A Note on Data Types Supporting Efficient Implementations of Polynomial Arithmetics

Joachim Apel
Universität Leipzig, Institut für Informatik
Augustusplatz 10–11, D–04109 Leipzig, Germany
E-mail: apel@informatik.uni-leipzig.de

Uwe Klaus
Hochschule für Grafik und Buchkunst Leipzig
Wächterstr. 11, D–04107 Leipzig, Germany
E-mail: uklaus@hgb-leipzig.de

Abstract

There are discussed implementational aspects of the special-purpose computer algebra system FELIX designed for computations in constructive algebra. In particular, data types developed for the representation of and computation with commutative and non-commutative polynomials are described. Furthermore, comparisons of time and memory requirements of different polynomial representations are reported.

1 Introduction

Developing our special-purpose computer algebra system FELIX we have payed many attention to the space and time efficiency of representing polynomials. The system is specially designed for computations in and with algebraic structures and substructures. The basic domains implemented so far are commutative polynomial rings, free non-commutative algebras, quotient rings, and finitely generated modules. [AK91b] gives a more detailed overview about the algebraic capabilities.

For simplicity we will call both, elements of polynomial rings and of non-commutative algebras, polynomials. Crucial for all applications is an efficient implementation of the polynomial arithmetics. Since Buchberger's algorithm plays a central role in the system we have to consider also operations related to term orderings and matching problems. This is the reason why we put many efforts in investigating different data structures for monomials and polynomials.

In this paper we want to report about different investigated possibilities of data representation. We will not deal with the design of new fast arithmetic algorithms. Our subject is to develop data types which are especially adapted to the requirements of the common algorithms. In order to obtain a maximal speed up we employed hardware features as well as low level programming. Such a data type consists of the definition of the associated memory area, the specification of some basic functions acting on the data, and the description of the memory management for this data type.

Many applications in computer algebra are extremely space consuming. One possibility to reduce the space expansion is to keep data unique, i.e. to ensure that an object is stored only once in the memory and another request provides a link to the old object. But the more complex objects are involved, e.g. lists or graphes, the higher is the management overhead caused by performing existence checks. Moreover, it is strongly advisable to avoid any destructive data manipulation. Our compromise was to apply unique representation strategies to the atomic data types but not to lists and arrays.

In Section 2 there are given different possibilities for representing polynomials. Both the question of building up normal forms from an algebraic point of view and the question of the internal representation of sums of terms are discussed. All the facts stated in that section are well-known and are included in the paper mostly for completeness.

The Sections 3 and 4 deal with questions concerning data types designed for exponent vectors of commutative terms and for words representing non-commutative terms. The decision for a particular data structure can always be only a compromise since the time and space behaviour depends on the proportions between the use of the different operations, on the number of variables, on the number of terms of the polynomials, etc..

The efficiency of data types will be compared by analyzing the computation of some rather complex examples. All computing times are measured on a Sun SPARCstation10 Model M51.

2 Polynomial representation

Before starting to describe implementational tricks we will discuss possible normal forms of elements of a polynomial ring $R = A[x_1, ..., x_n]$ from the algebraic point of view.

First of all, we want to state that R is an A-module. If A is a field it is even an A-vector space. The most common way to represent the elements of a polynomial ring in a computer algebra system is to utilize this module structure. The polynomials will be expanded with respect to the module basis formed by the elements $x_1^{i_1} \cdots x_n^{i_n}$. Such a normal form is called *distributed representation*.

Caused by the ring isomorphism $R \simeq R' = A[x_1, ..., x_{n-1}][x_n]$ it is possible to consider the polynomials as elements of the ring R'. Using the distributed representation in R' and representing the coefficients recursively also by this method provides a second normal form of polynomials called *recursive representation*.

If in addition A is a unique factorization domain then the same applies to the ring R. Presumed that the factorization in R may be carried out algorithmically there is another possibility of representing the elements of R in a unique way, namely, as the product of its factors.

In order to ensure uniqueness of the representations the module basis of R has to be ordered and in case of factorized representations additional conventions about the use of units have to be made.

Factorized normal forms are rather compact and can save a lot of memory. However, the management overhead, in particular the computation of normal forms, is incredible time consuming and, therefore, this kind of normal forms will not be considered in this paper.

The main advantage of the recursive representation is the possibility to introduce new variables during a session without reorganizing previously computed expressions.

Therefore, the recursive representation of polynomials is used very often in computer algebra systems which are working without explicit declaration of the algebraic working domain. The distributed form of a polynomial is preferable for many arithmetic operations. Some algorithms, e.g. if Gröbner basis techniques are involved, strongly depend on term orderings. In this case the distributed representation is almost unavoidable.

