Representing Polynomials in Computer Algebra Systems

Joachim Apel[*] Uwe Klaus[**]

Universität Leipzig
Fakultät für Mathematik und Informatik
Augustusplatz 10–11
D–04109 Leipzig, Germany

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 short introduction to the system given in section 2 should motivate our efforts towards polynomials. Section 3 illustrates the data management of the system.

In this paper we want to report about different investigated possibilities of data representation. Primary, we will not aim our attention at the question of complexity formulas since we did not develop new algorithms for performing the arithmetic operations. What we did is to design special data types for the internal representation of polynomials and monomials. 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. The result of such an approach is mainly to diminish the constant factors rather than the complexity. Storage complexity improvements could be achieved using unique data representations. But unique data representation has to be paid with management overhead which, on the other hand, may influence the time complexity by changing the costs of the basic operations.

In section 4 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 5 and 6 deal with questions concerning data types designed for exponent vectors of commutative monomials and for words representing non-commutative monomials. Some different data types will be compared by analysing the computation of some rather complex examples.

2 Algebraic scope of FELIX

FELIX 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.

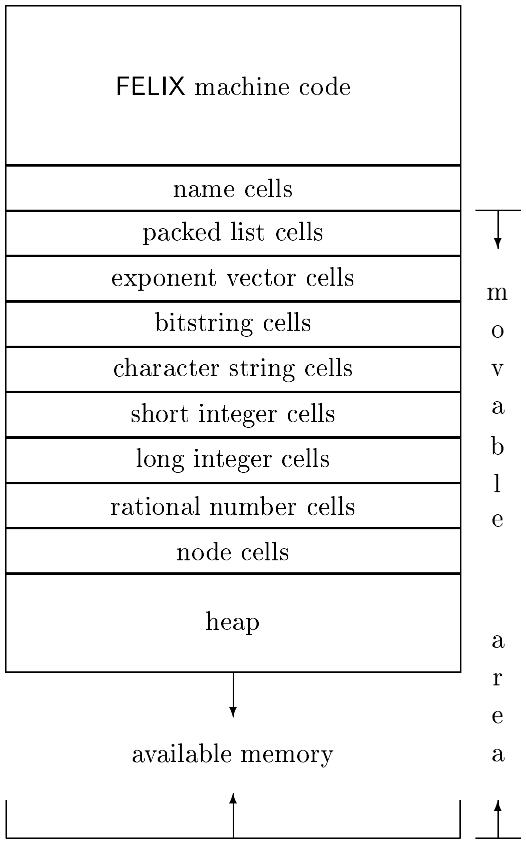
The system not only manages the calculation with elements of the above algebraic structures but also with the structures themselves and mappings between them as well. [AK91b] gives a more detailed overview about the algebraic capabilities.

For simplicity we will call both, elements of polynomial rings and of non-commutative algebras, polynomials. Crucial for all applications is an effective implementation of the polynomial arithmetics. Besides the usual ring arithmetics there are also to consider operations related to term orderings since Buchberger's algorithm plays a central rule in the system.

This is the reason why we put many efforts in investigating different data structures for monomials and polynomials. Our experiments reach from using hardware oriented data structures for monomials [AK91a]

[*]japel@informatik.uni-leipzig.de
[**]uklaus@informatik.uni-leipzig.de

Figure 1: FELIX memory map



up to organizing binary trees for the handling of terms. Furthermore, we gathered the terms of a polynomial in list form as well as in an array structure.

The data types differ in the space requirements for representing polynomials. The computing times for the polynomial arithmetics depend on the data structures, too. The choice of a data structure from the point of view of time behaviour can always be only a compromise since the 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. .

It proved to be preferable to introduce special atomic data types for the implementation of monomials. Atomic means that the data types including the basic operations acting on them are completely part of the system kernel.

3 Basic data types and memory management

As already mentioned during the development of FELIX a main goal was the implementation of special data types which reflect basic properties of the represented algebraic objects.

Nevertheless, the structure of the FELIX data is LISP–like. Any data is either an atom or a sequence of other data (lists). But the classes of atoms implemented so far are more extensive:

- *names*,
- *integers*,
- *rational numbers*,
- *character strings*,
- *exponent vectors*,
- *bitstrings*, and
- *packed lists*.

The conclusion is that in contrast to many LISP–implementations the storage model of FELIX is inhomogeneous, i.e. different data types are stored in different storage areas (see figure 1).

