SINGULAR in a Framework for Polynomial Computations

Hans Schönemann

University of Kaiserslautern, Centre for Computer Algebra,
67653 Kaiserslautern, Germany

Abstract. There are obvious advantages in having fast, specialized tools in a framework for polynomial computations and flexible communication links between them. Techniques to achieve the first goal are presented with SINGULAR as test bed: several implementation techniques used in SINGULAR are discussed. Approaches to fast and flexible communication links are proposed in the second part.

1 Introduction

SINGULAR is a specialized computer algebra system for polynomial computations. The kernel implements a variety of Gröbner base–type algorithms (generalized Buchberger's algorithm, standard basis in local rings, in rings with mixed order, syzygy computations, ...), algorithms to compute free resolutions of ideals, combinatorial algorithms for computations of invariants from standard bases (vector space dimensions, –bases, Hilbert function, ...) and algorithms for numerical solving of polynomial systems.

All others task have to use external tools, which include C/C++–libraries and external programs.

In order to be able to compute non–trivial examples we need an efficient implementation of the Gröbner base–type algorithms as well as efficient communication links between independent packages. The method of Gröbner bases (GB) is undoubtedly one of the most important and prominent success stories of the field of computer algebra. The heart of the GB method are computations of Gröbner or standard bases with Gröbner base–type algorithms. Unfortunately, these algorithms have a worst case exponential time and space complexity and tend to tremendously long running times and consumption of huge amounts of memory

While algorithmic improvements led to more successful algorithms for many classes of problems, an efficient implementation is necessary for all classes of problems: analyzing GB computations from an implementation point of view leads to many interesting questions. For example: how should polynomials and monomial be represented and their operations implemented? What is the best way to implement coefficients? How should the memory management be realized?

2 Some Historical Remarks

Gröbner base–type algorithms are an important tool in many areas of computational mathematics. It is quite difficult to find out historical details in the area of computer algebra, so I apologize for missing important systems. Also it is often difficult to give a date for such an evolving target like a software system — I took the dates of corresponding articles.

Buchberger [7], and independently Hironaka [10], presented the algorithm in the 60's. First implementations arose in the 80's: Buchberger (1985) in MuLISP, Möller/Mora in REDUCE (1986) [11]. The underlying systems provided all the low level stuff like memory management, polynomial representations, coefficient arithmetics and so on. Algorithmic improvements at that time were the elimination of useless pairs, etc. First direct implementations (CoCoA, 1988; Macaulay, 1987 [5]; Singular, 1989, [13]) were restricted to coefficients in $\mathbb{Z}/p$ and did not care much about actual data representations.

In 1994, the system Macaulay 3.0 (maybe also earlier versions) used to enumerate all possible monomials by an integer, according to the monomial ordering. Monomial comparison was very fast, but the maximal possible degree was limited. The leading terms had additionally the exponent vector available for fast divisibility tests (for a detailed discussion see [2]).

PoSSo, a scientific research and development project 1993 – 1995, introduced a monomial representation which contained both, the exponent vector and the exponent vector multiplied by the order matrix (see below). That provided fast monomial operations, but used a "lot" of memory for each monomial.

Faugère's algorithm F_4 (1999) reused the idea of enumerating the monomials — instead of enumerate all he gave numbers only to the occurring ones (and for each degree independently).

SINGULAR 1.4 used C data types `char, short, int, long` to get an smaller monomial representation and to allow vectorized monomial operations (see below).

The article [15] gives interviews with developers of Gröbner engines and contains interesting stories on early developments of Gröbner engines - unfortunately it's written in Japanese.

3 Basic Polynomial Operations and Representations

As a first question, one might wonder which kind of polynomial (and, consequently, monomial) representation is best suited for GB computations. Although there are several alternatives to choose from (e.g., distributive/recursive, sparse/dense), it is generally agreed that efficiency considerations lead to only one real choice: a dense distributive representation. That is, a polynomial is stored as a sequence of terms where each term consists of a coefficient and an array of exponents which encodes the respective monomial.

With such a representation, one has not only very efficient access to the leading monomial of a polynomial (which is the most frequently needed part of a polynomial during GB computations), but also to the single (exponent) values of a monomial.