Now, we consider the more technical details of the internal representation of a polynomial. Independent on recursive or distributive normal form a polynomial appears as a sum of monomials where each monomial is a product of a coefficient and a module basis element. If the ordering of the basis elements is equivalent to the natural numbers, i.e. for any basis element there are only finitely many lower ones, then the polynomial can be characterized simply by the sequence of its coefficients. For instance, all degree compatible orderings, i.e. such orderings where among two basis elements of different degree always that of lower degree is the smaller one, have the assumed property. But there are also a lot of important (even noetherian) orderings which do not satisfy the demand, e.g. pure lexicographical orderings. If a term ordering is equivalent to the natural numbers then the associated basis element can be deduced from the position of its coefficient in the sequence. In order to get a finite object this sequence will be truncated at the largest monomial with non-zero coefficient. This method has the advantage that no space is required for storing the basis elements. Furthermore, there are fast implementations for the arithmetic operations. On the other hand, also zero coefficients have to be stored which affects wasting memory and computing time in case of sparse polynomials.

The number of basis elements satisfying a certain degree limit indicates that dense polynomials are almost of no interest in multivariate applications. For instance, there are $\binom{d+n}{d}$ terms of degree not larger than d in the polynomial ring in n indeterminates, i.e. in the case of 10 indeterminates we have already 184,756 terms with degree up to 10. In the case of non-commutative polynomials the situation is still worse. There are n^d terms in n indeterminates of exactly degree d. The implementation of a data type should reflect this fact by representing a polynomial as a sequence of its non-zero monomials given by the coefficients and the associated terms. The question of the internal representation of terms will be discussed in the next two sections. The internal structure of the coefficients will not be considered here since it depends on the coefficient domain, e.g. a ring of numbers or a polynomial ring.

Neglecting the question of a tag for the data type of the object there remain two principle ways of storing polynomials. First there can be used lists (see Figure 1) of monomials. The second variant is to arrange the monomials in an array. An array, in FELIX also called packed list, is a coherent memory block of a certain length (see heap entry in Figure 2). The array method saves a lot of memory since pointers to succeeding monomials are not necessary. Furthermore, the array method allows to access any term of the polynomial in a constant time while the average time required for the list representation depends linearly on the number of terms. Working with Gröbner bases it might be useful to be able to reorder previously computed expressions with respect to a new term ordering. FELIX uses the quick sort algorithm which is known to have minimal complexity for sorting problems. The array structure fits much better to the required inplace changes and access to arbitrary elements than the list form. Moreover, sorting a list it is even better to convert it first into an array, sort it, and then to convert it back. Table 1 gives a comparison of times necessary to reorder

Figure 1: Representation of the sequence (< element #1 > , ... , < element #n >) as a binary tree using n node cells

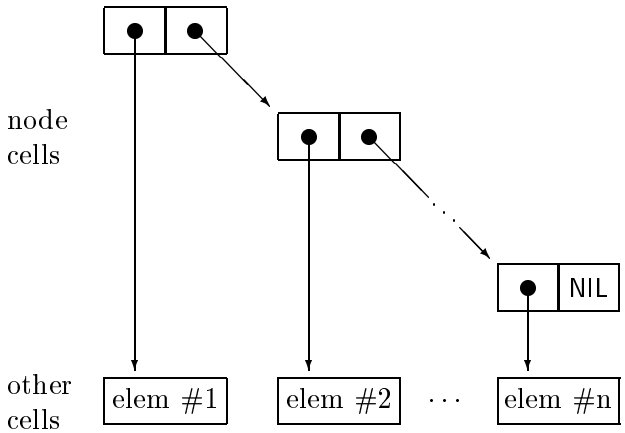
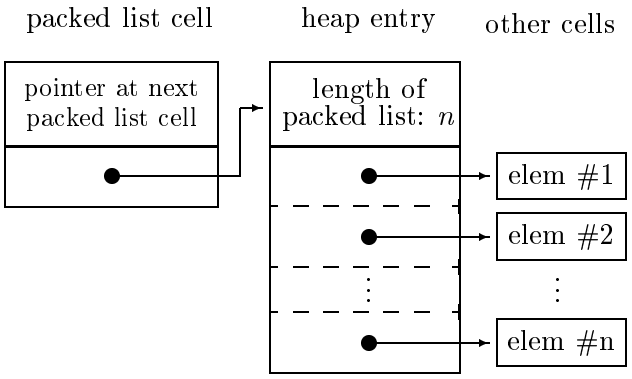


Figure 2: Representation of the sequence (< element #1 > , ... , < element #n >) as a packed list



polynomials. The polynomials were computed with respect to the degreewise reverse lexicographical term ordering. Afterwards they have been reordered according to the reverse lexicographical (revlex), lexicographical (lexic), and the total degree (totdeg) term ordering.

