An Algorithm for Permuting Variables of Recursively Represented Polynomials

Marc Conrad[1] and Susanne Schmitt[2]

[1] Marc.Conrad@luton.ac.uk, University of Luton, Park Square, Luton LU1 3JU
[2] sschmitt@mpi-sb.mpg.de, MPI für Informatik, Saarbrücken

Abstract. A straightforward way to represent multivariate polynomial in software is to implement them recursively as univariate polynomials over a polynomial ring. This is especially common in an object oriented context. We present a short algorithm which maps polynomials from one polynomial ring to another polynomial ring where the order of variables is permuted. This algorithm uses the recursive representation and does not change to another representation.

1 Introduction

There is no representation of multivariate or univariate polynomials in software that is optimal for all applications. For instance it is well known that for the computation of Groebner bases, polynomials are best represented in a distributed format as the major operations are performed with respect to monomials [4]. *Distributed* means here that a polynomial is represented as a sum of monomials, where each monomial is the product of powers of variables and a coefficient in a base ring R. In the context of highly parallel SIMD (Single Instruction Multiple Data) architecture a variant of TFD (Table of Finite Differences) has been proven to be useful [8]. In some applications the representation is dictated by the problem (rather than by the algorithm that solves that problem). For instance polynomials may be the result by an implicit construction. The only accessible operations are then to evaluate the polynomial on test values, and problems that are simple in other representation like checking for the polynomial being identical zero require sophisticated approaches [5]. In the following we ignore theses "Black Box" problems and assume that the coefficients are known and can be used for representation.

The conceptually simplest way to represent an *univariate* polynomial $f(x) = \alpha_n x^n + \cdots + \alpha_1 x + \alpha_0 \in R[x]$ is to store the coefficients $\alpha_n \in R$ using a vector containing either the coefficients $\alpha_n, \ldots, \alpha_1, \alpha_0$ (dense representation) or the pairs (k, α_k) with $\alpha_k \neq 0$ (sparse representation). We call R the coefficient ring of the polynomial ring.

Polynomials appear in a number of application domains, and we find a variety of possible base rings R, for instance complex numbers $\mathbb{C}$ [7], modular integers $\mathbb{Z}/m\mathbb{Z}$ [1], and integers $\mathbb{Z}$ as well as real numbers $\mathbb{R}$ [4]. Hence it is good practice to implement polynomials generically, for instance – in an object oriented context as in [2] and [6] – by using an abstract class for the base ring R.

An advantage of a generic approach is that an implementation of generic univariate polynomials includes multivariate polynomials with no additional effort. For instance the polynomial ring over two variables a and b, namely $R[a,b] \cong R[a][b] \cong R[b][a]$ can be obtained as the univariate ring of polynomials over the variable b with coefficients in $R[a]$. The problem here is that the rings $R[a][b]$ and $R[b][a]$, although isomorphic, are unrelated in their internal representation. In the literature there exists so far no algorithm that directly maps elements from $R[a][b]$ to $R[b][a]$ without changing from a recursive representation to a distributed (or any other) representation. We close this gap by presenting a recursive algorithm that works on an arbitrary number of variables. The strength of the algorithm is that it is simple in code and is therefore well suited for prototyping and ad hoc implementations.

In the general situation we consider multivariate polynomials of $R[x_1, \ldots, x_n]$ over a ring R that are implemented recursively as elements of $R[x_1] \ldots [x_n]$. 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 $S = R[x_1] \ldots [x_{n-1}]$.

For recursively represented multivariate polynomials we have by construction

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

when π is a nontrivial permutation. In order to perform calculations in the isomorphism class $R[x_1, \ldots, x_n]$, that means the ring of all polynomial rings of (1) modulo permutation of variables we need to compute the representation of a polynomial $p_1 \in P_1$ in P_π. Note also that switching between different representations is unavoidable when performing operations which are dependent on a specific variable as e.g. the computation of partial derivatives.

In the following we use the word "map" in the meaning that we calculate the representation of a ring element as an element of another ring. The main result of this paper is a new algorithm mapping an element $p_1 \in P_1$ to P_π. More general, the algorithm maps an element $p \in R'[x_{\pi(1)}] \ldots [x_{\pi(k)}]$ into $R[x_1] \ldots [x_n]$, where $k \leq n$, π is a permutation on $\{1, \ldots, n\}$, and elements of R' can be mapped into R. Section 2 introduces the algorithm and we prove its correctness. The complexity of the algorithm based both on theoretical discussion and experimental results is evaluated in Section 3.

In Section 4 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.

In the last section we give some implementation remarks. An example implementation can be found in the Java package com.perisic.ring [2]. The related web site http://ring.perisic.com contains also a small demo applet and source code of Java classes.

