Categories as classes and mixin composition, Heinz Kredel and Raphael Jolly

Contents

1. generic, strongly typed, object oriented computer algebra software
2. software, algorithm implementations can be packaged and re-combined using traits in a category-like fashion

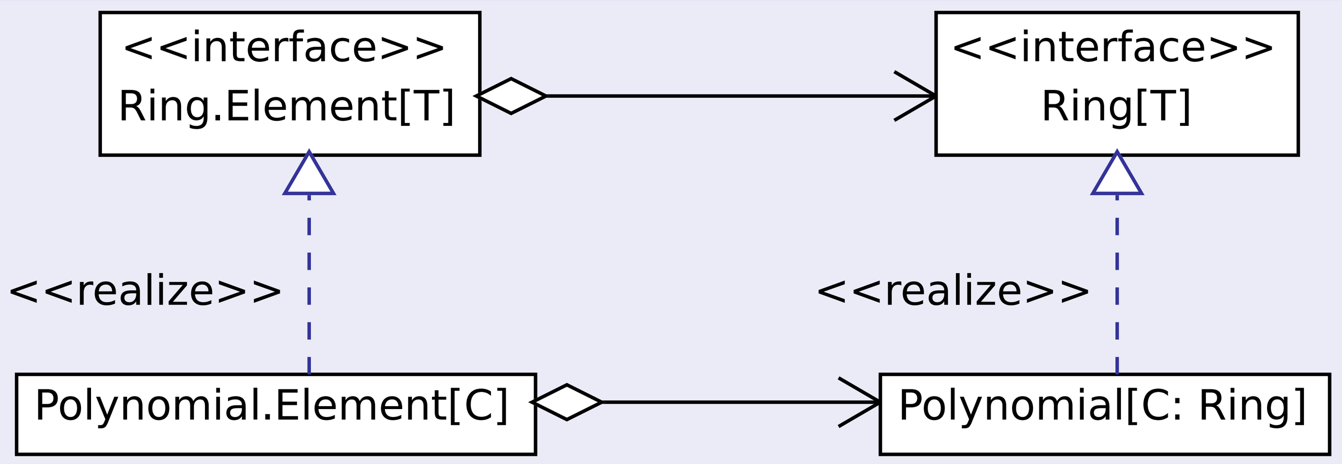
Acknowledgments: Thomas Becker, Wolfgang K. Seiler, Thomas Sturm, Axel Kramer, Jaime Gutierrez, Sherm Ostrowsky, Markus Aleksy

Introduction

The modeling of algebraic structures in a strongly typed, generic, object oriented computer algebra software has been presented with the systems JAS [5, 6] and ScAS [3]. The design and implementation of these strongly typed, generic and object oriented polynomial algorithm libraries in Java and Scala is presented in [4]. The libraries are enhanced for interactive usage with the help of the Jython and JRuby scripting languages. The libraries now provide several algorithm versions for greatest common divisor, squarefree decomposition, factorization and Gröbner bases computation in separate packages.

In this poster we discuss the problem of code organization and algebraic structure configuration and deployment. Elements of algebraic structures are implemented by classes and instantiated as objects with methods implementing the 'inner' algorithms of the structure in the programming language. The algorithm libraries, for example the construction of Gröbner bases, are kept in separate source code trees and packages. This code organization helps in the separation of the various possibilities for algorithm implementation and in the transparent selection of appropriate algorithms for a given problem. However, it is not always clear where to draw the line between 'inner' structure algorithms and 'external' library algorithms, and it is sometimes possible to implement calculation engines as part of the algebraic structures themselves. This technique is paralleled with the concept of categories as found in competing computer algebra software.

Generic, strongly typed, object oriented computer algebra software



Example

```
import scas._
import Implicits.QQ
implicit val r: Polynomial[Rational] = Polynomial.factory(QQ, "w")
val Array(w) = r.generators
val a: Polynomial.Element[Rational] = pow(w, 2) - 2
```

Algorithm libraries

- focus on multivariate polynomials over UFDs
- **greatest common divisor:** interface `GreatestCommonDivisor` with `gcd()`, `content()`, implementations for various polynomial remainder sequence (PRS) algorithms: simple, monic, primitive and the sub-resultant algorithm generic for any (UFD) coefficient ring, other implementations use Chinese remainder algorithms or Hensel lifting
- **squarefree decomposition:** interface `Squarefree`, generic implementations for finite or infinite coefficient fields or rings of characteristic 0 or p
- **factorization:** interface `Factorization`, implementation depends on the explicit coefficient ring but is generic in the sense that it can factor over arbitrary stacked coefficient field extensions, like mixed transcendental and algebraic extensions
- **factories** select appropriate algorithms for given coefficients

Code organization problem

The algebraic structures and elements together with the algorithm libraries provide a way to define precisely suitable combinations for given situations. Depending on the considered algortihms however, it can be desirable to implement calculation engines as part of the algebraic structures themselves.