A disadvantage of the array representation is that the computation of the tail of a polynomial depends linearly on the number of terms since all terms but the first have to be copied into a new array. In contrary, this operation can be executed in constant time for list structures. Furthermore, two polynomials which have the smallest terms in common can share this terms in the list representation. Similar arguments on the time and space behaviour apply to the addition of polynomials when the sum and one of the summands coincide in a certain number of smallest monomials.

Table 1: Times for reordering $(1 + x_1 + x_2 + x_3 + x_4 + x_5)^n$ (in sec.)

n	# of terms	packed list representation			list representation		
		revlex	lexic	totdeg	revlex	lexic	totdeg
10	3003	1.3	1.5	1.0	20.5	27.0	13.9
11	4368	2.3	2.1	2.1	33.5	54.3	33.5
12	6188	3.1	3.0	2.6	80.5	118.5	86.8
13	8568	4.5	4.8	4.2	193.9	206.9	162.1
14	11628	6.2	7.4	6.0	353.6	473.8	268.3

Direct access to arbitrary monomials is not required within the arithmetic operations. In principal, it is sufficient to consider the operations of adding two polynomials and of multiplying a polynomial by a monomial. All other operations such as subtraction, multiplication, and in a certain sense also division can be reduced to these two elementary operations. Addition is performed in a zipper like way and multiplication by a monomial requires only going once through the polynomial. The aptitude of list and array representation is almost equal. There is one particularity of the array method. A priori there is known only an upper bound for the length of the sum of two polynomials. In order to do not waste heap space the system FELIX includes a special array operation which allows to adjust the resulting heap entry to the correct length afterwards.

The same remarks apply to distributed normal forms in free non-commutative algebras $A\langle x_1, ..., x_n\rangle$. Recursive or even factorized forms are not available for algebraic reasons.

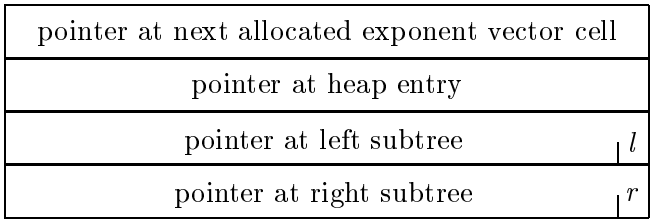
3 Representation of commutative terms

Fixing the sequence $x_1, x_2, \ldots, x_n$ of indeterminates a term $x_1^{j_1} x_2^{j_2} \cdots x_n^{j_n}$ is uniquely determined by the n-tuple of natural numbers $(j_1, j_2, \ldots, j_n)$. The concrete indeterminate names are only important for input and output operations. The information about them is part of the definition of the polynomial ring and each polynomial has a link to the ring it belongs to. In the following we discuss some possibilities for the representation of integer vectors.

3.1 Number lists

The most common possibility is to use a list representation for vectors, where we distinguish between the dense and the sparse form. The dense representation is the sequence of the n natural numbers $j_1, j_2, \ldots, j_n$. The position of a certain number in the sequence determines the indeterminate for which the number is the exponent. The sparse representation involves only the non-zero components of the vector. The term 1 is represented by the empty list and the term $x_{i_1}^{j_1} \cdots x_{i_m}^{j_m}$, where $j_k > 0$ for $1 \leq k \leq m$, by the list of pairs $((i_1, j_1), \ldots, (i_m, j_m))$. Depending on the number of indeterminates the one or the other form is preferable. Some tests were reported in [AK92]. The larger the number of indeterminates the better gets the sparse representation. This is not only valid from the point of view of memory requirements but considering computing times as well.

Figure 3: Exponent vector cell with AVL-tree pointers



3.2 Exponent vectors

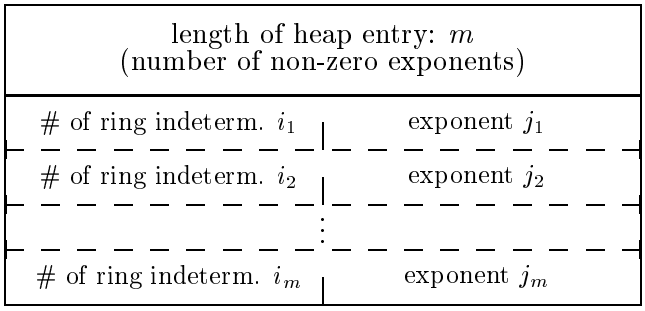
The use of number lists for the representation of commutative terms wastes a lot of memory due to the connecting pointers between the components. Operations which mainly utilize list properties such as insertion and deletion of elements are not applied in term arithmetics. So, we can apply a more compact storage model packing the sequence of components in a coherent memory region. In the case of terms the situation is even better than for polynomials (see Section 2) because the entries are always integers. So, bounding the allowed degree according to the machine word size it is useful to introduce a new array type of constant size integer entries. In FELIX the special data type for commutative terms is called *exponent vector*. It is based on a sparse representation which for dimension $n \geq 3$ proved to be preferable.

