*Preprint*

# Mapping multivariate polynomials into each other.

Marc Conrad[1]

[1] *The Perisic Guesthouse, Podstrana, Croatia (www.perisic.com).*
*From September 1st, 2002: University of Luton, UK*

## 1. Introduction

A straightforward approach for implementing multivariate polynomials of the form $R[x_1, \dots, x_n]$ over a ring $R$ is to implement them recursively as $R[x_1] \dots [x_n]$. That means we implement an arithmetic for univariate polynomials $S[x_n]$ for a generic ring $S$ and allow $S$ to be another polynomial ring, here $S = R[x_1] \dots [x_{n-1}]$.

This generic recursive solution suits to a programming context which allows polymorphism. Polymorphic strategies are standard in object oriented languages, but also possible in other environments, as in the C language via void$*$ pointer arithmetic, or in some Computer algebra systems which have a generic data type representing all mathematical entities (as e.g. in SIMATH (Zimmer, 1985)).

The basic arithmetic as addition, multiplication, gcd, etc. is rather easy to implement for univariate polynomials, even if the coefficient ring is only generically available. See e.g. (v.z.Gathen, 1999) for algorithms for univariate polynomials over arbitrary UFDs. Therefore the recursive implementation is especially useful for rapid prototyping or experiments in a new programming language.

However we note that for special application areas (e.g. Gröbner bases) a distributive representation of multivariate polynomials seems to be the better choice. (There, a polynomial is represented as a list of vectors where each vector $(a, e_1, \dots, e_n)$ represents a monomial $ax_1^{e_1} \cdots x_n^{e_n}$.)

For recursively represented multivariate polynomials we have

$$P_1 := R[x_1] \dots [x_n] \neq R[x_{\pi(1)}] \dots [x_{\pi(n)}] =: P_\pi \tag{1}$$

where $\pi$ is a nontrivial permutation. For some applications, for example the computation of partial derivatives for each variable, it is necessary or at least very convenient to compute the representation of a polynomial $p_1 \in P_1$ in $P_\pi$.

In the following we use the word "map" in the meaning that we calculate the representation of $p_1$ as an element of $P_\pi$. The main result of this paper is a new algorithm mapping an element $p_1 \in P_1$ to $P_\pi$.

The classical approach toward this problem is to unwind the recursion completely, then to compute the distributive representation, permute the exponent vectors and finally transform back into the recursive representation.

In the next section we describe a new algorithm which maps $p_1$ into $P_\pi$ avoiding a distributive representation. More general, the algorithm maps an element $p \in R'[x_{\pi(1)}]\dots[x_{\pi(k)}]$ into $R[x_1]\dots[x_n]$, where $k \leq n$, $\pi$ is a permutation on $\{1,\dots,n\}$, and elements of $R'$ can be mapped into $R$. The algorithm has the advantage of being short and simple in code and is therefore well suited for rapid prototyping and ad hoc implementations.

The efficiency of the new algorithm is – similar to the classical algorithm via a distributive representation – exponential in the number of variables. A detailed analysis which keeps in account the density of coefficients for each variable and different (sparse/dense) representations of *univariate* polynomials is behind the scope of this paper but experiments with the example implementation (Conrad, 2002) show a reasonable performance for practical use.

In Section 3 we show how to use the algorithm together with an exception handling mechanism to solve related problems as avoiding constructions as $R[x][x]$ or to map elements from one polynomial ring into another where the two sets of variables only have a common subset.

An example implementation can be found in the Java package com.perisic.ring (Conrad, 2002). The related web site http://ring.perisic.com contains also a small demo applet and source code of Java classes. In the last section we give some implementation remarks.

## 2. The Algorithm

In the following ring means a ring with 1, that means the ring contains a neutral element 1 of the multiplication. For convenience we discuss the algorithm in an object oriented context, that is we speak of *classes*, *methods*, etc. However as mentioned in the introduction, it could well be transferred into any language featuring another paradigm but supporting polymorphic behavior as e.g. the C language.
4cm

First we describe the context of the algorithm. An abstract base class `Ring` requires from its child classes the implementation of the basic arithmetic (addition, multiplication, ...). A second class `RingElement` stores the information about the elements of a ring. Each `RingElement` instance $a$ belongs to an instance of a child class $R$ of the `Ring`. We intuitively write for short $a \in R$. Examples for $R$ are the ring of integers, rational numbers (not included in Figure 1), or a polynomial ring. We assume that the class `PolynomialRing` extends the `Ring` and has two attributes, the `variable` and the `coefficient ring`. So, the polynomial ring $R = T[x]$ has the attributes "$x$" and $T$. Figure 1 shows an edited UML diagram of the minimal requirements for implementing multivariate polynomials over $\mathbf{Z}$ in this way. "Edited" means here that for the sake of clarity some methods are omitted.

