding classes.

Figure 1 shows the algebraic structures that are discussed here. The axioms that hold for a structure and all its descendants but for none of its ancestors are displayed along with the name of the structure.



ALGEBRAIC
$x = x$
$x = y \Rightarrow y = x$
$x = y \wedge y = z \Rightarrow x = z$
TOTAL_ORDER
$x \leq x$
$x \leq y \wedge y \leq z \Rightarrow x \leq z$
$x \leq y \wedge y \leq x \Rightarrow y = x$
$x \leq y \vee y \leq x$
SEMIGROUP
$(x * y) * z = x * (y * z)$
MONOID
$x * 1 = 1 * x = x$
GROUP
$x^{-1} * x = x * x^{-1} = 1$
rename $*$, $x^{-1}$, 1 to $+$, $-$, 0
RING
$x + y = y + x$
$x * (y + z) = x * y + x * z$
$(y + z) * x = y * x + z * x$
EUCLIDEAN_DOMAIN
$x * y = y * x$
$b \neq 0 \Rightarrow$
$\exists r, q : (a = b * q + r$
$\wedge degree(r) < degree(a))$
FIELD
$x * y = y * x$
$x \neq 0 \Rightarrow x * x^{-1} = 1$
Z
POLYNOMIAL
QUOTIENT_RING

Figure 1: Algebraic class hierarchy. Class names plus axioms that are added in a given class.

The most fundamental property of **algebraic** sets is that their elements can be tested for equality. Elements of a **total order** are always comparable by a relation "$\leq$".

Another branch of the hierarchy is concerned with generalizations of the arithmetic operations. The term **semigroup** subsumes all structures with a binary operator which is always applicable (total) and where parentheses can be omitted

[2]It is sufficient that the old axioms can be deduced from the new axioms. But in most practical cases the simple subset relationship applies.

(associativity). A **monoid** additionally requires the existence of a unique neutral element which behaves analogously to the number 1 for multiplication, i.e. it does not change the result when multiplied with an element of the algebraic structure. A **group** is a monoid with an inversion function that behaves like the $1/x$-function for the nonzero real numbers.

A **ring** subsumes all structures that understand two basic operations "$*$" and "$+$" where the addition behaves like a (commutative) group and the multiplication behaves like a monoid. These two operations are connected via the law of distributivity. Many important structures of abstract algebra are specialized rings. An **Euclidean domain** is a ring with a generalized analogue to the division property known from the integers. **Polynomials** are expressions consisting of the operations "$+$" and "$*$", constants from another ring (coefficients) and formal variables.

Elements of a **quotient ring** are Euclidean domain elements that are considered equal if they yield the same remainder when divided by a given basis element. A **field** generalizes the arithmetic properties of real, complex and rational numbers i.e. it is a commutative ring with a multiplicative inverse for the nonzero elements.

The **integers** are a typical example for an Euclidean domain and at the same time they are totally ordered.

## 2.2 Basic Ideas for the Implementation

The starting point for the implementation is the idea of mapping each algebraic structure to a class. The classes *ALGEBRAIC, TOTAL_ORDER, SEMI_GROUP, MONOID, GROUP, RING, EUCLIDEAN_DOMAIN* and *FIELD* are inherently abstract and are therefore implemented as deferred classes.

The classes *POLYNOMIAL, QUOTIENT_RING* and *Z* (integers) can be implemented directly or used as abstract superclasses for several (possibly compatible) implementations. *POLYNOMIAL*[$R \rightarrow$ *RING*] is a generic class defining the polynomial ring over coefficients of the type parameter $R$. Analogously, quotient rings are implemented as the generic class *QUOTIENT_RING*[$R \rightarrow$ *EUCLIDEAN_DOMAIN*].

Besides the basic arithmetic operations a number of interesting and useful yet relatively simple algorithms of computer algebra have been implemented.

- The integer exponentiation ($x^n$) has an efficient ($O(\log n)$) implementation that works for any *SEMI_GROUP*.
- The Euclidean algorithm for determining the greatest common divisor can be defined in *EUCLIDEAN_DOMAIN*.
- The Chinese remainder algorithm which is useful for applications like fast long-integer arithmetic and polynomial interpolation can be defined to work on any collection of *QUOTIENT_RING* elements.