Though, all vectors $(i_1, i_2, \ldots, i_n)$ are of the same length n for a fixed polynomial ring the data type of exponent vectors includes objects of variable length since we used the sparse representation. Therefore, a specific dynamic data management is required. In FELIX this is supported by splitting the memory in different regions. More precisely, an exponent vector consists of an exponent vector cell, an object of constant size, and a heap entry of variable size where the exponents are stored. For details we refer to [AK92]. Data management operations such as allocation or releasing of exponent vectors will act only over the cell region in most situations.

An exponent vector cell is built up according to Figure 3. The corresponding heap entry for a term $x_{i_1}^{j_1} x_{i_2}^{j_2} \cdots x_{i_m}^{j_m} \in R = A[x_1, ..., x_n]$ is shown in Figure 4.

Implementing this feature we packed both parts, the index of an indeterminate and its associated exponent, into a single machine word (32 bits) which ensures good memory exploition and fast access. For practical reasons, the division of the word is asymmetrical. The index of an indeterminate is represented by only one byte which restricts the total number of ring indeterminates to 256. The remaining three bytes are dedicated to the exponent. The vector components should be open also for negative entries since the representation of terms is not the only purpose of exponent vectors. They are also used for constructing ordering matrices necessary for defining term orderings. The exponent parts are stored with respect to the two's complement which provides the range $-8388608 \ldots 8388607$ sufficient for most practical applications. Note, that the word holding the length of the heap entry contains also the number n of ring indeterminates. Since this number is only of interest if different rings are considered simultaneously we did not figure out it for the sake of simplicity.

Figure 4: Heap entry of the exponent vector $x_{i_1}^{j_1} x_{i_2}^{j_2} \cdots x_{i_m}^{j_m}$



The facts described so far deal with the aim of storing exponent vectors in a compact way. Introducing a special data type for algebraic objects suggests to equip the data type with the most important algebraic operations acting on the objects. This equipment should be done in the system kernel where the performance can be made much better than defining it in the high level programming language. Since the FELIX kernel is written in an assembler language the code of kernel functions is generally fast. Furthermore, additional features not available in high level functions can be used, e.g. a kernel function may hold intermediate results or arguments accessed more than once directly in processor registers.

The total number of functions associated with exponent vectors is sixteen, among them constructors, selectors, monoid operations, divisibility operations, and ordering operations. The most important are listed below:

- **VSET(***integerlist***)** ... Converts the list of integers to a vector.
- **VNTH(***integer*,*vector***)** ... Projects to a component of a vector.
- **VECTOR(***expression***)** ... Tests whether an arbitrary expression evaluates to a vector.
- **VPLUS(***vector#1*,*vector#2***)** ... Adds both vectors.
- **VSCALAR(***vector#1*,*vector#2***)** ... Yields the dot product of the vectors.
- **VDEGREE(***vector***)** ... Yields the total degree, i.e. the sum of the components, of the vector.
- **VMAX(***vector#1*,*vector#2***)** ... Computes the vector of maxima of the corresponding components of the input vectors.
- **VDIFF(***vector#1*,*vector#2***)** ... Computes the difference of the two vectors.
- **VEQUAL(***vector#1*,*vector#2***)** ... Tests whether two vectors are equal.
- **VORDER(***vector#1*,*vector#2*,*matrix***)** ... Tests whether the first vector is less than the second with respect to the ordering described by the matrix.
- **VLEXIC(***vector#1*,*vector#2***)** ... Especially designed for the lexicographical ordering. Similar to VORDER.