Now, what type should the exponent values have? Efficiency considerations lead again to only one realistic choice, namely to fixed–length integers whose size is smaller or equal to the size of a machine word. While assuring the most generality, operations on and representations of arbitrary–length exponent values usually incur an intolerable slow–down of GB computations. Of course, a fixed–length exponent size restricts the range of representable exponent values. However, exponent bound restrictions are usually not very critical for GB computations: on the one hand, the (worst case) complexity of GB computations grows exponentially with the degree of the input polynomials, i.e., large exponents usually make a problem practically uncomputable. On the other hand, checks on bounds of the exponent values can be realized by degree bound checks in the outer loops of the GB computation (e.g., during the S–pair selection) which makes exponent value checks in subsequent monomial operations unnecessary.

3.1 Basic Monomial Operations and Representations

Given an integer $n > 0$ we define the set of exponent vectors M_n by $\{\alpha = (\alpha_1, \ldots, \alpha_n) | \alpha \in \mathbb{N}^n\}$. Notice that monomials usually denote terms of the form $c\, x_1^{\alpha_1} \ldots x_n^{\alpha_n}$. However, in this section we do only consider the exponent vector of a monomial and shall, therefore, use the words exponent vector and monomial interchangeably (i.e., we identify a monomial with its exponent vector).

Monomials play a central role in GB computations. In this section, we describe the basic monomial operations and discuss basic facts about monomial (resp. polynomial) representations for GB computations.

Monomial Operations The basic monomial operations within GB computations are:

1. computations of the degree (resp. weighted degree):
   the *degree* (resp. *weighted degree*) of a monomial α is the sum of the exponents $\deg(\alpha) := \sum_{i=1}^{n} \alpha_i$ (resp. the weighted sum with respect to a weight vector w: $\deg(\alpha) := \sum_{i=1}^{n} \alpha_i\, w_i$);
2. test for divisibility:
   $\alpha | \beta \Leftrightarrow \forall i \in \{1 \ldots n\} : \alpha_i \leq \beta_i$;
3. addition of two monomials:
   $\gamma := \alpha + \beta$ with $\forall i \in \{1 \ldots n\} : \gamma_i = \alpha_i + \beta_i$;
4. comparison of two monomials with respect to a monomial ordering.

A *monomial ordering* $>$ (term ordering) on the set of monomials M_n is a total ordering on M_n which is compatible with the natural semigroup structure, i.e., $\alpha > \beta$ implies $\gamma + \alpha > \gamma + \beta$ for any $\gamma \in M_n$. A monomial ordering is a well-ordering if $(0, \dots, 0)$ is the smallest monomial. We furthermore call an ordering negative if $(0, \dots, 0)$ is the largest monomial.

Robbiano (cf. [14]) proved that any monomial ordering $>$ can be defined by a matrix $A \in GL(n, \mathbb{R})$: $\alpha > \beta \Leftrightarrow A\alpha >_{lex} A\beta$. Matrix-based descriptions of monomial orderings are very general, but have the disadvantage that their realization in an actual implementation is usually rather time-consuming. Therefore, they are not very widely used in practice.

Instead, the most frequently used descriptions of orderings have at most two defining conditions: a (possibly weighted) degree and a (normal or reverse) lexicographical comparison. We call such orderings ***simple orderings***.

Most orderings can be described as block orderings: order subsets of the variables by simple orderings.

For monomials $\alpha, \beta \in M_n$ let

$$\operatorname{lex}(\alpha,\beta) = \begin{cases} 1, & \text{if } \exists i : \alpha_1 = \beta_1, \dots, \alpha_{i-1} = \beta_{i-1}, \alpha_i > \beta_i \\ 0, & \text{if } \alpha = \beta \\ -1, & \text{otherwise,} \end{cases}$$

$$\operatorname{rlex}(\alpha,\beta) = \begin{cases} 1, & \text{if } \exists i : \alpha_n = \beta_n, \dots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i \\ 0, & \text{if } \alpha = \beta \\ -1, & \text{otherwise,} \end{cases}$$

$$\operatorname{Deg}(\alpha,\beta) = \begin{cases} 1, & \text{if } \deg(\alpha) > \deg(\beta) \\ 0, & \text{if } \deg(\alpha) = \deg(\beta) \\ -1, & \text{otherwise.} \end{cases}$$

Then we can define $\alpha > \beta$ for simple monomial orderings by lex, rlex and Deg.