Beyond this relatively small number of classes[3] there is a virtually unlimited potential for expansion. On the one hand the hierarchy can be expanded sideways by including other abstract subhierarchies (e.g. vector spaces). On the other hand downward expansion yields many more specific classes. The current implementation includes implementation classes for *Z* and *POLYNOMIAL* and the fields *REAL_R*, *RATIONAL*[$R \rightarrow$ *EUCLIDEAN_DOMAIN*] and *QUOTIENT_FIELD*.

[3]The actual implementation is based on a more fine-grained design. It includes the additional abstract structures PARTIAL_ORDER, ABELIAN_GROUP and DIVISION_RING.

# 3 Designing a Class Hierarchy

A well known saying of Albert Einstein is: "Make it as simple as possible but no simpler." Translated into the design of class hierarchies this means that there is a tradeoff between small and simple and large and powerful class hierarchies. A very simple hierarchy may be too rudimentary to provide a basis for code inheritance and too rigid to be suitable for a wide spectrum of applications. On the other hand, a very complex hierarchy takes longer to develop, is hard to understand and may be complicated to use. As in physics it is desirable to start with some simple and generally applicable principles and derive a clean, easy-to-understand and powerful hierarchy from these principles.

The algebra example yielded a number of examples that are able to bring out this tradeoff in a particularly clear way.

## 3.1 Problems with Simple Hierarchies

Some typical problems of too-simple hierarchies have been encountered with the standard library of ISE Eiffel 2.3:

- The only way to input a real number is using *readreal* on a *FILE*. But often it is more desirable to get the number from a string or some other user-defined data structure. The solution would be a class similar to Smalltalk's streams that is more general than a file.
- One possible implementation of *POLYNOMIAL* is to represent a polynomial as an ordered collection of coefficients. The operations on polynomials are in principle independent of the implementation of a particular collection class. In order to allow for fine-tuning for various special purposes it might therefore be useful to leave open the choice of the collection class implementation. (By introducing a generic parameter specifying the collection class to be used.) But since there is no common interface for arrays, lists, etc., this is not possible.

These examples have in common that a lack of depth in the library is a more prominent problem than the sheer number of available classes and methods. On the other hand it is often not easy to anticipate the needs of software developers. Therefore it may be considered a fundamental constraint of the object-oriented programming paradigm that, while it is simple to expand the inheritance graph downward towards more specific classes, it is in general difficult to let it grow upward by subsuming common properties of several classes into a more general class.

## 3.2 Explosion in the Number of Classes

The class hierarchy that tries to provide for all possible applications and changes can also cause serious problems. A simple and very typical example is the integers. The subranges of positive, negative, nonnegative and nonpositive integers have differing algebraic properties that might be expressed in the class hierarchy. On the other hand integers might be implemented as short integers or as variable/constant-length long-integers with decimal, binary or residual representation. The problem is that every subrange can be combined with each implementation to yield a new class. It is a very cumbersome task to write, describe, debug or understand such a large number of classes. This problem was avoided by not using different classes for different integer ranges. As a consequence preconditions had to be used in order to check for the appropriate ranges (e.g. in the integer exponentiation which has different domains for *SEMIGROUP, MONOID* and *GROUP*). In general there seems to be a tradeoff between the complexity of a class hierarchy and the complexity of assertions.

An interesting example for problems of a similar type ocurred when the design of a subhierarchy for matrices was considered. The least that a *MATRIX* class must expect from the matrix elements is that their elements understand two *SEMIGROUP*-operations "+" and "∗" (i.e. "+", "∗" must be total associative functions). This is sufficient to define a matrix multiplication. If the additive and multiplicative identity elements *zero*, *one* are defined for the matrix elements it becomes possible to define the identity matrix. Elements of an *ABELIAN_RING* (which have a commutative multiplication) make Strassen's fast matrix multiplication possible. *FIELD* elements (for which a division is defined) allow it to use Gaussian elimination and related algorithms. Finally Gaussian elimination can be made more accurate if an appropriate comparison is defined on the elements.

Another degree of freedom arises from the different useful representations for matrices. The straightforward way is to use a two-dimensional array. Also important are various sparse-matrix representations or submatrices containing a pointer to a parent matrix. Furthermore special representations for banded matrices, homogeneous coordinate transformations, etc., might be useful.