Table 2: Hit rate of already computed exponent vectors in per cent

	$n=2$	$n=5$	$n=8$	$n=15$
$(x_1+x_2^2+x_3^3)^n$	26.3	46.4	54.0	59.9
$(x_1+\ldots+x_5^5)^n$	30.4	59.3	67.4	73.3
$(x_1+\ldots+x_{10}^{10})^n$	32.5	69.7	77.5	80.1
$(1+x_1+x_2+x_3)^n$	31.2	72.3	83.0	91.1
$(1+x_1+\ldots+x_5)^n$	38.9	75.7	84.9	92.0
$(1+x_1+\ldots+x_{10})^n$	44.6	77.9	86.2	92.6

3.3 Implementation of AVL-trees

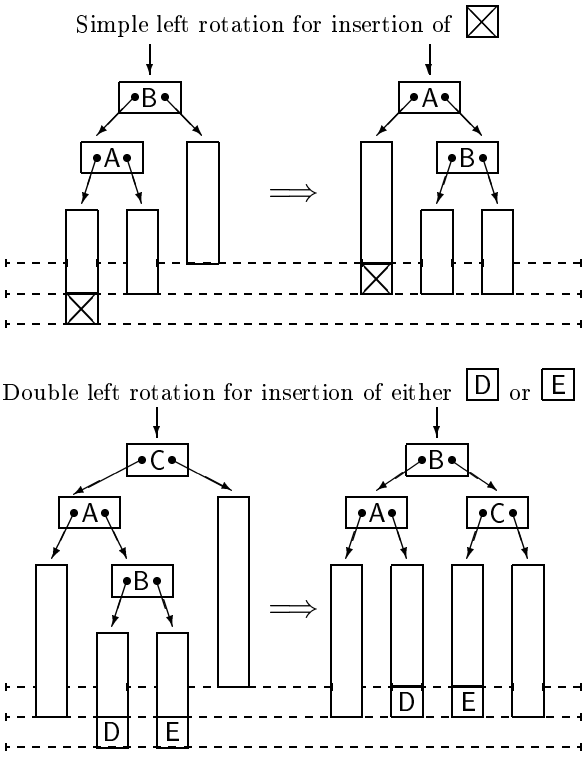
As far as discussed in the previous section there is nothing mentioned about multiple creating and storing the same exponent vector. A simple consideration shows that it is very likely that many vectors will be created several times during a session. If the product of two polynomials is computed the result will contain only terms of degree at most the sum of the degrees of them. On the other hand, there will appear l intermediate terms where l is the product of the numbers of terms of the polynomials. Assumed the polynomials contain n indeterminates and they are dense of degree d_1 and d_2, respectively. Then the polynomials consist of $\binom{d_1+n}{n}$ respectively $\binom{d_2+n}{n}$ terms. Consequently, the product contains $\binom{d_1+d_2+n}{n}$ terms. The number of intermediate terms is $\binom{d_1+n}{n}\binom{d_2+n}{n}$. In the univariate case that yields the enormous difference between d_1+d_2+1 and $(d_1+1)(d_2+1)$. For more variables the situation gets still more dramatical, e.g. if $n=10, d_1=4, d_2=6$ then the number of necessary terms is $352,716$ and this of intermediate computed ones is $1,001*8,008=8,016,008$.

In practical, the situation is a bit better than described above since dense polynomials are the worst case which is very unlikely to appear as mentioned in Section 2. In the best situation, which appears also not very often, no multiple vectors occur, as for instance in the example $(x^2+y)(x+y^2)$.

Analysing the exponent vectors computed e.g. during usual polynomial arithmetics one detects that many of them are already stored on the heap. Table 2 gives an impression on how often one meets already created exponent vectors during arithmetics. There are chosen two families of sparse and dense polynomials. The hit rate is much higher in the dense case. Furthermore, the hit rate increases with the complexity of the examples. In subsection 3.4 this will be stressed once more by the Tables 4 and 5 where the hit rates are much higher than 90%.

Since the probability that the same exponent vector occurs several times is very high it is natural to ask for a storage strategy keeping vectors unique. The most common way to implement data types where any object occurs at most at one place in the memory is the arrangement of hash tables. But how to find a suitable hash key which splits the exponent vectors of an arbitrary number of variables in balanced classes? Such natural properties as the degree would not be a good key since the resulting distribution is far from being balanced. The danger is enormous that the exponents appearing in special applications inherit some common structure which could lead to an unbalanced distribution of the occuring vectors.