The interpretation of *names* is context dependent. The same name can be used as a notation of a global variable as well as of an operator.

The implementation of *integers* distinguishes between short and long numbers. Shorts are within the range of a machine word and stored directly in their cells.

The data type of *exponent vectors* was created to support a commutative polynomial arithmetic. It is based on a sparse representation of monomial exponent vectors. Within FELIX there are included sixteen machine routines which implementing a polynomial arithmetic efficiently perform most of the required operations (see section 5).

The *bitstrings* correspond to the non-commutative case. A non-commutative monomial is stored by a sequence of integers which represent the ring indeterminates. These integers are coded with some bits only since the number of ring indeterminates is usually small (see section 6).

Long integers, *character strings*, *exponent vectors*, *bitstrings*, and *packed lists*, which correspond to data of variable size, are represented by two parts: a cell where they are registered (see figure 2), and a heap entry where their elements are stored (see figures 3).

Sequences (lists) are built either as binary trees by node cells (see figure 4) in the usual LISP–like way (see figure 5) or by arrays, the so called packed lists (see figure 6).

There are two different kinds of garbage collection. The first one is caused if no cells are available. It is performed in a usual way. First, there are marked all the current occupied data beginning with the initial data, the name cells, the temporary computed elements on runtime stacks, the constants of the linked modules, etc. and then recycled all the unused cells.

Figure 2: Components of a cell which represents a variable size object

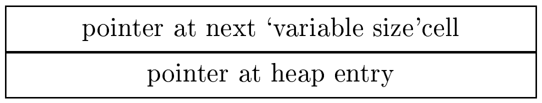


Figure 3: Components of a heap entry

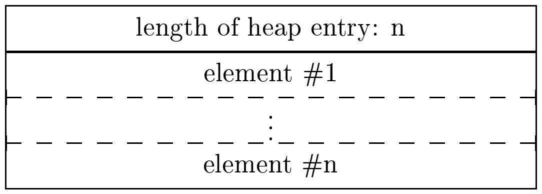
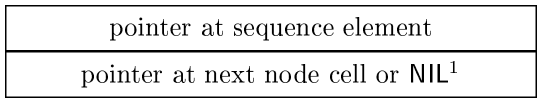


Figure 4: Components of a node cell


pointer at sequence element
pointer at next node cell or NIL[1]

The whole available memory (see figure 1) may be used to extend the heap. If a new heap entry is requested then it will be stored immediately behind the heap and the pointer at the top of the available memory will be updated. It may happen that the remaining available memory is too small to create a new entry, although, lots of memory are unused. For this reason, there is a second kind of garbage collection to compress the heap. Since the chain formed by the first pointers of all 'variable size' cells reflects the time of creation of the heap entries, passing through this chain causes working off the heap linearly from the top to the bottom. This enables compression without managing gaps.

A combined compression and movement of the heap towards the memory end provides the necessary space to extent the number of cells of any sort. So, cells can be created at that time when they are requested.

The performed experiments related with unique data representation have shown that in the case of integers a unique representation of shorts only is a good compromise. It guarantees a good memory exploitation and does not require too much additional computing time to manage corresponding hash tables.

[1]Marks end of sequence

Figure 5: Representation of the sequence (< element #1 > , . . . , < element #n >) as a binary tree using n node cells

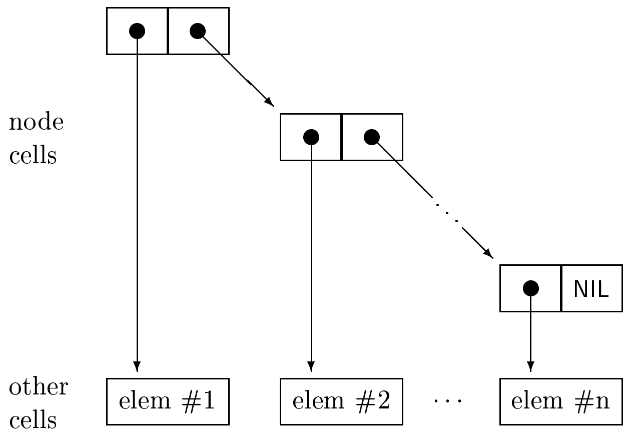
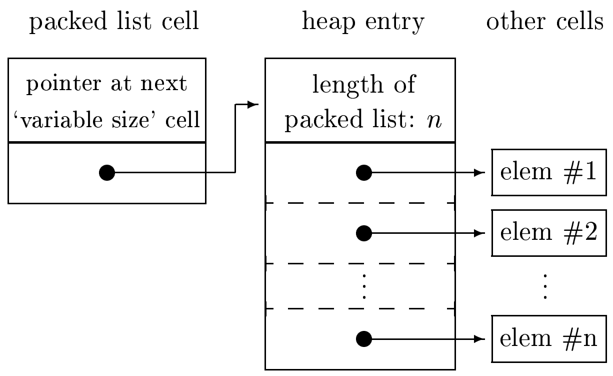


