ACM Communications in Computer Algebra, Vol. 48, No. 4, Issue 190, December 2014

The design of Maple's sum-of-products and POLY data structures for representing mathematical objects.

Michael Monagan and Roman Pearce
Center for Experimental and Constructive Mathematics,
Simon Fraser University, Burnaby, British Columbia.

EMail: mmonagan@cecm.sfu.ca and rpearcea@cecm.sfu.ca

Abstract

The principal data structure Maple uses to represent polynomials and general mathematical expressions involving functions like $\sin x, e^{2x}, y'(x), \binom{n}{k}$ etc., is known to the Maple developers as the sum-of-products data structure. Gaston Gonnet, as the primary author of the Maple kernel, designed and implemented this data structure in the early 1980s. As part of the process of simplifying a mathematical formula, he represented every Maple object and every sub-object uniquely in memory. This makes testing for equality of expressions very fast. In this article, on occasion of Gonnet's retirement, we present details of his design, its pros and cons, and changes we and others have made to it over the years. One of the cons of the sum-of-products data structure is it is not as efficient at multiplying multivariate polynomials as other special purpose computer algebra systems. We describe a new data structure called POLY that we added to Maple 17 (released in 2013) to improve performance for polynomials in Maple, and recent work done for Maple 18 (released in 2014).

1 Introduction

The sum-of-products data structure is the main data structure Maple uses to represent mathematical formulas or expressions. As a data structure, it is a directed acyclic graph in which each node in the graph is encoded by a vector (array) of machine words. The nodes represent numbers, sums, products, powers, functions, etc. We will refer to the nodes as sub-expressions and as sub-objects. An example is given in Figure 1.

A unique feature of Maple among computer algebra systems is that almost all objects and sub-objects in Maple are represented uniquely in memory. This is done using hashing after algebraic rules have been used to simplify an object. As a consequence, Maple can test two objects for equality using a pointer comparison that costs one clock cycle. This makes many Maple operations very efficient. This feature and the sum-of-products data representation are mentioned only briefly by Char, Geddes, Gentleman and Gonnet in [2], the first paper on the design of Maple. Details are not presented and the disadvantages of these design decisions are not discussed.

In this article we provide details showing how Maple represents objects and how this sharing of objects and sub-objects is implemented in Section 2. In Section 3 we discuss two advantages of this design and in Section 4 three disadvantages. We focus on problems with this design and the solutions that we and others have made to address the problems. A fourth disadvantage is that polynomial multiplication and division, which are fundamental to the overall efficiency of a computer algebra system like Maple, are relatively slow compared with special purpose computer algebra systems like Magma [1] and Singular [7] and others that automatically use dedicated polynomial data structures. In Section 5 we will give details of our new polynomial data structure that we have added to the Maple 17 kernel to address this.

2 Maple's Sum-Of-Products Representation

Figure 1 below shows the default polynomial data structure in Maple 16 and all previous versions of Maple for the polynomial $f = 9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$. It is a "sum-of-products" where each monomial in the polynomial is stored as a separate Maple object, a `PROD` object.

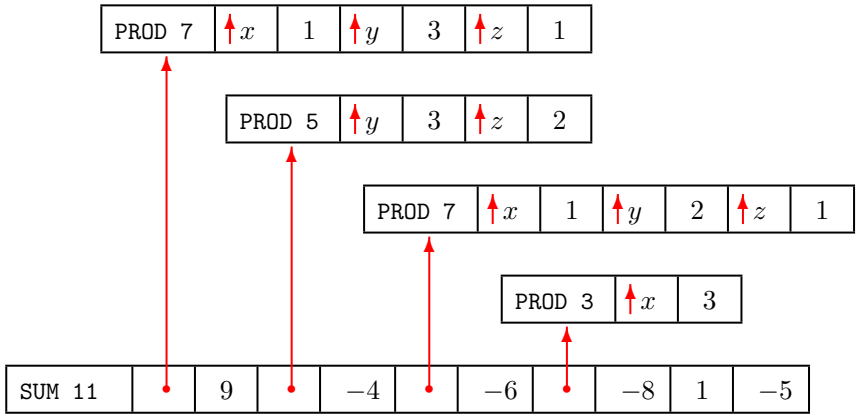

Figure 1: Maple's sum-of-products representation for the polynomial $9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$.

Each object in Maple has a header word that encodes the type of the object and its length in machine words. Thus the `SUM` in Figure 1 occupies 11 words of memory. The remaining words of each object contain the content of the object. For a `SUM`, this is a sequence of (monomial,coefficient) pairs and for a `PROD`, it is a sequence of (base,exponent) pairs as shown in Figure 1. The coefficients and exponents must be numbers, that is, integers, fractions, decimal numbers, or complex numbers.