We furthermore call a monomial ordering $>$ a ***degree based monomial ordering*** if $\forall\ \alpha, \beta : \deg(\alpha) > \deg(\beta) \Rightarrow \alpha > \beta$ (e.g., Dp and dp and their weighted relatives are degree based orderings).

Due to the nature of the GB algorithm, monomial operations are by far the most frequently used primitive operations. For example, monomial comparisons are performed much more often than, and monomial additions at least as often as, arithmetic operations over the coefficient field. The number of divisibility tests depends very much on the given input ideal, but is usually very large, as well.

Nevertheless, whether or not monomial operations dominate the running time of a GB computation depends on the coefficient field of the underlying polynomial ring: monomial operations are certainly run-time dominating for finite fields with a small[1] characteristic (e.g., integers modulo a small prime

[1] say, smaller than the square root of the maximal representable machine integer, i.e. smaller than $\sqrt{INT\_MAX}$

number), since an arithmetic operation over these fields can usually be realized much faster than a monomial operation. However, for fields of characteristic 0 (like the rational numbers), GB computations are usually dominated by the arithmetic operations over these fields, since the time needed for these operations is proportional to the size of the coefficients which tend to grow rapidly during a GB computation.

Therefore, improvements in the efficiency of monomial operations will have less of an impact on GB computations over fields of characteristic 0.

Monomial Representations As an illustration, and for later reference, we show below SINGULAR's internal Term_t data structure:

```
struct  Term_t
{
  Term_t*     next;
  void*       coef;
  long        exp[1];
};
```

Following the arguments outlined above, a SINGULAR polynomial is represented as a linked list of terms, where each term consists of a coefficient (implemented as a hidden type: could be a pointer or a long) and a monomial. A monomial is represented by a generalized exponent vector, which may contain degree fields (corresponding to a Deg–conditions) and contains the exponents in an order given by the monomial ordering. The size of the Term_t structure is dynamically set at run–time, depending on the monomial ordering and the number of variables in the current polynomial ring.

Based on a monomial representation like SINGULAR's, the basic monomial operations are implemented by *vectorized* realizations of their definitions.

Vectorized Monomial Operations The main idea behind what we call *vectorized monomial operations* is the following: provided that the size of the machine word is a multiple (say, m) of the size of one exponent, we perform monomial operations on machine words, instead of directly on exponents. Into this vector we include also weights (if the monomial order requires it). By doing so, we process a vector of m exponents with word operations only, thereby reducing the length of the inner loops of the monomial operations and avoiding non–aligned accesses of exponents.

The structure of this vector is determined by the monomial ordering:

1. calculate a bound on the exponents to determine the needed space: each exponent is smaller than 2^e;
2. decompose the ordering into blocks of simple orderings;
3. decompose each block into at most 2 defining conditions from lex, rlex, Deg;
4. allocate in the generalized exponent vector for each condition:

- Deg: a `long` for the weighted degree,
- lex: k `long` representing $\alpha_1, \ldots, \alpha_n$ where $k := [(ne + w - 1)/w]$ and w is the size of a `long` in bits,
- rlex: k `long` representing $\alpha_n, \ldots, \alpha_1$ where $k := [(ne + w - 1)/w]$ and w is the size of a `long` in bits,

5. allocate all not yet covered exponents at the end.

The monomial operations on generalized exponent vectors can be used to add and subtract them efficiently. [2] shows that, assuming a stricter bound, even divisibility tests can be vectorized, but we use a pre–test (see below).

In comparison to [2] this approach

- can handle any order represented by blocks of simple orderings, not only pure simple orderings;
- the size of the generalized exponent vector is a multiple of the machine word size;
- no distinction between big endian and little endian machine is necessary;
- the position of a single variable depends on the monomial ordering: access is difficult and time consuming.