Figure 6: Representation of the sequence (< element #1 > , . . . , < element #n >) as a packed list



4 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

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

		packed list representation			list representation		
n	# of terms	revlex	lexic	totdeg	revlex	lexic	totdeg
10	3003	6.9	5.5	5.2	88.6	92.2	71.4
11	4368	10.2	7.9	8.1	195.5	202.3	137.5
12	6188	16.4	13.3	12.3	432.4	435.0	280.5
13	8568	21.9	19.3	16.8	846.2	793.4	548.6
14	11628	40.2	27.9	27.8	1697.5	1558.1	990.3

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.

To get really normal forms the module basis of R has to be ordered. In the case of factorized representations additional conventions about the use of units are necessary.

Both module basis normal forms have the advantage that all arithmetic operations are rather easy to perform. In opposite, using factorized normal forms causes serious difficulties as soon as summation is envolved. This is probably the main reason that such a form of representation is not used in the computer algebra systems known to the authors, although, it often could save a lot of memory.

In the following, we will neglect the factorized representation since our investigations are also restricted to the module basis normal forms.

The main advantage of the recursive representation is the possibility of introducing new variables during the session without reorganizing previously computed expressions. If a new variable is considered to be the last adjuncted one the previous expressions are simply coefficients of the module basis element 1. Therefore, the recursive representation of polynomials is very often used in not algebraic structure oriented computer algebra systems, i.e. in systems which do not require the explicit definition of a working structure.

The distributed form of a polynomial is preferable for many arithmetic operations. Some algorithms, particularly, 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 required property. Otherwise, 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 broken off at the largest power product with non-zero coefficient. This method has the advantage that no space for storing the basis elements is required. Furthermore, there are fast implementations for the arithmetic operations. Otherwise, also zero coefficients have to be stored which makes only sense for dense polynomials.

In multivariate applications dense polynomials are not of great interest as the number of basis elements with a certain degree limit shows. There are $\binom{d+n}{d}$ power products 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 power products with degree up to 10. In the case of non-commutative polynomials the situation is still worse. There are n^d power products in n indeterminates of exactly degree d.

Therefore, computer algebra systems work with sparse representation. In this case a polynomial is represented by the sequence of its non-zero monomials given by the coefficients and the associated power products. The question of the internal representation of the power products will be discussed in the next two sections. The internal structure of the coefficients will not be considered 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 type of the object there remain two principle forms of storing a polynomial . First it can be a list (see figure 5) 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 6). 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. Actually, sorting a list it is better first to convert it into an array, sort it, and then to convert it back.

Table 1 gives a comparison of times necessary to reorder 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 orderings.

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 has to be copied into a new array. In contrary this operation can be executed in constant time for list structures. This consideration shows a second advantage of the list representation. Two polynomials which have the smallest terms in common may share this terms what may save memory. Such polynomials occur not only if a tail is computed but often also after adding two polynomials.

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 as subtraction, multiplication, and in a certain sense also division can be reduced to these two elementary operations. Neglecting the monomial operations addition is performed in a zipper like way and multiplication by a monomial requires only going ones through the polynomial. The aptitude of list and array representation is almost equal. The list representation has some small advantages with respect to the addition. If one of the polynomials is completely worked off it is sufficient to append the whole list of remaining monomials of the other polynomial in one single step at the result. In contrary, the same situation requires copying references to all remaining monomials into the array of the resulting polynomial using the array representation. Furthermore, the length of the result is unknown at the beginning of the computation but an array can not be enlarged subsequently. The length of the sum is limited by the sum of the lengths of the input polynomials. This length bound of the result seems us good enough to create an array of this limit length where the result will be put in. The system FELIX includes an array operation which allows to cut off the result to the right length at the end of the computation.