Not shown explicitly in Figure 1 is how the variables x, y, z are represented. What is stored in the `PROD` objects are pointers (indicated by ↑) to `NAME` objects that are represented like this | `NAME 3` | *nil* | x |. A variable is a `NAME` whose second word is its value. If a variable has not been assigned a value then its value is nil. The third and subsequent words store the characters of the variable ending with at least one 0 byte.

Small integers are stored immediately in the data structure. A large integer is stored as a pointer to a separate `INTPOS` or `INTNEG` object. Small integers are encoded as follows to distinguish them from pointers that have a 0 least significant bit. On a 64 bit computer, if an integer x satisfies $-2^{62} < x < 2^{62}$, then x is stored as $2x + 1$ that has 1 as the least significant bit. We call such an integer an *immediate* integer.

Looking at the data structure, consider what Maple must do to compute the degree of the polynomial in x and the set $\{x, y, z\}$ of all the variables in the polynomial. If the polynomial has t terms in n variables, both operations are $O(nt)$ because Maple must traverse the entire data structure to determine the degree in x and find all variables. Even when the degree operation sees x in the monomial xy^3z in Figure 1, it must continue looking for x in the remaining fields of | `PROD 7` | x | 1 | y | 3 | z | 1 | because this sub-object could be $x(x+1)^3z$ or $xy^3 \sin x$. In fact, almost all Maple operations are at least $O(nt)$ because the contents of a Maple sum-of-products object are unknown.

Figure 2 below shows how Maple represents the expression $g = 2x^2 \sin x - 3 \sin x \cos x + 5 \cos x$.

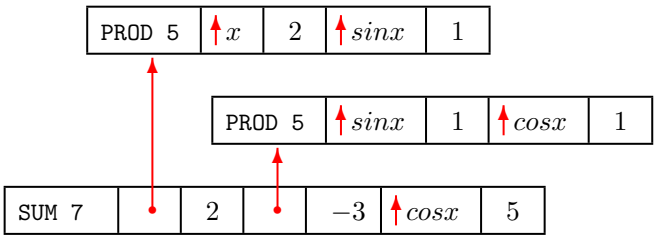

Figure 2: The sum-of-products representation for $g = 2x^2 \sin x - 3 \sin x \cos x + 5 \cos x$.

Notice that the top two levels of the data structure are `SUM` and `PROD` objects. Figure 3 shows how the functions $\sin x$ and $\cos x$ are represented. Note that even though $\sin x$ and $\cos x$ are functions of one argument, the argument sequence `SEQ 2` | $\uparrow x$ is not simplified to x for uniformity.

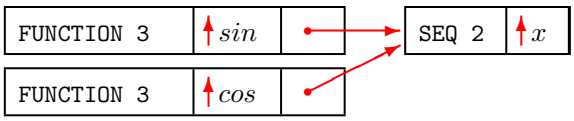

Figure 3: Maple's data representation for the functions $\sin x$ and $\cos x$.

Notice in Figure 2 there are two pointers to $\sin x$ and two pointers to $\cos x$. Are there two copies of the $\sin x$ object and two copies of $\cos x$ object? No. When the `SUM` object in Figure 2 is simplified, if two copies of $\sin x$ and $\cos x$ are identified, only the first is kept. This unique representation of objects is automatic. It clearly saves space for formulas involving functions. It is less obvious that this can save space for polynomials. Looking back at Figure 1, except for the sharing of the variables, there are no common subexpressions that can be shared. However, if you are computing with several fully expanded polynomials in x, y, z then very likely some or many of the monomials in x, y, z will appear repeatedly in the set of polynomials. They will all be shared.

2.1 Unique representation using hashing.

We provide details on how hashing is used in simplification to identify equal objects and how unique representation of objects is in turn exploited to to speed up simplification. Suppose the user inputs the polynomial $f = 3x^2 + xy^2 + 2y^2x + 3/2y^2 - x^2$ to Maple. It will be stored as

SUM 11	$\uparrow x^2$	3	$\uparrow xy^2$	1	$\uparrow y^2x$	2	$\uparrow y^2$	3/2	$\uparrow x^2$	-1

How does Maple simplify it? It uses the following procedure.

Step 1 (simplify monomials recursively)

Simplify the monomials $x^2, xy^2, yx^2, y^2, x^2$ recursively but not the coefficients $3, 1, 2, 3/2, -1$. To improve efficiency, Maple uses one bit in the header word of each object to indicate whether it has been simplified already. After this step each monomial will be represented uniquely, that is, there will be one copy of the monomial x^2 stored as `PROD 3` | $\uparrow x$ | 2 and the two pointers in the `SUM` to x^2 will be the same.