Expressing the various combinations of representation and algebraic properties directly as a class hierarchy would be prohibitively complex. Fortunately genericity offers a solution. The different representations can be implemented as compatible subclasses of a nonalgebraic class *MATRIX_REPRESENTATION* which defines an interface for element access. The matrix classes of the algebra class hierarchy will then have a representation class as a generic parameter and the matrix will be stored in an attribute of the representation type. For a matrix class allowing for Gaussian elimination this might look like the following:

```
class FIELD_MATRIX[
        F → FIELD,
        M → MATRIX_REPRESENTATION]...
feature content: M...
```

A drawback of this approach is that it introduces an additional indirection. This does not only imply a (moderate) time and space overhead but also additional conceptual complexities that do not seem to be necessary a priori.

There are many similar cases where the problem with interactions of implementations, representations and properties occurs. In addition the setting of abstract algebra makes another problem particularly clear. In principle any set of axioms which is derivable from a basic set of useful axioms may yield an algebraic structure that is a potential superclass of the original class. Since it is difficult to add new superclasses to an existing hierarchy the designer has to anticipate all useful general classes.

## 4 Design of Class Interfaces

The design of good class interfaces for object-oriented libraries is even more difficult than for conventional libraries because it must facilitate the introduction of new classes by the user. Additionally the fine-grained pattern of reuse considered here is particularly demanding. It would be hard to convince programmers to use library functions with complicated interfaces or poor performance if only a couple of lines of source code can be saved. Again, the algebra example is a good vehicle for exemplifying problems that appear to be fairly common but are hard to separate from problem-specific "noise" in the context of other applications.

```
class POLYNOMIAL[R→RING] ...
  one: like Current is
    local anchor: R;
    do
      !!anchor; -- i l l e g a l
      Result := zero.put_coeff(anchor.one, 0);
    end ; -- one
```

Figure 2: Instance creation on a parameter type

## 4.1 Instance Creation

There are at least two different usage patterns for instance creation. One is the initialization of data structures by application programs. For this purpose creation methods should have arguments that make it possible to create adequately instantiated objects by one single call to a creation method. Consequently the interfaces of creation methods may differ between different subclasses of an abstract concept. In this context it is quite probable that several creation methods[4] may be useful for one class.

But instance creation is also necessary in generic implementations. For this purpose it is more important to have one uniform interface for all subclasses of a given class. The best choice for such a compatible creation method will often be a version without any parameters.

### Instance Creation on Generic Parameters

Instance creation in generic functions poses a severe problem in Eiffel[5] when generic parameters are involved. For example consider the generic class *POLYNOMIAL*$[R \to RING]$ and note that *RING* is deferred. Now let's assume the multiplicative identity element *one* is to be implemented. Since the creation method for *POLYNOMIAL* returns a zero-polynomial, all that is to be done is to set the 0-th coefficient to the *one* of $R$ (figure 2). But its impossible to simply call $R$'s *one*-function since no ring element is available. The next thought is to create a ring element that could in turn answer the *one* message. But this is not possible since RING is deferred. It is also impossible to simply access some coefficient of *Current* because a zero-polynomial has no coefficients. This could be changed by requiring a zero-polynomial to have the 0-th coefficient *zero* but this would only shift the problem into the creation method of *POLYNOMIAL*. How could a creation method without parameters possibly get hold of an element of $R$?

Meyer mentions problems of this type in [Meyer 92] and proposes a solution which makes clients pay a "toll" for calling a problematic method (*one* in this case) by requiring them to pass an instance of the generic parameter's class. But this only works under the assumption that all possible clients know more about the application than the supplier. In the given application this is nonsense because *POLYNOMIAL* may even be a client of itself (e.g. for multivariate polynomials)!

Meyer also points out a straightforward implementation that would allow creation of parameter-class instances. The actual type parameters are known at run-time and can be stored along with the objects. This mechanism is not used

[4]This is possible in Eiffel 3.0.

[5]And probably most other strongly typed object-oriented languages offering generic parameters.

in Eiffel because its simple minded adoption could lead to a considerable space overhead.

But a sophisticated compiler should be able to eliminate most type-tags by detecting which classes don't need this information and by trying to infer the actual type parameters at compile-time. The techniques necessary for this analysis appear to be similar to techniques for eliminating dynamic method-dispatch (see [Chambers 91]). The price paid for these optimizations is giving up separate compilation and a certain space overhead for generating several versions of some methods customized for different uses.