For elements of free non-commutative algebras $A\langle x_1, ..., x_n\rangle$ the same remarks as to the distributed normal form of polynomials apply. Recursive or even factorized forms are not available for algebraic reasons.

5 Representation of commutative monomials

In this section monomial will denote a pure power product, i.e. it contains no coefficient.

5.1 Number lists

Using the distributed polynomial representation the sequence of indeterminates should be stored in one place and not appear in each monomial. The simplest way would be to assign the list of indeterminates to a global variable. FELIX as an algebraic domain oriented system holds such lists in the data describing the ring. Any polynomial contains a reference to the domain it belongs to.

Forgetting the names of the indeterminates a monomial is a (finite) sequence of natural numbers, the exponents of the several indeterminates. In any computer algebra system it is possible to arrange these numbers in lists. Similar as in the case of polynomials there can be distinguished sparse and dense representations for the exponent lists. Which of these both forms is preferable depends on the number of indeterminates involved. 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 the computing time as well.

5.2 Exponent vectors

The use of number lists for the representation of commutative monomials wasts a lot of memory since there are necessary connecting pointers between the components. Operations which mainly utilize list properties such as insertion and deletion of elements are not necessary in monomial arithmetics. When a new exponent vector is created its length is already known at the very beginning. Furthermore, there is no distinguished component of such a vector. Therefore, any component should be accessable by the same effort.

An alternative form which avoids the additional memory requirement of lists and provides access to arbitrary components in constant time is the principal data type of arrays where the elements are stored as a sequence in a coherent memory region. Such a data model was already described in section 4. In the case of monomials the situation is still better than for polynomials because the entries are always integers. Since array entries have to be of constant size which can not be assumed for arbitrary objects the general type of packed lists which is used for instance for representing polynomials consists of a pointer sequence. Restricting the range of the exponents to a certain maximal integer all array components contain elements of the same size. This forces the introduction of a new array type consisting directly of integer entries.

In FELIX such a special data type representing commutative monomials is formed by the exponent vectors. Although, during computation in one specific algebraic structure selected by the user the length of an exponent vector is fixed we do not use arrays of that constant length. The reason is analogous to the case of sparse and dense list representation. The larger the number of indeterminates the ofter zero has to be stored. Of course, this alters some of the remarks made before, e.g. that concerning the access to arbitrary components, but the advantages dominate.

The decision to use arrays (e.g. exponent vectors) of variable length forces a specific dynamic data management. As already described in section 3 it is done by the indroduction of two different memory regions, the region of exponent vector cells and the heap (see figure 1). Remember that in FELIX every algebraic object is represented by a single cell. So, an exponent vector contained in a more complex object is presented by a pointer at its corresponding vector cell.

A vector cell is built up according to figure 2. The corresponding heap entry for a monomial $x_{i_1}^{j_1} x_{i_2}^{j_2} \cdots x_{i_m}^{j_m}$ from the polynomial ring $R = A[x_1, ..., x_n]$ is shown in figure 7.

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 monomials is not the only purpose of exponent vectors. They are also used for constructing ordering matrices necessary for defining term orderings. In this case negative numbers are required as soon as non-noetherian term orderings are considered. The exponent parts are stored with respect to the two's complement which provides the range $-8388609 \ldots 8388608$.

Figure 7: Heap entry of the exponent vector $x_{i_1}^{j_1} x_{i_2}^{j_2} \cdots x_{i_m}^{j_m}$

length of heap entry: m (number of non-zero exponents)	
# of ring indeterm. i_1	exponent j_1
# of ring indeterm. i_2	exponent j_2
⋮	
# of ring indeterm. i_m	exponent 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 some 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 can be used which for non-kernel functions can not be assumed, e.g. a kernel function may hold intermediate results or arguments accessed more than once directly in processor registers.

The list of operations contains constructors, selectors, monoid operations, divisibility operations, and ordering operations. It follows the list of the most important functions associated to the data type of exponent vectors. Some functions concerning computations in non-commutative polynomial rings will be left out.

- **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.