For an element $r \in T[x]$ we write $r = \sum b_i x^i = b_0 x^0 + \cdots + b_n x^n$ with $b_i \in T$ which implies an internal representation containing the values of the $b_i$. The

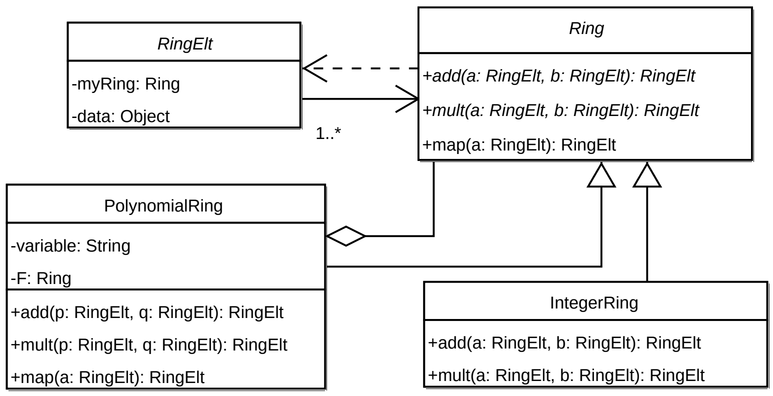
RingElt
-myRing: Ring
-data: Object
1..*
Ring
+add(a: RingElt, b: RingElt): RingElt
+mult(a: RingElt, b: RingElt): RingElt
+map(a: RingElt): RingElt
PolynomialRing
-variable: String
-F: Ring
+add(p: RingElt, q: RingElt): RingElt
+mult(p: RingElt, q: RingElt): RingElt
+map(a: RingElt): RingElt
IntegerRing
+add(a: RingElt, b: RingElt): RingElt
+mult(a: RingElt, b: RingElt): RingElt

**Figure 1:** Edited UML diagram of the ring/polynomial relationship

nature of the representation (sparse/dense/mixed) of $r \in T[x]$ is irrelevant for our algorithm.

A special method of a ring $R$ is the method `map(RingElement` $b$`)`. For a `RingElement` instance $b \in S$ where $S$ is another ring it returns $a \in R$ such that $a = \kappa(b)$ where $\kappa$ is a canonical (possibly partial defined) function $\kappa : S \to R$.

So for example the map method of the field $\mathbf{Q}$ of rational numbers will return the ring element $a = b/1$ for an integral argument $b \in \mathbf{Z}$. Vice versa, a map method of $\mathbf{Z}$ for fractions $p/q \in \mathbf{Q}$ is only partially defined, e.g. for $q = 1$.

Note that $\mathbf{Z}$ can be mapped into each ring $R$ via the mapping $\pm n \mapsto \pm(1_R + \cdots + 1_R)$ ($n$ times), where $1_R$ is the 1 of the ring $R$.

With the notation introduced above our aim is to give an algorithm for the `map` method of the `PolynomialRing` class. For this we use the following recursively defined algorithm.

**Algorithm 1**

*Input:*

- *A ring element $s \in S$ where $S$ is a ring.*
- *A polynomial ring $T[x]$ where $T$ is a ring.*

*Output:*

- *$s' \in T[x]$ with $s = s'$.*

*The algorithm: Return a value depending on the different cases below.*
*(Note: The only nontrivial case is case V.)*

Case I: *If $S = T[x]$ return $s' := s$.*

Case II: *If $S = T$ return $s' := sx^0$.*

Case III: *If $S$ is not a polynomial ring then map $s$ into $t \in T$ and return $s' := tx^0$.*

Case IV: *If $S = U[x]$, where $U$ is a ring, we have $s = \sum a_i x^i$. In this case map each $a_i \in U$ to $b_i \in T$ and return $s' := \sum b_i x^i$.*

Case V: *If $S = U[y]$ with $y \neq x$ and $U$ a ring, we have $s = \sum a_i y^i$. Map $y \in \mathbf{Z}[y]$ to $y_T \in T$ and map $a_i \in U$ into $c_i \in T[x]$. Let $y_{T[x]} := y_T x^0 \in T[x]$. Return the result of the computation $s' := \sum c_i y^i_{T[x]}$.*

Note that with $S = R[x_{\pi(1)}]\ldots[x_{\pi(k)}]$, $T = R[x_1]\ldots[x_{n-1}]$ and $x = x_n$ we obtain the situation of the introduction.

