A real world use of higher kinds in the domain of computer algebra

Raphael Jolly
Databeans, Paris, France
raphael.jolly@free.fr

ABSTRACT

This papers presents a solution to a problem which arises at the crossing of computer algebra and object oriented programming. Up to now, in an object oriented approach, some advanced yet widely used algorithms like greatest common divisors of multivariate polynomials still required a trade-off between type-safety and code duplication. This is not true anymore, since the recent progress in computer language research which brought higher-kinded types. These provide the Scala language with sufficient expressiveness as to address elegantly the polynomial GCD issue. A detailed implementation of our solution is given.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Data types and structures; G.4 [**Mathematical Software**]: Algorithm design and analysis

Keywords

Object Oriented Programming, Generic programming, Higher kinded types, Computer algebra

1. INTRODUCTION

The recent arrival of generics in general purpose, widely available computer languages such as Java [1] has triggered a new, exciting area of exploration for computer algebra researchers. This is a deserved return as, according to [5], this programming technique itself is an offspring of computer algebra : "Generic programming is invented in the first half of the 1970s as a means to reuse the code of algebraic algorithms over abstract domains, such as Gaussian elimination."

The killer application [5] of Generics in general purpose languages is the Collection framework. In computer algebra they are typically used to abstract over the base ring of polynomials. This parallel is not surprising, as polynomials are litte more than collections when it comes to act as containers for a given type of elements.

The Java Algebra System (JAS) [9] is the project that, to our knowledge, pushed this idea the furthest, with algorithms for computing Groebner bases, greatest common divisors (GCD) and factorizations of univariate as well as multivariate polynomials, over integer, rational, modinteger base rings, and also polynomial base rings (recursively), rational functions, finite fields, algebraic numbers. It also covers such areas as solvable (non-commutative) polynomials, comprehensive Groebner bases, and regular rings.

In the sequel, we are concerned with the specific topic of GCD computation. The current JAS implementation suffers from several drawbacks which are discussed at length in [6, 8, 7] and that I aim to address in this paper.

1.1 Outline

Section 2 summarizes the GCD implementation of JAS. Section 3 introduces a Scala re-implementation of JAS' classes based on higher-kinded types which nicely solves the encountered issue. Section 4 concludes.

2. GREATEST COMMON DIVISORS IN JAS

2.1 Euclid's algorithm

The GCD algorithms implemented in JAS are based on Euclid's algorithm. In the univariate case the implementation is straightforward, but in the multivariate case one has to first convert the polynomials to a univariate recursive representation, that is to say one needs to translate elements of $\mathbb{K}[x_1, ..., x_n]$ to elements of $\mathbb{K}[x_1, ..., x_{n-1}][x_n]$, see [2].

2.2 Problem statement

The problem is stated as follows [7] : "If we use our current implementation of `GenPolynomial`, we observe that our type system will unfortunately lead to code duplication. Consider the greatest common divisor method `gcd()` with the specification

```
GenPolynomial<C> gcd(GenPolynomial<C> P,
    GenPolynomial<C> S)
```

This method will be a driver for the recursion. It will check if the number of variables in the polynomials is one, or if it is greater than one. In the first case, a method for the recursion base case must be called

```
GenPolynomial<C> baseGcd(GenPolynomial<C> P,
    GenPolynomial<C> S)
```

In the second case, the polynomials have to be converted to recursive representation and a method for the recursion case must be called

```
GenPolynomial<GenPolynomial<C>>
    recursiveUnivariateGcd(
    GenPolynomial<GenPolynomial<C>> P,
    GenPolynomial<GenPolynomial<C>> S)
```

The type of the parameters for `recursiveUnivariateGcd()` is univariate polynomials with (multivariate) polynomials as coefficients. The Java code for `baseGcd()` and `recursiveUnivariateGcd()` is mainly the same, but because of the type system, the methods must have different parameter types."