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	—
$(1+x_1+x_2+x_3)^n$	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	—

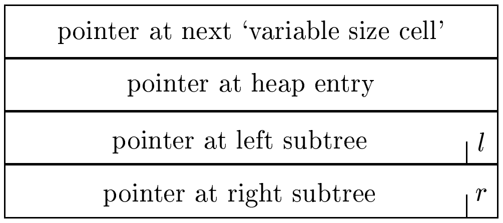
- **VEQUAL(***vector#1*,*vector#2***)** ... Tests whether two vectors are equal.
- **VORDER(***vector#1*,*vector#2*, *matrix***)** ... Tests whether the first vector is less than the second with respect to the ordering described by the matrix.
- **VLEXIC(***vector#1*,*vector#2***)** ... Especially designed for the lexicographical ordering. Similar to VORDER.

5.3 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 monomials of degree at most the sum of the degrees of them. Otherwise, there will appear l intermediate monomials 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 $\binom{d_1+d_2+n}{n}$ monomials. The number of intermediate monomials 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 monomials is $352,716$ and this of intermediate computed ones is $1,001 * 8,008 = 8,016,008$. Note that these numbers depend only on the polynomials and not on the use of dense or sparse representation.

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 4. In the best situation, which appears also not very often, no multiple vectors occur, as for instance in the example $(x^2+y)(x+y^2)$.

Figure 8: Exponent vector cell with AVL-tree pointers



Looking at the exponent vectors computed e.g. during usual polynomial arithmetics one can detect that many of these exponent vectors are already stored within the heap. Table 2 gives an impression on how often one meets already created exponent vectors during arithmetics. There are choosen 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 5.4 this will be stressed once more by the tables 4 and 5 where the hit rates are much higher than 90%.

Since the chances that the same exponent vector occurs several times are very good it is natural to ask for a storage strategy which keeps the vectors unique. The most common way to implement data types where any object is at most once in the memory is to arrange them in 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 large that the exponents appearing in special applications inherit some common structure which could lead to an unbalanced distribution of the occuring vectors.

Therefore, the final decision was to supplement the basic data with another additional structure allowing to search the heap for a certain exponent vector. Binary trees are usually a good choice for such processes. Because insertion and deletion (caused by garbage collections) play an important role special balanced binary trees, the so called AVL-trees [AVL62] are applied to the FELIX exponent vector management.

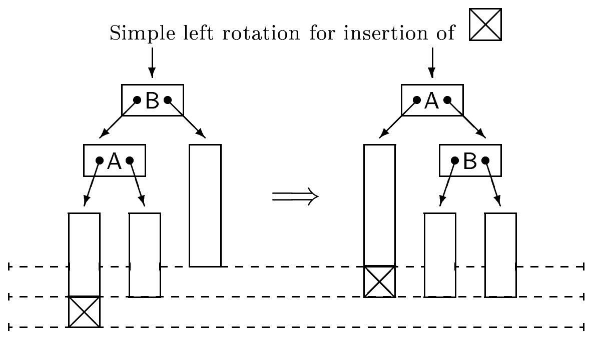
AVL-trees have the property that for every node the difference of the depths of the left and the right subtree is at most one. The complexity of searching, insertion, and deletion of a certain node is $O(\log n)$ (n ... number of nodes) even in the worst case.

Implementing AVL-trees we have to supplement the

[2] C is optional

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

Figure 9: Reordering of AVL-trees during insertion



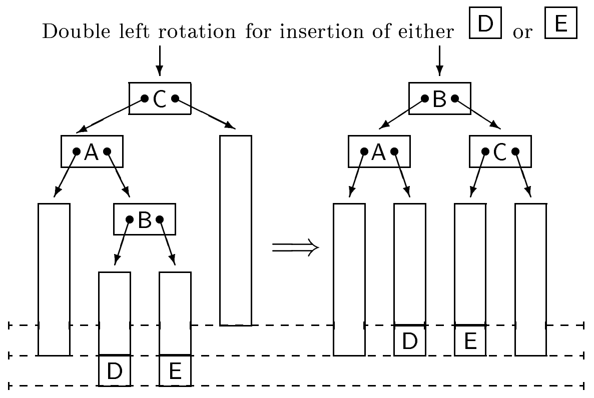
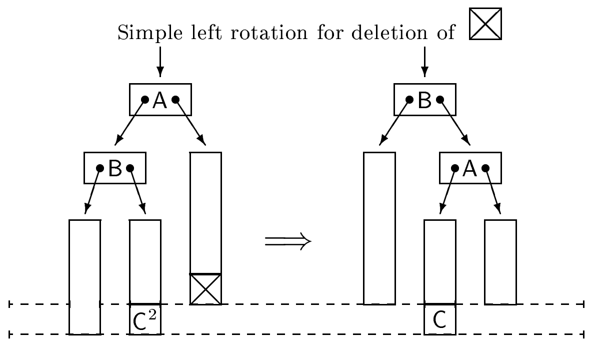
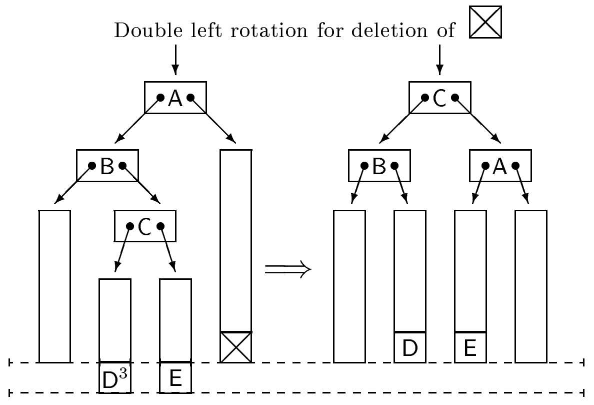