Figure 5: Reordering of AVL-trees during insertion



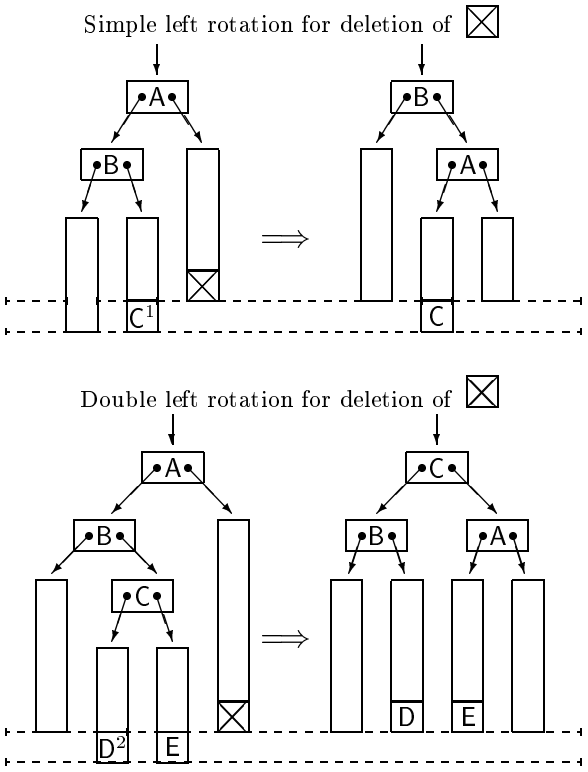
Therefore, the final decision was to supplement the memory layout with another additional structure allowing to search the heap for a certain exponent vector in a fast way. Balanced binary trees are usually a good choice for such processes. In order to support also fast insertion and deletion operations there are used *AVL-trees* [AVL62] for the FELIX exponent vector management.

AVL-trees have the property that for every node the difference of the depths of the left and the right subtree is at most one. The worst case complexity of searching, insertion, and deletion of a certain node is $O(\log N)$ (N ... number of nodes).

Implementing AVL-trees we have to supplement the exponent vector cells to build a binary tree (see Figure 3). These two additional parts can be interpreted as the left and the right subtree and contain pointers at other exponent vector cells or at the atom NIL representing the empty list. Since all cells are aligned at machine word size the least bit of a cell pointer is always zero. This allows to pack the necessary information about balance into the least bits of the pointers at subtree cells. The bits l and r in Figure 3 are set if the corresponding subtree is deeper and reset, otherwise.

By convention, nodes of left subtrees represent smaller and right subtrees represent greater exponent vectors. The ordering necessary for constructing these binary trees is quite simple. It depends only on the exponent vector's heap entry illustrated in Figure 4. First the lengths of the heap entries, i.e. the number of non-zero exponents, is compared. If both lengths are equal the elements of the heap entries will be compared

Figure 6: Reordering of AVL-trees during deletion



according to the usual lexicographical ordering. This procedure has the advantage that the number of accesses to heap entry components is minimal.

Whenever a term is computed a heap entry for the exponent vector is created on the top of the heap, but no corresponding vector cell is allocated at this moment. Next, we have to check whether there already exists an old heap entry decoding the same term. This is performed by a recursive search through the AVL-tree of vector cells starting at the root node. If the search is successful then the new heap entry is released and the found cell becomes the result of the computation. Otherwise, the search ends at NIL. In this case we have to create a new exponent vector cell pointing at our heap entry and to insert it into the AVL-tree.

Insertion is performed in the following way. First, a vector cell is allocated and its four parts (see Figure 3) are initialized (both subtree pointers are set to NIL, $l = r = 0$). Now, all parent nodes along the searching path have to be checked bottom up for correct balance information to keep the AVL-tree property. As explained in [W83] after insertion into an AVL-tree at most one operation is necessary to reorder the tree. Figure 5 sketches two of these principal reorderings, the simple left (three pointers have to be updated) and the double left (five pointers have to be updated) rotation. The corresponding simple and double right rotations are symmetric.

[1] C is optional

[2] D and E are optional but at least one has to appear

Deletion of nodes takes place only during garbage collections. It is performed in a similar way as insertion, but it can require reordering at every node along the searching path. Figure 6 shows the two left deleting rotations, the right ones are again symmetric.

Note, the following heuristic. If more than the half of vector cells have to be deleted during a garbage collection then it proved to be better not to delete the cells sequentially but to rebuild the whole AVL-tree.

3.4 Example

The comparison of the proposed methods will be illustrated by an example which has been widely investigated by many authors (see e.g. [BF91] and [D87]). The gained results are representative for most examples treated by the authors.