2 The Algorithm

In the following a ring means a ring with 1. First we describe the context of the algorithm. An abstract base class `Ring` requires from its child classes the

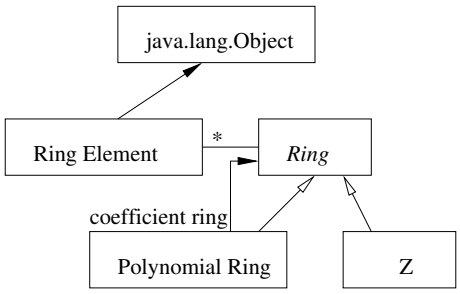

Fig. 1. UML diagram of the ring/polynomial relationship

implementation of the basic arithmetic (addition, multiplication, ...). A second class `RingElement` stores the information about the elements of a ring. Each `RingElement` instance a is associated 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 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 UML diagram for the classes that implement multivariate polynomials over $\mathbb{Z}$ in this way.

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 $b_i \in T$ hence implying an internal representation containing the values of the b_i.

Each ring R has a 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 κ is a canonical (possibly partial defined) function $\kappa : S \to R$.

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

Note that $\mathbb{Z}$ 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 (Map).
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* $y \in \mathbb{Z}[y]$ *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* $s' := \sum c_i y_{T[x]}^i$.

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

Theorem 2. *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 $l = l(S, T[x]) := m + n$ 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 $l(\mathbb{Z}[y], T) = n < l$ and $l(U, T[x]) < l$ and therefore the algorithm terminates by induction.

□

3 Complexity of the Algorithm

Algorithms for the arithmetic of multivariate polynomials in recursive representation are obviously exponential in the number of variables, as the simple task to lookup all coefficients is exponential in the number of variables. However in applications involving a large number of variables usually most of the coefficients are equal zero (because otherwise it would be hopeless to get any result at all) leading to considerable shortcuts for all algorithms.

Hence, for understanding the performance of Algorithm 1 we choose the strategy to relate the complexity of our algorithm to the complexity of a simple operation on multivariate polynomials, namely addition. For a polynomial p let the **density** $m(p)$ be the number of monomials in distributed representation.

The main interpretation of the theorem that follows is that Algorithm 1 is feasible for a polynomial p whenever the Addition of two polynomials similar

to p (i.e. having the same density as p) is feasible as well. For simplicity we assume a slightly less general situation than in Algorithm 1. After the proof of the theorem we will discuss some quantitative experimental results.

Theorem 3. *Let R be a ring, $\pi : \{1, ..., n\} \to \{1, ..., n\}$ be a permutation, $S_n = R[x_{\pi(1)}] \dots [x_{\pi(n)}]$ and $T_n = R[x_1] \dots [x_n]$. For a polynomials $s \in S_n$ let $d(s)$ be an upper bound for the degree of all univariate polynomials involved in the recursive representation and $m(s)$ be the density.*

The complexity of Algorithm 1 (i.e. mapping $s \in S_n$ into T_n) is

$$O((d(s) + 1)A(n, m(s))),$$

where $A(n, m)$ is the complexity of the addition of two polynomials in n variables with density at most m.

Proof. In the following we write $f \sim g$ for $f = O(g)$. Let $C(i, n)$ be the complexity of mapping an element of S_i (of fixed density $m = m(s)$ and upper bound $d = d(s)$) into T_n.

The most expensive of the cases is case V. Here we get the recursive formula

$$C(i, n) \sim (d + 1)C(i - 1, n) + (d + 1)A(i, m),$$

where (in the notation of Algorithm 1) the $C(i - 1, n)$ term comes from the mapping of the coefficients a_j of $s = \sum a_j y^j$ into $T[x]$. The cost of the mapping of the monomial y into T and other operations (multiplication by a monomial) are dominated by $C(i - 1, n)$.

Noting that $C(0, n)$ is linear in n and therefore dominated by $A(n, m)$, we get

$$C(n, n) \sim \sum_{i=0}^{n} (d + 1)^{i+1} A(n - i, m).$$

The complexity of addition increases by a factor $d + 1$ when one variable is added, that means we have $A(k, m) \sim (d+1)A(k-1, m)$. This proves $C(n, n) \sim (d + 1)A(n, m)$ □

For an experimental investigation of the quantitative relationship between mapping and addition we used the Monte-Carlo approach to test the algorithm on random input data. The test program is written in Java. We run the test (using the Java 1.5 virtual machine) twice on two different machines, first on an Intel Celeron 600 MHz under Windows 98 and then on an Intel Pentium 1.80 GHz processor under Windows 2000. In the following we relate the data to the Celeron and give the corresponding values for the Pentium in brackets. We will see that the values are very similar which indicates that the results are largely independent from the absolute speed of the environment.