Figure 10: Reordering of AVL-trees during deletion





exponent vector cells to build a binary tree (see figure 8). These two additional parts contain pointers at other exponent vector cells or in the case of leaves NIL and can be interpreted as left and right subtree. Since all cells are alined at machine word size the least bit of all pointers is zero. The necessary information about balance is packed into the least bits of the pointers at subtrees. These two bits (l, r in figure 8) of an exponent vector cell contain zeros if the both subtrees have the same depth or one if the corresponding subtree is deeper.

By convention, nodes of left subtrees always represent smaller and right subtrees greater exponent vectors. The ordering necessary for constructing these binary trees is quite simple. It depends only on the exponent vector's heap entry as shown in figure 7. 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 according to the usual lexicographical ordering. This procedure has the advantage that the number of accesses to heap entry components is minimal.

Whenever a monomial is computed a heap entry for the exponent vector, which we will denote by $(*)$, is created on the top of the heap without allocation of a corresponding vector cell. Beginning with the root node the AVL-tree has to be searched recursively for $(*)$. As above described the node's heap entry is compared with the just computed one. In the case of equality, the node's vector cell is the desired result for the computed monomial, and the heap memory of $(*)$ can be released immediately. Otherwise, according to the result of the comparison the search has to be continued with the root of the left or the right subtree. If branching is impossible because the pointer at this subtree is NIL the entry $(*)$ has to be inserted into the AVL-tree.

Insertion is performed in the following way. First, a vector cell is allocated and its four parts (see figure 8) are initialized (both subtree pointers are set to NIL, $l = r = 0$). Now, all parent nodes along the searching path have to be checked 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 9 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.

Deletion of nodes is part of the garbage collection. It is performed analogously to the insertion. First, the corresponding node is searched within the AVL-tree and

Table 3: Computation without AVL-trees

	$n=5$	$n=6$	$n=7$
comp. time (in sec)	24	1,780	165,959
+ garb. coll. time	10	180	10,039

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
max. AVL-tree depth	10	12	16
average search 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)	27	1,804	173,437
+ garb. coll. time	8	153	6,718

removed by changing the subtree pointer of the parent's node to NIL. Now, all parent nodes along the searching path have to be checked for correct balance information. In contrast to insertion reordering can be necessary in every node of the searching path. Figure 10 shows the two left deleting rotations, the right ones are again symmetric.

Note the particularity of deletion. In contrast to insertion during garbage collection many vector cells have to be removed all at once. If more than the half is to be deleted it is better to rebuild the whole AVL-tree than a successive deletion.

5.4 Example

The comparison of the proposed methods will be illustrated by an example which has been widely investigated also by other 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.