2.3 Discussion

What comes first to mind, is that if we want to have only one method (say `univariateGcd()`), we must give it a fresh type parameter. We could then call it with either `C` or `GenPolynomial<C>` as type parameter. For this to be possible, we need to implement the method either as static, or in a different class. But we must take into account the fact that we'll need several flavors of Euclid's algorithm, like simple, primitive, subresultant etc. This means that a static method is very impractical. Let's then define our method in a different class. Since the polynomial factories are at hand, why not use them ? In such case, we get:

```
class GenPolynomialRing<C extends RingElem<C>>
    implements RingFactory<GenPolynomial<C>> {

  GenPolynomial<C> gcd(GenPolynomial<C> P,
      GenPolynomial<C> S) {
// if we're univariate, call univariateGcd() ;
// if not, convert to recursive representation
// and then call gcd() on the factory of the
// resulting polynomials
  }

  GenPolynomial<C> univariateGcd(
      GenPolynomial<C> P,
      GenPolynomial<C> S) {
// Euclid's algorithm
  }
}
```

To get several versions of the method `univariateGcd()` we need to subclass `GenPolynomialRing`. But if we want to remain type-safe, this in turn requires to use a self type:

```
abstract class GenPolynomialRing<T extends
    GenPolynomial<T, C>, C extends RingElem<C>>
    implements RingFactory<T> {}
```

Concrete classes can be derived from this abstract class in the following manner:

```
final class GenPolynomialRingWithSimpleGCD<
    C extends RingElem<C>> implements
    GenPolynomialRing<
    GenPolynomialWithSimpleGCD<C>, C> {}
```

At this point we must introduce the methods that will make the conversion of representation, and see how it leads us into trouble.

2.4 Conversion of representation

Refering again to [7], we first need a method to convert a distributed polynomial A to a recursive polynomial in the polynomial ring defined by a ring factory rf.

```
GenPolynomial<GenPolynomial<C>> recursive(
    GenPolynomialRing<GenPolynomial<C>> rf,
    GenPolynomial<C> A)
```

The recursive polynomial ring rf is defined in the calling method `gcd()` as a univariate polynomial ring with multivariate polynomials as coefficients.

Second, we need a method to convert a recursive polynomial B to a distributed polynomial. The target ring is simply the enclosing class' instance (`this`) so we don't need to pass it as parameter.

```
GenPolynomial<C> distribute(
    GenPolynomial<GenPolynomial<C>> B)
```

Now, if we want to use the self type, we must replace each occurence of `GenPolynomial<C>` by `T`. The problem is with `GenPolynomialRing< GenPolynomial<C> >`, which can only become `GenPolynomialRing<?, T>` where the question mark stands for something we are unable to specify.

So, as we can see, the signature of the conversion methods is such that using a self type here is impossible. Thus, we will have to move to a more expressive language than Java.

3. GCD RE-IMPLEMENTATION WITH HIGHER-KINDED TYPES

3.1 Higher-order polymorphism

Higher-order polymorphism, also called type constructor polymorphism, was introduced in Scala in 2007 [10]. It allows to use so called type constructors as type parameters. A type constructor is the means by which one gets a new type from a type parameter, as in `List : A -> List[A]`. Here `List` is the type constructor. A higher kinded type would then be defined as for instance `class MyClass[Container[X] <: List[X]]`, where `Container` could be replaced by any subtype of `List`, regardless of its type parameter, as in `new MyClass[ArrayList]` etc.

3.2 GCD implementation in Scala

The subsequent code fragments are excerpts from ScAS [4], a project with the very purpose of investigating how we could address the shortcomings of JAS using Scala.

Below we redefine our polynomial factory in Scala, with a higher-kinded type.

```
object Polynomial {
  abstract class Factory[
      T[C <: Ring[C]] <: Polynomial[T[C], C],
      C <: Ring[C]](val ring: Ring.Factory[C],
      variables: Array[String]) extends
      Ring.Factory[Polynomial[T[C], C]] {

    def split: Factory[T, T[C]]

    def gcd(x: T[C], y: T[C]): T[C] = if
        (variables.length > 1) {
      val s = split
```

```
      valueOf(valueOf(s, x).gcd(valueOf(s, y)))
    } else {
      val (a, p) = contentAndPrimitivePart(x)
      val (b, q) = contentAndPrimitivePart(y)
      multiply(gcd1(p, q).primitivePart, a.gcd(b))
    }

    def valueOf(s: Factory[T, T[C]],
        w: T[C]): T[T[C]] = {
// convert to recursive representation
    }

    def valueOf(w: T[T[C]]): T[C] = {
// convert back to distributed representation
    }

    def gcd1(x: T[C], y: T[C]): T[C] = {
// Euclid's algorithm
    }
  }
}
```