### 4.2 Dealing with Compatible Types

Whenever several implementations share a common interface, a variety of problems with (automatic) type conversion occur. The classical example are arithmetic operations. For example, how to implement a "+" that accepts any combination of reals, integers, rationals, etc.? This reappears in the algebra example because abstract algebra may be viewed as a generalization of arithmetic. LaLonde and Pugh [LaLonde 90] discuss some solutions for a dynamically typed language like Smalltalk. But in the presence of strong typing a number of additional problems arise.

A good example is the "$\leq$"-relation. Its parameters in *TOTAL_ORDER* are declared via declaration by association. But for $Z$ and its possible implementations this is no longer appropriate because all integers are supposed to be compatible regardless of their implementation (long integers, short integers, etc.). Therefore methods that are declared in the superclass must be redeclared (e.g. "$<$", "$\leq$", "$>$"). This is cumbersome and a possible source off errors. It will even lead to the loss of generic implementations for redeclared features whose code is still applicable. Weber [Weber 91] proposes a way to solve this problem by replacing declaration by association with a mechanism similar to generic classes. He also points out that this problem may even lead to a violation of class correctness.

Some conventions of algebra can't be grasped by traditional class relationships at all: Simpler sets are often identified with a subset of a more complex set. For example real numbers are interpreted as complex numbers with zero imaginary part or a ring $R$ is identified with the constant polynomials over $R$. In order to express this by a class relationship, *RING* would have to be made a subclass of *POLYNOMIAL*[$R \to$ *RING*] which is certainly not possible because *POLYNOMIAL* is already a subclass of *RING* (see figure 1). The *actual* instance of $R$ could be viewed as a subclass of *POLYNOMIAL*[$R \to$ *RING*]. But this can't be expressed in Eiffel.

Another solution would be overloading as in Ada or C++. An even more powerful mechanism for type conversions has been implemented for the computer algebra system ScratchPad. But besides making the compiler very complicated these mechanisms often produce unexpected sequences of type conversions that make debugging difficult. Therefore it might be viewed as an open question how far automatic type conversion should go.

# 5 Formal Design Methods

### 5.1 Class Correctness

A fundamental rule of object-oriented design is that a (re)implementation of a method must meet its specification in a superclass. This implies that the method's preconditions must never be stronger than their ancestor's preconditions. The rule proved to be a critical test for the straightforward design outlined in section 2. This test led to some redesign in the class hierarchy, not only making it sounder but also better suited for code and behavior inheritance. But there are some remaining problems .

### Dealing with Implementation Limitations

An implementation of an infinite set will often impose constraints on the range of its members (expressed as class invariants). These constraints will in turn result in additional preconditions for some features. For example integers between -128 and 127 imply the precondition $Current*Current \leq 127$ for the *square* function.[6] But since *square* is in principle a total function its precondition in $Z$ appears to be **true** which is obviously not stronger than $Current*Current \leq 127$.

One way to remove this paradox is, to introduce a predicate *in_range* that tests if an integer can be represented in the current implementation. In $Z$ it is deferred. The resulting precondition for *square* in $Z$ is *in_range*(*Current*Current*) and there is no additional precondition in the implementation. Instead the implementation defines *in_range* (as $Current \geq -128 \wedge Current \leq 127$).

### The Domain of the Division Operation

*FIELD*s are *RING*s that additionally inherit from *GROUP*. But the multiplicative group of a field happens to be not quite the same as the multiplicative monoid of the underlying *RING* because the 0 is excluded (you can't divide by 0). Expressing this difference in the class hierarchy would make it impossible to reuse any of the features from *RING*.

Instead, a predicate *is_unit* was introduced in MONOID and used as a precondition for "/" (**require** *other.is_unit*). As opposed to the *in_range* predicate discussed above which can only be understood from an implementation point of view, *is_unit* is purely algebraic notion. The only unusual thing is that *is_unit* must *not* be defined to be **true** in *GROUP*, but it has to remain deferred in order to make *FIELD* a multiplicative group.