**Theorem 1** *Algorithm 1 is correct.*

*Proof:* From the construction it is straightforward by checking each single case that the algorithm delivers the correct result if it terminates. So it remains to show that the algorithm in fact terminates. In particular we have to show that Case V does not lead to an infinite recursion. We prove this by induction to $l = l(T[x], S) := n + m$ where $n$ is the number of variables of $T[x]$ and $m$ is the number of variables of $S$ with $m = 0$ if $S$ is not a polynomial ring.

- For $l = 1$ we have $n = 1$ and $m = 0$. This means we are in one of the cases II or III and $T$ is not a polynomial ring. In these cases the algorithm obviously terminates.
- Assume now $l > 1$. For $m = 0$ we are in case III. By induction we know that $s$ can be mapped into $T$ and therefore the algorithm terminates in this case. For $m > 0$ the critical case is case V. But in this case we have $l(\mathbf{Z}[y], T) = n < l$ and $l(U, T[x]) < l$ and therefore the algorithm terminates by induction.

q.e.d.

## 3. Related Problems and Extensions

Modern languages as C++ and Java usually have an exception handling mechanism. This means that in an error situation the program does not necessarily terminate. Instead an exception is *thrown* which can be *catched* and dealt with. See e.g. (Deitl, 1999) or any other Java/C++ textbook for details.

In the following we assume that the `map` method of a ring is implemented such that an error is thrown in the case that there is no canonical map from the input parameter into the ring. In fact we need only the minimal requirement that a ring $R$ which is not a polynomial ring throws an error if we try to map a polynomial into $R$.

With this extension we obtain easily an algorithm for avoiding constructions as $\mathbf{Z}[x][y][x]$, that means an ambiguity of variable names.

**Algorithm 2** *Input:*

- *A ring $T$ and a variable $x$.*

*Output:*

- *true, if $x$ is a variable of $T$ and false otherwise.*

*The algorithm:*

1. *Try to map $x \in \mathbf{Z}[x]$ into $T$ via Algorithm 1.*

*2. If no exception is thrown return true. Otherwise return false.*

Another useful extension of Algorithm 1 is the introduction of a new Case IVa which is checked before Case V.

**Algorithm 1'** *Extend Algorithm 1 with an additional case between Case IV and Case V:*

Case IVa: *If* $S = U[y]$ *with* $y \neq x$ *and* $U$ *a ring, and, in addition,* $s = uy^0$ *with* $u \in U$*, then map* $u$ *into* $s' \in T[x]$ *and return* $s'$*.*

This additional case allows the mapping of polynomials of polynomial rings, where the sets of variables only have a common subset. So, for instance we can map $z+yz^2 \in \mathbf{Z}[x][y][z]$ into $\mathbf{Z}[z][a][y]$ because $x$ does not occur in the polynomial $z + yz^2$.

## 4. Implementation Remarks

The Java package com.perisic.ring (Conrad, 2002) contains an example implementation of Algorithm 1 together with the extensions of Section 3 (exception handling, Case IVa). The source code of all classes is free for scientific purpose. There are some minor modifications, which are listed in the following. Please see (Conrad, 2002) for details.

1. There exists an additional method which parses a string and converts it into a polynomial. This method is used in Case V. Instead of mapping $y \in \mathbf{Z}[y]$ into $T$, the string "$y$" is mapped into T via the String mapping method.
2. The class diagram in Figure 1 is edited. In reality the classes contain many more methods and attributes, as division, gcd, etc.
3. In the package the class `RingElt` is abstract and has child classes `PolynomialRingElt`, `IntegerRingElt` etc. which contain the data. The polynomials are stored in dense representation as arrays of ring elements which is sufficient for the experimental nature of the package.
4. Additional rings are available in the package, for instance rational numbers, complex numbers, cyclotomic fields, and modular rings.

## References

H. M. Deitl, P. J. Deitl, *Java: How to program*, Prentice Hall, New Jersey, (1999).

J. v. z. Gathen, J. Gerhard,
*Modern Computer Algebra*, Cambridge University Press, Cambridge, (1999).

M. Conrad, com.perisic.ring. *A Java class package for multivariate polynomials*, http://ring.perisic.com, (2002).

H. G. Zimmer et al., SIMATH. *A computer algebra system for algorithmic number theory*, http://www.simath.info, (1985).