Here we define the element type `Polynomial` corresponding to the above factory.

```
abstract class Polynomial[T <: Polynomial[T, C],
    C <: Ring[C](val factory: Polynomial.Factory[
    T, C]) extends Ring[T] {

  def primitivePart =
      factory.contentAndPrimitivePart(this)._2

  def gcd(that: T) =
      factory.gcd(this, that)
}
```

These abstract classes are then implemented as follows. Notice how `PolynomialWithSimpleGCD` is used without its type parameter in the type parameters of `Polynomial.Factory`.

```
object PolynomialWithSimpleGCD {
  final class Factory[C <: Ring[C]](
      val ring: Ring.Factory[C],
      variables: Array[Variable]) extends
      Polynomial.Factory[PolynomialWithSimpleGCD,
      C](ring, variables) {

    def location = variables.length - 1

    def split = new Factory(new Factory(ring,
        variables.take(location).force),
        variables.drop(location).force)
  }
}
```

In the element type `PolynomialWithSimpleGCD`, a higher-kinded type is not needed, and `PolynomialWithSimpleGCD` is written with its type parameter in the type parameter list of `Polynomial`:

```
final class PolynomialWithSimpleGCD[ C <: Ring[
    C]](override val factory:
    PolynomialWithSimpleGCD.Factory[C]) extends
    Polynomial[PolynomialWithSimpleGCD[C], C](
    factory)
```

The interested reader is invited to consult the source code of our working prototype in [4] to see these principles in action, since the present format only allows for rough sketches.

4. CONCLUSION

Applications of higher-kinds are presently restricted to mostly language-theoretic areas such as comprehensions or parser combinators [10]. It was not obvious that they could be useful in domains closer to the real world like computer algebra. In the case of multivariate GCD computations, the key issue that requires higher-kinds is the need to convert between recursive and distributed representations. On a side note, today there exist improved algorithms that don't use conversions of representation [3]. However, implementations based on Euclid's algorithm are still useful, if only for performance comparison with the new ones. Moreover, the case of GCD computations is not isolated in computer algebra, as there are similar issues of conversion of representation in for instance polynomial factorization. We still have to investigate if our technique can be adapted to such algorithms.

5. ACKNOWLEDGMENTS

I thank Heinz Kredel for associating me to his research for the last couple of years.

6. REFERENCES

[1] G. Bracha. Generics in the java programming language. Technical report, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.

[2] J. Davenport, Y. Siret, and E. Tournier. *Computer Algebra, Systems and Algorithms for Algebraic Computation*. http://staff.bath.ac.uk/masjhd/masternew.pdf, 1993.

[3] E. Gaydar. Calculation of polynomial gcd by grobner basis. *Preprint*, 1991.

[4] R. Jolly. Scas - Scala algebra system. Technical report, http://github.com/rjolly/scas, 2010.

[5] E. Kaltofen. Challenges of symbolic computation: My favorite open problems. *J. Symb. Comput.*, 2000.

[6] H. Kredel. On the Design of a Java Computer Algebra System. In *Proc. PPPJ 2006*, pages 143–152. University of Mannheim, 2006.

[7] H. Kredel. Multivariate greatest common divisors in the Java Computer Algebra System. In *Proc. Automated Deduction in Geometry (ADG)*, pages 41–61. East China Normal University, Shanghai, 2008.

[8] H. Kredel. On a Java Computer Algebra System, its performance and applications. *Science of Computer Programming*, 70(2-3):185–207, 2008.

[9] H. Kredel. The Java algebra system (JAS). Technical report, http://krum.rz.uni-mannheim.de/jas/, since 2000.

[10] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. *ACM SIGPLAN Notices*, 2008.