Some source code fragments can probably explain it best: Figure 1 shows (somewhat simplified) versions of our implementation of the vectorized monomial operations. Some explanatory remarks are in order:

`n_w` is a global variable denoting the length of the exponent vectors in machine words.

`LexSgn` (used in `MonComp`) is a global variable which is used to appropriately manipulate the return value of the comparison routine and whose value is set during the construction of the monomial ordering from the basic conditions Deg, lex and rlex.

Notice that `MonAdd` works on three monomials and it is most often used as a "hidden" initializer (or, assignment), since monomial additions are the "natural source" of most new monomials.

Our actual implementation contains various, tedious to describe, but more or less obvious, optimizations (like loop unrolling, use of pointer arithmetic, replacement of multiplications by bit operations, etc). We apply, furthermore, the idea of "vectorized operations" to monomial assignments and equality tests, too. However, we shall not describe here the details of these routines, since their implementation is more or less obvious and they have less of an impact on the running time than the basic monomial operations.

The test for divisibility is a little bit different: while a vectorized version is possible (see [2]), it assumes a stronger bound on the exponent size, so we use another approach: most of the divisibility test will give a negative result, a simple bit operation on the bit–support of a monomial avoids most of the divisibility tests (this technique is used in CoCoA [6]: the bit–support is a `long` variable with set bits for each non–zero coefficient in the monomial). If the number of variables is larger than the bit size of `long`, we omit the last

```
// r->VarOffset encodes the position in p->exp
// r->BitOffset encodes the  number of bits to shift to the right
inline int GetExp(poly p, int v, ring r)
{
 return (p->exp[r->VarOffset[v]] >> (r->BitOffset[v]))
          & r->bitmask;
}
inline void SetExp(poly p, int v, int e, ring r)
{
  int shift = r->VarOffset[v];
  unsigned long ee = ((unsigned long)e) << shift;
  int offset = r->BitOffset[v];
  p->exp[offset]  &= ~( r->bitmask << shift );
  p->exp[ offset ] |= ee;
}
inline long MonComp(Term_t* a, Term_t* b)
{
  // return = 0, if a = b
  //        > 0, if a > b
  //        < 0, if a < b

   for (long i = 0; i<n_w; i++)
   {
     d=a->exp[i]-b->exp[i];
     if (d) return d*LexSgn[block];
   }
   return 0;
}

inline void MonAdd(Term_t* c, Term_t* a, Term_t* b)
{
   // Set c = a + b
   for (long i=0; i<n_w; i++)
     c->exp[i] = a->exp[i] + b->exp[i];
}
```

Figure 1. Vectorized monomial operations in SINGULAR

variables, if we have enough bits we use two (or more) for each variable: this block is zero for a zero exponent, has one set bit for exponent one, two set bits for exponent two and so on; and finally all bits of the block set for a larger exponent.

So much for the theory, now let us look at some actual timings: Table 1 shows various timings illustrating the effects of the vectorized monomial operations described in this section. In the first column, we list the used examples — details about these can be found at `http://www.symbolicdata.`

Table 1. Timings for vectorized monomial operations: SINGULAR in polynomial rings over coefficient field $\mathbb{Z}/32003$

Example	Singular 2.0 (s)	Singular 1.0 (s)	improvement
Ellipsoid-2	99.8	> 1000	> 10
Noon-8	123	> 1000	> 8.1
f855	236	> 1000	> 4.2
Ecyclic-7	48.3	527	11
DiscrC2	14.6	174	12

org. All GB computations were done using the degree reverse lexicographical ordering (dp) over the coefficient field $\mathbb{Z}/32003$.

The dominance of basic monomial operations and experiences vectorizing them are given in [2]. This also applies to the generalized approach described in this paper. This approach is more flexible, in particular we can use any number of bits per exponent (as long as the bounds allow it). As we would expect, the more exponents are encoded into one machine word, the faster the GB computation is accomplished. This has two main reasons: first, more exponents are handled by one machine operation; and second, less memory is used, and therefore, the memory performance increased (e.g., number of cache misses is reduced). However, we also need to keep in mind that the more exponents are encoded into one word, the smaller are the upper bounds on the value of a single exponent. But we can "dynamically" switch from one exponent size to the next larger one, whenever it becomes necessary — it is only necessary to copy the data to the representation.