A similar problem arises with *EUCLIDEAN_DOMAIN*. The usual definition found in algebra textbooks relies on a total division operation. On the other hand the class *POLYNOMIAL* does not have a total division operation. (Division works only for polynomials with an invertible leading coefficient.) But since *POLYNOMIAL* behaves in many ways like an Euclidean domain it should inherit from *EUCLIDEAN_DOMAIN* in one way or the other. Since it does not seem to make sense to define separate subhierarchies for invertible and noninvertible polynomials it was decided to define the division operation for Euclidean domains as a partial function. This makes it possible to make *POLYNOMIAL* a subclass of *EUCLIDEAN_DOMAIN*.

### Class Families

A *QUOTIENT_RING* contains elements of an *EUCLIDEAN_DOMAIN* that are used modulo a base element of the same type (e.g., $Z_n$, the integers modulo $n$). In fact there is a new quotient ring for each base. The obvious implementation in Eiffel stores remainder and base in instance variables of type *EUCLIDEAN_-DOMAIN* (see figure 3). But this imposes the additional constraint for all binary operations (e.g. "+") that the bases must be equal. Thus the precondition for "+" is stronger in *QUOTIENT_RING* than in its ancestors and this is illegal.

[6]Note that this precondition cannot be checked at runtime. Section 5.2 discusses this kind of problems.

```
class SEMIGROUP ...
  infix "+"(other: like Current) :like Current
    is
    require compatible(other) ...

class QUOTIENT_RING
        [R → EUCLIDEAN_DOMAIN]...
inherit RING ...
define compatible, infix "+"...
feature
  remainder, base: R; ...
  compatible(other: like Current) :BOOLEAN
    is
    do Result := base.eq(other.base)
    end ;
  infix "+"(other: like Current) : like Current
    is
    do
      !!Result.make(
        (remainder + other.remainder)
        mod base,
        base)
    end ; -- infix "+" ...
invariant
  normalized: remainder.degree < base.degree
end -- QUOTIENT_RING
```

Figure 3: Specification of "+" in *QUOTIENT_RING*

This problem can be solved by introducing a binary predicate *compatible* that checks whether a binary operation is defined for a pair of operands (see figure 3 for the "+"-operation). Although this solution works, it is a little bit dirty. The class *QUOTIENT_RING* is not really an Euclidean domain but an entire family of Euclidean domains. One class for each base.

A more satisfactory solution would be possible if the compiler would allow objects (and not only types) as generic parameters. *QUOTIENT_RING* could then be declared as

```
class QUOTIENT_RING
[R → EUCLIDEAN_DOMAIN, base : R] ...
feature remainder: R; ...
```

This would implicitly make *base* available as a constant method. The arithmetic operations could be defined as before and no additional assertions would be necessary.

An implementation of this concept is probably not trivial.[7] The compiler should do compile-time type checking wherever possible but in some cases run-time type checking would be necessary. Possibly large numbers of new classes would be generated at run-time. The minimum requirement for making this efficient will be to allow members of a class family to share method-lookup tables.

[7]A limited version of this concept is available in the C++ template mechanism [Ellis 91].

```
deferred class MONOID ...
feature infix "*" ...
-- invariant ∀x, y, z ∈ MONOID :
--                ((x * y) * z).eq(x * (y * z))...

deferred class RING
inherit
   MONOID; -- for multiplicative operations
   GROUP
      rename infix "*" as infix "+",
             infix "/" as infix "-",
             is_unit   as is_unit_group,
             inverse   as prefix "-",
             one       as zero,
             infix "↑" as mult_z ...
```

Figure 4: Multiple inheritance in *RING*

## 5.2 Assertions

Preconditions, postconditions and class invariants are useful for concisely defining and documenting the expected behavior of code. Since assertions are inherited, formal specification also profits from the object-oriented programming paradigm. Formal specification appears to be relatively easy for abstract algebra because many assertions can be taken from algebra textbooks and because the library has mainly value semantics, i.e. few methods change the state of an object.

### Quantification

Assertions using quantified expressions are not available in Eiffel. Currently quantified assertions must be put in comments. Although these comments can't be used by the runtime component, the source-code tools (`flat`, `short`) are able to propagate them to inheriting classes. Nevertheless it might be useful to explicitly include quantification in the syntax of Eiffel because more sophisticated tools might want to process assertions nontextually. One case where this would be useful is exemplified in figure 4. Since the *MONOID* operations "$*$" and *one* yield two sets of operations in *RING* ("$*$", *one* and "$+$", *zero*) the system should automatically produce new invariants for "$+$" and *zero*. For example invariant inheritance should yield two associativity laws for *RING*:

$$\forall x, y, z \in RING : ((x * y) * z).eq(x * (y * z))$$
$$\forall x, y, z \in RING : ((x + y) + z).eq(x + (y + z))$$

If there shall be a chance[8] for the source-code tools to generate these two sets of assertions, quantification must be expressible. Even more important is the possible use of quantified assertions as input for formal software development and verification tools.