The task consists in computing Gröbner bases for a family of systems of algebraic equations (cyclic n-th roots).

$$\begin{aligned} x_1 + x_2 + \ldots + x_n &= 0 \\ x_1x_2 + x_2x_3 + \ldots + x_{n-1}x_n + x_nx_1 &= 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\ x_1\cdots x_{n-1} + x_2\cdots x_n + \ldots + x_nx_1\cdots x_{n-2} &= 0 \\ x_1\cdots x_n &= 1 \end{aligned}$$

The polynomials obtained by subtracting left and right hand sides of each equation generate an ideal for which the Gröbner basis shall be computed. The term ordering used in the tests is first according to the total degree and then reverse lexicographical. Note that we used a rather simple implementation of Buchberger's algorithm using standard selection strategy and applying criterions for avoiding unnecessary reductions. In general, it is not comparable with the current state of the art, but our intention was just to have a complex example making extensive use of exponent vector operations. Our tests cover the cases $n = 5, 6, 7$. Whereas we calculated over the field of rational numbers for $n = 5$ and $n = 6$ there was applied characteristic 31991 in the case $n = 7$. This prime is lucky for the example. In particular, the vector space dimension of the quotient ring is the same in both characteristics 0 and 31991.

We used three different managing strategies for exponent vectors. The first strategy (see Table 3) does not use unique data representation and the exponent vectors are linearly stored on the heap. The characteristic of the two other strategies is that the vectors are kept unique and arranged in AVL-trees. Full management (see Table 4) means that the space of currently unused exponent vectors will be recovered by garbage collections. We tested also a restricted AVL-strategy (see Table 5) which keeps all exponent vectors created so far. Such a method completely avoids costly deletion and re-creation of exponent vectors. Finally, Table 6 presents a comparison of the memory requirements of the three strategies.

The first observation is that there are no significant differences between the computing times for the three strategies. Because of many outside influences, e.g. memory equipment, it is always difficult to give a reliable interpretation of garbage collection times. So far, the linear model is the fastest since there is no overhead for the exponent vector creation process. Though, also the deletion of exponent vectors is cheeper the total amount of garbage collection time is higher for the linear model since the heap is exhausted more frequently. Due to longer average search pathes the time advantage expected for the restricted AVL-strategy is more than lost.

Table 3: Computation without AVL-trees

	$n=5$	$n=6$	$n=7$
comp. time (in sec)	5	370	34,580
+ garb. coll. time	2	37	2,092

Table 4: Computation using full AVL-tree management

	$n=5$	$n=6$	$n=7$
requested vectors	18,213	671,161	151,238,022
created vectors	1,093	16,031	1,109,253
deleted vectors	850	15,685	1,107,029
hit rate (in per cent)	94.0	97.6	99.3
max. AVL-tree depth	10	12	16
average search path length	6.8	8.1	10.7
insertion rotations	600	7,243	513,191
simple left	166	2,081	146,423
double left	138	1,909	129,668
simple right	127	1,707	123,394
double right	169	1,546	113,706
deletion rotations	210	3,093	252,113
simple left	71	1,135	90,460
double left	36	396	32,704
simple right	65	1,152	92,008
double right	38	410	36,941
comp. time (in sec)	6	376	36,139
+ garb. coll. time	2	32	1,400

Concerning the space requirement the full AVL-tree management is the outstanding strategy. The amount of heap memory occupied by exponent vectors was measured at two distinguished points. The first moment was when the intermediate ideal bases reached their maximal sizes (see first part of Table 6). This point was chosen as one of large, not necessarily maximal, memory demand. Second, the situations after finishing the calculations were investigated (see second part of Table 6).

In summary, we state that the full AVL-tree management is the most preferable strategy since its memory demand is significantly the smallest while the computing times are almost the same for all three variants.

4 Representation of non–commutative terms

The set of words over the alphabet $\{x_1, ..., x_n\}$ forms an A-module basis of the free non-commutative algebra $A\langle x_1, ..., x_n\rangle$. The basic algebraic operations are based on word operations. First of all, the words form a monoid with respect to the concatenation. Furthermore, the words have to be ordered with respect to an ordering compatible to

Table 5: Computation using restricted AVL-tree management

	$n=5$	$n=6$	$n=7$
requested vectors	18,213	671,161	151,238,022
created vectors	621	3,357	62,444
hit rate (in per cent)	96.6	99.5	99.9
AVL-tree depth	12	15	20
average search path length	8.1	10.0	13.8
comp. time (in sec)	5	374	36,095
+ garb. coll. time	1	33	1,528

Table 6: Comparison of heap memory requirements of vectors (in byte)

	$n=5$	$n=6$	$n=7$
max. length of intermediate basis	23	65	402
no AVL-trees	4,488	35,724	1,235,992
full AVL-tree management	1,772	5,628	37,336
restricted AVL-tree management	8,512	58,460	1,247,608
final length of basis	21	44	209
no AVL-trees	3,312	14,344	421,012
full AVL-tree management	1,268	2,952	18,376
restricted AVL-tree management	9,548	61,952	1,755,456

concatenation. Finally, there are required matching operations for detecting subword and overlapping properties. In the following subsections there will be given some possibilities how to build up data types representing words and supporting the necessary operations.