We should like to point out that the monomial operations for the different orderings are identical — the different ordering results in a different encoding only. Therefore, we can use one inlined routine instead of function pointers (and, therefore, each monomial operation requires a function call). Already the in–place realization of monomial operations results in a considerable efficiency gain which by far compensates the additional overhead incurred by the additional indirection for accessing one (single) exponent value (i.e., the overhead incurred by the `GetExp` macro shown above).

Last, but not least, the combination of all these factors with the new memory management leads to a rather significant improvement over the "old" SINGULAR version 1.0.

4 Arithmetic in Fields

Computer algebra systems use a variety of numbers from fields as one basic data type. Numerical algorithms require floating point numbers (with several levels of accuracy, possibly adjustable), while other algorithms from computer algebra use exact arithmetic.

4.1 Arithmetic in $\mathbb{Z}/p$ for Small Primes p

Representation by a representant from $[0 \dots p-1]$ fits into a machine word. Addition and subtraction are implemented in the usual way, multiplication and division depend on the properties of the CPU used. The two possibilities are

1. compute the inverse via Euclidean algorithm and keep a table of already computed inverses;
2. use a log–based representation for multiplication and division and keep tables for converting from one representation to the other.

We found that the relative speed of these two methods depends on the CPU type (and its memory band width): on PC processors there is nearly no difference while workstations (SUN, HP) prefer the second possibility.

4.2 Rational Numbers

For the arithmetic in $\mathbb{Z}$ or $\mathbb{Q}$ we rely on the GMP library ([9]), with the following two additions:

- during GB computations we try to avoid rationals: so we have only one large number per monomial;
- small integers may be inlined as LISP interpreters do: they are represented by an immediate integer handle, containing the value instead of pointing to it, which has the following form: (guard bit, sign bit, bit 27, ..., bit 0, tag 0, tag 1). Immediate integer handles carry the tag '01' i.e. the last bit is 1. This distinguishes immediate integers from other handles which point to structures aligned on four byte boundaries and, therefore, have last bit zero (the second bit is reserved as tag to allow extensions of this scheme). Using immediates as pointers and dereferencing them gives address errors. To aid overflow check the most significant two bits must always be equal, that is to say that the sign bit of immediate integers has a guard bit, (see [16]).

5 Arithmetics for Polynomials: Refinement: Bucket Addition

In situations with a lot of additions, where only little knowledge about the intermediate results is required, the *geobucket* method provides good results — it avoids the $O(n^2)$–complexity in additions (merge of sorted lists) by postponing them. It is used during computation of a Gröbner basis: the reduction routine will be replaced by one which will handle long polynomials more efficiently using geobuckets, which accommodate the terms in buckets of geometrically increasing length (see [17]). This method was first used successfully by Thomas Yan, at that time a graduate student in CS at Cornell. Other algorithms which profit from geobuckets are the evaluation of polynomials and the Bareiss algorithm [4].

6 Memory Management

Most of SINGULAR's computations boil down to primitive polynomial operations such as copying, deleting, adding, and multiplying of polynomials. For example, standard bases computations over finite fields spent (on average) 90 % of their time realizing the operation `p - m*q` where `m` is a monomial, and `p,q` are polynomials.

SINGULAR uses linked lists of monomials as data structure for representing polynomials. A monomial is represented by a memory block which stores a coefficient and an exponent vector. The size of this memory block varies: its minimum size is three machine words (one word for each, the pointer to the "next" monomial, the coefficient and the exponent vector), its maximal size is (almost) unlimited, and we can assume that its average size is four to six machine words (i.e., the exponent vector is two to four words long).

From a low-level point of view, polynomial operations are operations on linked lists which go as follows:

1. do something with the exponent vector (e.g., copy, compare, add), and possibly do something with the coefficient (e.g., add, multiply, compare with zero) of a monomial;
2. advance to the next monomial and/or allocate a new or free an existing monomial.