For 1000 (26664) polynomials we computed the time for mapping of the polynomial to another polynomial ring and the time for addition of the polynomial to itself. The data that was recorded for each polynomial p are the number of variables $n(p)$, the upper bound $d(p)$ and the density $m(p)$ from Theorem 2, and

the quotient $\rho(p) := M(p)/A(p)$ where $M(p)$ is the time for mapping the polynomial ring in another ring with randomly permuted variables and $A(p)$ the time for adding the polynomial to itself. The polynomials are randomly constructed in the range $3 \leq d(p) \leq 13$, $2 \leq m(p) \leq 202$ and $3 \leq m(p) \leq 33$. The output is a comma separated list that can be used as data source for a spreadsheet program (as e.g. Microsoft's Excel in our example). The program is available upon request [3]. The source code of the Java library com.perisic.ring [2] used in the tests is available as open source on http://ring.perisic.com.

As results we get $0.001 < \rho(p) < 34.5$ ($0.001 < \rho(p) < 38.1$). The average of $\rho(p)$ over all polynomials p is 13.8 (14.1) and the median is 12.7 (13.4). We think that the median is more relevant in this context as there can be exceptional values in the test data that may have been produced by other processes, e.g. the Java garbage collection or system processes. The statistical correlation between $\rho(p)$ and $d(p)$ is 0.528 (0.55), that can be interpreted that the formula $\rho(p) \sim d(p)$ identified in Theorem 2 is valid but that there are also other factors (as e.g. the entropy of the permutation) to be considered. There is no correlation between $\rho(p)$ and the $m(p)$ or $n(p)$ (that means the correlation is close to zero). This is again in accordance to the result of Theorem 2.

For the quotient $\rho(p)/d(p)$ we obtain $0.0001 < \rho(p)/d(p) < 5.172$ ($0.0001 < \rho(p)/d(p) < 5.890$). A closer analysis shows that the values close to zero come from polynomials where the permutation is the identity (hence the mapping is trivial). If we ignore these cases we get that $0.48 < \rho(p)$ ($1.01 < \rho(p)$). These values illustrate the constants that are implicit in the Big-O notation. The median of all values $\rho(p)/d(p)$ is 1.963 (1.974) and the average is 2.035 (2.046).

Observation 4. *Heuristical data suggests that the complexity of Algorithm 1 is in average $2(d+1)A(n)$ where d is an upper bound for the degree of all univariate polynomials involved in the recursive representation and $A(n)$ is the complexity of the addition of two polynomials as in Theorem 2.*

4 Related Problems and Extensions

First we consider the problem of ambiguity of variable names. We want to avoid constructions as $\mathbb{Z}[x][y][x]$. Therefore we implement the `map` method of a ring such that an exception 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 exception if we try to map a polynomial into R.

Algorithm 5 (Safe map).
Input:

- *A ring T and a variable x.*

Output:

[3] `Marc.Conrad@luton.ac.uk`

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

The algorithm:

1. *Try to map $x \in \mathbb{Z}[x]$ 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. 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 $z + yz^2 \in \mathbb{Z}[x][y][z]$ into $\mathbb{Z}[z][a][y]$ because x does not occur in the polynomial $z + yz^2$.

Algorithm 6 (Extended map).
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'.*

5 Implementation Remarks

The Java package com.perisic.ring [2] contains an example implementation of Algorithm 1 together with the extensions of Section 4 (exception handling, case IVa). The package has been designed to provide an experimental conceptualization of Mathematics in software [3]. It is distributed under the GPL and available from http://ring.perisic.com. There are some minor modifications differences between the algorithm implemented in that package and Algorithm 1 that are listed in the following. Please see [2] 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 $y \in \mathbb{Z}[y]$ into T, the string "y" is mapped into T via the String mapping method. Both ways are conceptually equivalent.
2. 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.
3. Additional rings are available in the package, for instance rational numbers, complex numbers, cyclotomic fields, and modular rings.

References

1. D. Barrington, R. Beigel, S. Rudich, *Representing boolean functions as polynomials module composite numbers*, Proc of the 24th annual ACM symposium on Theory of computing, pp. 455–461 (2004).

2. M. Conrad, *com.perisic.ring – A Java class package for multivariate polynomials*, http://ring.perisic.com.
3. M. Conrad, T. French, *Exploring the synergies between the object-oriented paradigm and mathematics: a java led approach*, International Journal of Mathematical Education in Science and Technology, Vol 35, No.5,733-742 (2004).
4. J. v. z. Gathen, J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, Cambridge (1999)
5. R. Lipton, N. Vishnoi, *Deterministic Identity Testing for Multivariate Polynomials*, Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 756–760 (2003).
6. V. Niculescu, *A Design Proposal for an Object Oriented Algebraic Library*, Studia Universitatis "Babes-Bolyai", Informatica XLVIII, No. 1, 2003
7. A. Sommese, J. Verschelde, C. Wampler, *Numerical Factorization of Multivariate Complex Polynomials*, Theoretical Computer Science **315**, 2–3, 2004.
8. E. Zima, *Mixed representation of polynomials oriented towards fast parallel shift*, Proc. of the second international symposium on Parallel symbolic computation, pp. 150–155 (1997).