Our test series dealing with non-commutative rings are still very small. In comparison to polynomial rings the examples split even more into two classes, namely, trivial and almost unsolvable applications. A special handicap is that the termination of the Buchberger algorithm is not ensured [M86].

4.1 Lists of indeterminates

The most obvious way to represent a word is to consider the list of its characters. Similar to the commutative case character sequences may be stored more efficiently using dynamic array structures. Of course, instead of the sequence of indeterminates we can use simply the sequence of corresponding indices.

4.2 Hardware supported data type

For each natural number $B > n$ there is a surjective mapping from the set of non-commutative terms to the natural numbers by assigning the value $\sum_{j=0}^{m} i_j B^j$ to the term $x_{i_0} x_{i_1} \cdots x_{i_m}$. Let us fix B. In [AK91a] it is shown how the arithmetic, matching, and ordering operations between non-commutative terms can be transformed to inte-

ger operations between their above described code numbers and another two integers belonging to each term. The additional code numbers allow more efficient implementations for some operations but their information is redundant. In particular, they ensure that all important operations can be performed almost without costly decoding and encoding of terms. But there is a snag in this method. The integers encoding the terms can be, and actually will be, rather large. It is not advisable to use the long integers included in FELIX for the term representation since the integer operations applied to the codes are not simple, e.g. they include the computation of remainders of integer division.

A compromise between the restriction to machine size integers, which allow to represent only a very small range of terms, and long integers, which are not supported by direct processor instructions, is the use of 8-byte integers and to perform the arithmetics using the floating point unit or an arithmetic coprocessor.

4.3 Bitstrings

But also the restriction to 8-byte integers turned out to be rather strong. Limitations going along with this encoding were presented in [AK91a]. Finally, we created a new data type which is simpler but more general than the coprocessor method.

Non-commutative terms will be again represented by sequences of indices of indeterminates. In most cases the number n of indeterminates is much smaller than the largest integer representable by a machine word and the code of an indeterminate requires only some bits. Therefore, we implemented the data type of *bitstrings*. A bitstring employs a coherent memory region containing the total number of ring indeterminates, the degree of the term, and the sequence of indices in a packed form.

There are fifteen kernel functions operating over bitstrings, e.g. constructors, selectors, arithmetic functions, ordering tests, and functions for converting between exponent vectors and bitstrings. A special subgroup of the arithmetic functions is dedicated to substring and overlapping problems.

Similar to the case of exponent vectors it arises the question of unique representations. Within multiplication the portion of multiple created terms will be smaller than in the case of commutative polynomials. Nevertheless, the advantages are still large enough to justify a unique representation strategy. For this purpose the bitstrings are again arranged in an AVL-tree. The ordering used in the tree is first according to the total number of ring indeterminates, then to the degree, and last lexicographical with respect to the sequence of indices.

When we developed the data type of bitstrings we expected that it would not be as fast as the hardware supported floating point method since more operations have to be performed digit by digit. But we were pleasantly surprised that the bitstring method turned out to be the faster one since it does not require so much processor communication and does not apply the expensive floating point arithmetics for simple integer calculations.

References

[AVL62] G.M. Adelson-Velskij, E.M. Landis, Odin algoritm organisazii informazii. Doklady Akademii Nauk SSSR, 146 (1962), pp. 263-266 (in Russian).

[AK91a] J. Apel, U. Klaus, Implementation aspects for non-commutative domains. Proc. Computer Algebra in Physical Research, ed. D.V.Shirkov, V.A.Rostovtsev, V.P.Gerdt, World Scientific, pp. 127-132, 1991.

[AK91b] J. Apel, U. Klaus, FELIX – an assistant for algebraists. Proc. ISSAC'91, ed. S.M. Watt, ACM Press, pp. 382-389, 1991.

[AK92] J. Apel, U. Klaus, Data Representation and In-built Compilation in the Computer Algebra Program FELIX. L.N.C.S. **721**, pp. 173-192, 1993.

[BF91] J. Backelin, R. Froeberg, How we proved that there are exactly 924 cyclic 7-roots. Proc. ISSAC'91, ed. S.M. Watt, ACM Press, pp. 103-111, 1991.

[D87] J.H. Davenport, Looking at a set of equations. Bath Computer Science Technical Report 87-06, 1987.

[M86] T. Mora, Gröbner bases for non-commutative polynomial rings. L.N.C.S. **229**, pp. 353-362, 1986.

[W83] N. Wirth, Algorithmen und Datenstrukturen, B.G. Teubner Stuttgart, 1983.