Assuming that coefficient operations are (inlined) operations on a single machine word (as they are, for example, for coefficients from most finite fields), and observing that the exponent vector operations are linear w.r.t. the length of the exponent vector, we come to the following major conclusion:

For most computations, a traversal to the next memory block or an allocation/deallocation of a memory block is, on average, required after every four to six machine word operations.

The major requirements of a memory manager for SINGULAR become immediately clear from this conclusion:

(1) allocation/deallocation of (small) memory blocks must be extremely fast If the allocation/deallocation of a memory blocks requires more than a couple of machine instructions it would become the bottleneck of the computations. In particular, even the overhead of a function call for a memory operation is already too expensive;
(2) consecutive memory blocks in linked lists must have a high locality of reference. Compared with the performed operations on list elements, cache misses (or, even worse, page misses) of memory blocks are extremely expensive: we estimate that one cache miss of a memory block costs at least ten or more list element operations. Hence, if the addresses of consecutive list elements are not physically closed to each other (i.e., if the linked lists have a low locality of reference), then resolving cache (or, page) misses would become the bottleneck of computations.

Furthermore, there are the following more or less obvious requirements on a memory manager for SINGULAR:

(3) the size overhead to maintain small blocks of memory must be small. If managing one memory block requires one or more words (e.g., to store its size) then the total overhead of the memory manager could sum up to 25 % of the overall used memory, which is an unacceptable waste of memory;
(4) the memory manager must have a clean API and it must support debugging. The API of the memory manager must be similar to the standard API's of memory managers, otherwise its usability is greatly limited. Furthermore, it needs to provide error checking and debugging support (to detect overwriting, twice freed or not–freed memory blocks, etc.) since errors in the usage of the memory manager are otherwise very hard to find and correct;
(5) the memory manager must be customizable, tunable, extensible and portable. The memory manager should support customizations (such as "what to do if no more memory is available"); its parameters (e.g., allocation policies) should be tunable, it should be extensible to allow easy implementations of furthergoing functionality (like overloading of C++ constructors and destructors, etc), and it needs to be portable to all available operating systems.

To the best of our knowledge, there is currently no memory manager available which satisfies these (admittedly in part extreme) requirements. Therefore, we designed and implemented `omalloc.` ([1]).

`omalloc` manages small blocks of memory on a per–page basis. That is, each used page is split up into a page header and equally sized memory blocks. The page header has a size of six words (i.e., 24 bytes on a 32 bit machine), and stores (among others) a pointer to the free–list and a counter of the used memory blocks of this page.

On *memory allocation*, an appropriate page (i.e., one which has a non–empty free list of the appropriate block size) is determined, based on the used memory allocation mechanism and its arguments (see below). The counter of the page is incremented, and the provided memory block is dequeued from the free–list of the page.

On *memory deallocation*, the address of the memory block is used to determine the page (header) which manages the memory block to be freed. The counter of the page is decremented, and the memory block is enqueued into the free–list. If decrementing the counter yields zero, i.e., if there are no more used memory blocks in this page, then this page is dequeued from its bin and put back into a global pool of unused pages.

This design results in

- allocation/deallocation of small memory blocks requires (on average) only a couple (say, 5-10) assembler instructions;

- `omalloc` provides an extremely high locality of reference (especially for consecutive elements of linked-list like structures). To realize this, recall that consecutively allocated memory blocks are physically almost always from the same memory page (the current memory page of a bin is only changed when it becomes full or empty);
- small maintenance size overhead: Since `omalloc` manages small memory blocks on a per-page basis, there is no maintenance-overhead on a per-block basis, but only on a per-page basis. This overhead amounts to 24 Bytes per memory page, i.e., we only need 24 Bytes maintenance overhead per 4096 Bytes, or 0.6 %.

  For larger memory blocks, this overhead grows since there might be "left-overs" from the division of a memory page into equally sized chunks of memory blocks. Furthermore, there is the additional (but rather small) overhead of keeping Bins and other "global" maintenance structures.