Categories in computer algebra systems

1. Axiom, Aldor: abstract classes in OOP [2, 9]
2. Magma, Sage: classes with same representation [1, 8]

Mixins for category-like code organization

Reusable components [7] consist in splitting software in as many pieces as needed or possible, and to re-assemble these according to the principle of composition. Hierarchical composition and peer composition (also called mixin composition) are two variations of this principle. We illustrate their respective usage with the example of GCD computation. We consider components for each algorithm flavor and combine them using either hierarchical or mixin composition. The code samples are given in the computer language Scala using its concept of *traits*.

References

- W. Bosma, J. J. Cannon, and C. Playoust. The Magma algebra system I: The user language. *J. Symb. Comput.*, 24(3/4):235–265, 1997.
- R. Jenks and R. Sutor, editors. *axiom The Scientific Computation System*. Springer, 1992.
- R. Jolly. ScAS - Scala algebra system. Technical report, https://github.com/rjolly/scas, accessed Jan 2012, since 2010.
- R. Jolly and H. Kredel. Generic, type-safe and object oriented computer algebra software. In *Proc. CASC 2010*, pages 162–177. Springer, LNCS 6244, 2010.
- H. Kredel. The Java algebra system (JAS). Technical report, http://krum.rz.uni-mannheim.de/jas/, accessed Jan 2012, since 2000.
- H. Kredel. On a Java Computer Algebra System, its performance and applications. *Science of Computer Programming*, 70(2-3):185–207, 2008.
- M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. OOPSLA '05*, pages 41–57. ACM, 2005.
- W. Stein. *SAGE Mathematics Software*. The SAGE Group, since 2005. http://www.sagemath.org, accessed Oct 2011.
- S. Watt. Aldor. In *Computer Algebra Handbook, Springer*, pages 265–270, 2003.

Preliminary settings

```
import scas.structure.Ring // declares method plus etc.
import Polynomial.Element
trait Polynomial[C: Ring] extends Ring[Element[C]] {
  def plus(x: Element[C], y: Element[C]) = ...
  ...
}
object Polynomial {
  trait Element[C] extends Ring.Element[Element[C]]
}
```

Peer vs hierarchical composition

In hierarchical composition, work is delegated to a member ring:

```
trait GCDEngine[C: Ring] {
  val ring: Polynomial[C]
  def gcd(x: Element[C], y: Element[C]): Element[C]
}
trait GCDSimple[C: Ring] extends GCDEngine[C] {
  def gcd(x: Element[C], y: Element[C]) = // use ring.plus etc.
}
val e = new GCDSimple[BigInteger] {
  val ring = new Polynomial[BigInteger]
}
```

The same effect can be obtained through inheritance (peer composition):

```
trait GCDSimple[C: Ring] extends Polynomial[C] {
  def gcd(x: Element[C], y: Element[C]) = // use this.plus etc.
}
val r = new GCDSimple[BigInteger]
```

Delegation decouples components - inheritance increases coupling.

Mixin composition and categories

In the mixin case, we can combine several algorithms through multiple inheritance:

```
trait GCDEngineX[C: Ring] extends Polynomial[C] {
  def gcd(x: Element[C], y: Element[C]) = ...
}
trait SquarefreeEngineY[C: Ring] extends Polynomial[C] {
  def squarefreePart(x: Element[C]): Element[C] = ...
  def squarefreeFactors(x: Element[C]): List[Element[C]] = ...
}
trait FactorEngineZ[C: Ring] extends Polynomial[C] {
  def factorList(x: Element[C]): List[Element[C]] = ...
  def factors(x: Element[C]): Map[Element[C], Long] = ...
}
val r = new GcdEngineX[BigRational]
        with SquarefreeEngineY[BigRational]
        with FactorEngineZ[BigRational]
```

Then `r` represents a polynomial category. Some algorithms may need further specialization of the coefficient type:

```
trait GCDModular extends Polynomial[BigInteger] {
  def gcd(x: Element[BigInteger], y: Element[BigInteger]) = ...
}
val r = new GCDModular
```

The desired packaging can be pre-setup or chosen automatically according to the coefficient type:

```
val r = Polynomial.factory(ring, pp)
```

The `factory` method might return an object of type `GCDModular` if ring is `BigInteger` and so on. This category scheme using mixins ties together algebraic structures with some specific algorithm implementations and so solves the packaging problem.