### Run-time Assertion Checking

In general, run-time assertions-checking is useful for debugging because problems are reported early. But there are assertions that are useful for specification purposes only. They cannot be checked at run-time because this would lead to

[8]Current versions of `flat` and `short` do not even duplicate assertions that *are* expressible.

```
gcd(other: like Current) : like Current is
      -- greatest common divisor,
      -- one if relatively prime
   local a0, a1, t: like Current
   do
      from a0 := Current; a1 := other;
      invariant a0.gcd(a1).eq(gcd(other))
      variant a1.degree.as_integer
      loop
         t := a1;
         a1 := a0 mod a1;
         a0 := t;
      end ;
      Result := a0;
   ensure ...
   end -- gcd
```

Figure 5: Greatest common divisor

an infinite recursion. A typical example is the loop invariant of the method for computing the greatest common divisor of two instances of an *EUCLIDEAN_-DOMAIN* (Figure 5). There is nothing wrong with this assertion. It is used in algebra textbooks to prove the correctness of the gcd function. The problem is that the *gcd* in the loop invariant does not mean the method *gcd* but a value that fulfills the definition of a greatest common divisor. This distinction cannot be made in Eiffel and therefore assertion checking will run into infinite recursions if the problematic assertions are not commented out.

A possible remedy would be to introduce syntactic means for marking these *inert* functions such that the compiler will not generate code for trouble making parts of assertions. But this would further complicate Eiffel's syntax. Another approach works by switching off assertion checking inside recursive calls made by assertions. Although this might fail to evaluate all assertions that can be evaluated it seems to be a satisfactory solution. This approach can be further simplified by entirely switching off assertion checking inside assertions. In this case the compiler does not need to be able to detect recursive assertions.

## 6 Conclusions

The preceding sections have explored the consequences of trying to implement a hierarchy of algebraic data types. Abstract algebra proved to yield a lot of interesting test cases that involve a minimum of code. One of the reasons for this success seems to be that there is a close relationship between the axiomatization of a mathematical structure and a class.

An interesting experience is that formal specification of object-oriented software using Eiffel has a number of pitfalls and that it might even be useful to make some changes to the language based on these problems.

A more general experience is that the design of a class hierarchy is far from being trivial even if there seems to be a naturally given structure as in the case of the well established relations between algebraic structures. Important criteria that can help to design a suitable structure are class correctness and the number of required classes.

## 7 Acknowledgements

I would like to thank E.F. Gehringer without whom this paper would still be an unreadable project report. I am also grateful to F. Weber who pointed out solutions to many of the encountered design problems. Many ideas are also due to fruitful discussion with S. Egner, Dr. Singer and F. Ulmer.

## References

[Chambers 91] C. Chambers, D. Ungar, Making Pure Object-Oriented Languages Practical, *OOPSLA 91, pp. 1–15.*

[Ellis 91] M. A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, 1991.

[LaLonde 90] W.R. LaLonde, J.R. Pugh, *Inside Smalltalk*, Vol. 1, Prentice Hall, 1990.

[Lipson 81] John D. Lipson, *Elements of Algebra and Algebraic Computing*, Addison Wesley, 1981.

[Herstein 86] I.N. Herstein, *Abstract Algebra*, Macmillan Publishing Company, 1986.

[Klir 88] G.C. Klir, T.A. Folger, *Fuzzy Sets, Uncertainty and Information*, Prentice Hall, 1988.

[Meyer 89] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[Meyer 91] Bertrand Meyer, *Eiffel the Language*, 1991.

[Meyer 92] Bertrand Meyer, *Eiffel the Language, Prentice Hall*, 1992.

[Uhl 90] J. Uhl. *A Systematic Catalogue of Reusable Abstract Data Types*, Springer, 1990.

[Weber 91] F. Weber, Getting Class Correctness and System Correctness Equivalent — How to get Covariance Right, *TOOLS 8*, R. Ege, M. Singh, B. Meyer editors, Prentice Hall, 1992.