7 A Proposal for Distributing Polynomials

Computing in a distributed fashion is coming of age, driven in equal parts by advances in technology and by the desire to simply and efficiently access the functionality of a growing collection of specialized, stand-alone packages from within a single framework. The challenge of providing connectivity is to produce homogeneity in a heterogeneous environment, overcoming differences at several levels (machine, application, language, etc.). This is complicated by the desire to also make the connection efficient. We have explored this problem within the context of symbolic computation and, in particular, with respect to systems specially designed for polynomial computations.

We tested our ideas with the mechanisms found in the Multi Protocol: dictionaries, prototypes, and annotations. A detailed description of the implementation is given in [3].

A polynomial is more than just an expression: the system usually has a default representation (dense-distributive in the case of SINGULAR, ordered by the monomial ordering). Furthermore, the number of variables, the coefficient field are also known in advance and, usually, constant for all arguments of polynomial operations or structures.

The polynomial objects consist of several parts: the description of the structure (which may be shared with other objects of the same type), the actual raw data, and a list of properties. Understanding the structure is necessary for each client: they would not be able to decode the data otherwise. On the other side, the properties should be considered as hints: a typesetting programme does not need to understand the property of an ideal to be a Gröbner basis.

The following list of requirements should be fulfilled by a communication protocol:

- the encoding of polynomial data should closely follow the internal representation of the sender;
- several possibilities for the encoding of polynomial data must be provided: corresponding to several possibilities for data representation (distributive, recursive, as expression tree);
- the description of the structure should be shared between objects;
- possibility to provide additional information which may or may not be used;
- flexibility in the set of provided operations and properties.

While MP provides the first requirements, it fails to fulfill the last one: it is difficult to extend and only at compile time.

Currently there are several protocols in use which communicate mathematical expressions. OpenMath ([12]) seems to be the most promising in terms of generality, but it has still to prove that it provides also fast links — but, as I think, it may become the standard we need.

References

1. Bachmann, O.: An Overview of `omalloc`, Preprint 2001.
2. Bachmann, O.; Schönemann, H.: Monomial Representations for Gröbner Bases Computations, ISSAC 1998, 309-316.
3. Bachmann, O.; Gray, S.; Schönemann, H.: MPP: a Framework for Distributed Polynomial Computations, ISSAC 1996, 103-112.
4. Bareiss, R.H.; Sylvester's identity and multistep integer-preserving Gaussian elimination, Math. Comput. 22(1968), 565-578.
5. Bayer, D.; Stillman, M.: A Theorem on Refining Division Orders by the Reverse Lexicographic Order, Duke Math. J. 55 (1987), 321-328.
6. Bigatti, A.M.: Computation of Hilbert-Poincaré Series, J. Pure and Appl. Algebra 119, 237–253, 1997.
7. Buchberger, B.: Ein Algorthmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Ideal, Thesis, Univ. Innsbruck, 1965.
8. Faugère, J.-C.: A new efficient algorithm for computing Grobner bases (F4). Journal of Pure and Applied Algebra 139, 1-3, 61-88 (1999).
9. Granlund, T.: The GNU Multiple Precision Arithmetic Library, `http://www.swox.com/gmp/`
10. Hironaka, H.: Resolution of singularities of an algebraic variety over a field of characteristic zero. I, II, Ann. of Math. (2) 79, 109-203 (1964), 79, 205-326, (1964).
11. Möller, H.M., Mora, T.: New Constructive Methods in Classical Ideal Theory. Journal of Algebra 100, 138-178 (1986)
12. OpenMath, `http://www.openmath.org/`
13. Pfister, G.; Schönemann, H.: Singularities with exact Poincaré complex but not quasihomogeneous, Revista Matematica 2 (1989)
14. Robbiano, L.: Term Orderings on the Polynomial Ring, Proceedings of EUROCAL 85, Lecture Notes in Computer Science 204
15. Takayama, N.: Developpers of Gröbner Engines, Sugaku no Tanoshimi Vol 11, 35–48,(1999)
16. Schönert, M. et al.: AP – Groups, Algorithms, and Programming. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, fifth edition, 1995.
17. Yan, T.: The Geobucket Data Structure for Polynomials, Journal of Symbolic Computation 25, 3 (1998), 285-294.