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
AVL-tree depth	12	15	20
average search length	8.1	10.0	13.8
comp. time (in sec)	24	1,793	173,227
+ garb. coll. time	6	157	7,334

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

	$n=5$	$n=6$	$n=7$
max. length of int. basis	23	65	402
no AVL-trees	4,488	35,724	1,235,992
full AVL-tree man.	1,772	5,628	37,336
restricted AVL-tree man.	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 man.	1,268	2,952	18,376
restricted AVL-tree man.	9,548	61,952	1,755,456

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

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. Our considerations 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 stored at the heap linearly. Inside the both other strategies the vectors are stored in a unique way and arranged in AVL-trees. The difference is that in the full management (see table 4) the space of unused exponent vectors will be recovered by garbage collections. Although, the complexities of insertion and deletion of exponent vectors are equal it seems that cancellation is more costly. Therefore, we checked also a restricted strategy (see table 5) which does not include deletion of once created vectors. Finally, table 6 presents a comparison of the memory requirements of the three strategies.

The experiments show that the linear model is the fastest. This is caused by omitting the search for already existing vectors but it has to be paid partially by garbage collection time since the exponent vectors occupy much more heap memory than using the full AVL-tree strategy as table 6 shows. Note, that the heap memory occupied by exponent vectors is still larger in the case of restrictive tree management. The influence to the garbage collection is smaller since these heap entries at most move towards the top of the heap. The near they are to the top the less likely gets another transport.

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 choosen as one of large, not necessarily maximal, memory demand. Second, the situations after finishing the calculations were investigated (see second part of table 6).

To complete the comparisons it remains the time analysis of full and restricted AVL-tree management. Besides all deletion operations the restricted method saves also many insertions. Nevertheless, the difference of the computing times is not significant. The time for deletion and insertion economized using the restricted AVL-tree strategy has to be paid back for longer average searching.

In summary, 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.

6 Representation of non–commutative monomials

The words over the indeterminates form an A-module basis of the non-commutative algebra $A\langle x_1, ..., x_n\rangle$. The algebra arithmetic is based on some 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 concatenation. Finally, there are required matching operations for detecting subword and overlaping properties. In the following subsections there will be given some possibilities to build up data types representing words which also support the necessary operations.

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

6.1 Lists of indeterminates

The most common representation of a word is the sequence of its letters. This way may be used in any computer algebra system. If the system contains dynamic array structures the sequence may be stored again more memory efficient. Of course, the indeterminates may be enumerated and replaced by associated numbers in the sequence.

6.2 Hardware supported data type

Considering the non-commutative monomials as sequences of natural numbers lower than a certain limit they appear as representations of integers in a position system to a sufficient large basis. So, any non-commutative monomial will be assigned an integer in a very natural way. This mapping should be surjective. Therefore, the enumeration of the indeterminates should start with 1 to avoid leading zeros. In [AK91a] it is shown how the arithmetic, matching, and ordering operations between non-commutative monomials can be transformed to integer operations between their above described code numbers and another two integers characterizing the monomials. These two additional numbers reflect the monomials forgetting the non-commutativity. They are carrying only some help information which makes some ordering and matching tests faster. In conclusion, it may be stated that all calculations between non-commutative monomials may be done using coding triples of natural numbers without decoding them.

But there is a snag in this method. The integers coding the monomials can be, and actually will be, rather large. It is not advisable to use the long integers included in FELIX for the monomial representation since the integer operations applied to the coding numbers are not very 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 monomials, and long integers, which are not supported by direct processor instructions, is the use of 8-byte integers and to perform the arithmetic using the coprocessor Intel 80x87.

6.3 Bitstrings

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

The non-commutative monomials will be again represented by sequences of indicees of indeterminates. Using an array for storing the sequence might waste memory since depending on the number of ring indeterminates some bits will be sufficient for any component of the array. Therefore, we created the data type of bitstrings. A bitstring employs a coherent memory region containing the total number of ring indeterminates, the degree of the monomial, and the sequence of indicees.

There are fifteen kernel functions operating over bitstrings. The list of these functions should not be given but roughly it includes construtors, selectors, arithmetic functions, ordering tests, and functions for converting between exponent vectors and bitstrings. Among the arithmetic functions there are besides the bitstring concatenation also such concerned with substring and overlapping problems.

Similar to the case of exponent vectors it arises the question of unique representation. Within multiplication the portion of multiple created monomials 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 digits.

When we developed the data type of bitstrings we supposed that it would be not as fast as the coprocessor method since more operations have to be performed digit by digit. But we were pleasently surprised that the bitstring calculations are very fast even faster than the coprocessor arithmetics. This is due to the facts that the communication between main- and coprocessor is very costly and the coprocessor method uses the more expensive floating point arithmetic 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 assistent 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. to appear in Proc. DISCO 92, Bath, 1992.

[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 (in German).