Step 2 (add coefficients)

Add coefficients of like monomials. For small polynomials with few terms Maple does this without sorting. For large polynomials Maple sorts the `SUM` on the monomials, that is, on the pointers treating them as unsigned integers so that like terms are now adjacent.

SUM 11	$\uparrow x^2$	2	$\uparrow x^2$	-1	$\uparrow xy^2$	1	$\uparrow xy^2$	2	$\uparrow y^2$	$\uparrow 3/2$

.

Sorting by pointers is very fast. In other computer algebra systems, to compare monomials one would have to use a general comparison. This is one reason why Maple is fast at polynomial arithmetic.

Now we add the coefficients of like terms in linear time keeping only terms with non-zero coefficient. After disposing of the memory no longer needed we obtain

SUM 7	$\uparrow x^2$	1	$\uparrow xy^2$	3	$\uparrow y^2$	$\uparrow 3/2$

.

Step 3 (simplify coefficients)

Simplify the coefficients that are not immediate integers so that they are also uniquely represented. In our example the fraction 3/2 that is represented by | RATIONAL 3 | 3 | 2 | is simplified.

Step 4 (uniquification)

Compute a hash value for the `SUM` object. Do this using the numerical values of the coefficients and monomials treating pointers as unsigned integers. The hash function must be commutative across the terms of the sum so that, for example, hash$(2x+3y)$ = hash$(3y+2x)$. Similarly, the hash function for a product must be commutative across the factors of the product so that hash(x^2y^3) = hash(y^3x^2).

Search the `simpl` table, a global hash table of all Maple objects to see if this new object already exists. If it has the same hash value as the hash value of an object in the `simpl` table, then compare the terms of the two sums by comparing pointers and immediate integers.

If this object already exists in the `simpl` table return a pointer to the copy in the `simpl` table. Otherwise set a bit in the header word to indicate that this object has been simplified, insert this object into the `simpl` table and return this object.

An obvious question is why in Step 2 are small polynomials not sorted? The original reason was that Maple tries to preserve a user's preference for a formula. For example, if the user enters $\sqrt{1-x^2}$ it was considered desirable to not simplify this to be $\sqrt{-x^2+1}$. Similarly it may be desirable to see $(B-A)^2$ instead of $(-A+B)^2$. However, sorting large sums by pointers results in a random looking outputs that is unsatisfactory. We address the issue of sorting in Section 4.1.

2.2 Hashing.

To store objects uniquely in memory, Maple maintains a hash table of all objects called the `simpl` table. The hash function for `SUM`s and `PROD`s is a function of the pointers. So it too takes advantage of unique representation. The hash function used for a `SUM` and `PROD` object in Maple 6 is shown in the following C code snippet. Here s is the object and the values of the variables r, L, i, p are signed integers, 32 bits on a 32 bit machine and 64 bits on a 64 bit machine.

```
case SUM_CASE:
case PROD_CASE:
    r = ID_DAG(s);
    L = LENGTH(s);
    /* commutative signature function. */
    for( i=1; i < L; i+=2 ) {
        p = HASHMUL * I(s[i]) + I(s[i+1]);
        p += p >> 4;
```

```
    r += p * (2*p+1);
}
break;
```

In this code the LENGTH function returns the length of the object in words, the ID_DAG function returns the bit encoding of the type of the object, the macro I(...) is a cast to an integer type, and the subscripts $s[i]$ and $s[i+1]$ access a (monomial,coefficient) pair.

The value of the constant HASHMUL is 1027 which is a pseudo primitive root (has maximal order) mod 2^{32} and 2^{64}. A comment in the code says that 1664525 was previously used but changed to 1027 because 1027 in binary has 3 one bits so that multiplication by 1027 can be optimized to shifts and adds for RISC machines. The constant 1664525 was taken from Knuth's table of results of the spectral test on page 102 of [8].

Now this hash function is commutative on the terms of a sum and the factors of a product, so that $\text{hash}(2x+3y) = \text{hash}(3y+2x)$ and $\text{hash}(x^2y^3) = \text{hash}(y^3x^2)$, but unlike the original hash function in [2], it is not commutative across the coefficients and exponents so that $\text{hash}(2x+3y) \neq \text{hash}(3x+2y)$ and $\text{hash}(y^3x^2) \neq \text{hash}(y^2x^3)$ with high probability. The hash function used prior to Maple 4.2 was commutative on the coefficients and monomials in a sum and the exponents and variables in a product. We noticed this was a problem for Vandermonde determinants and changed the hash function to use the above code. We explain the problem.

Let V_n be the $n \times n$ Vandermonde matrix in n variables. For example

$$V_4 = \begin{bmatrix} 1 & w & w^2 & w^3 \\ 1 & x & x^2 & x^3 \\ 1 & y & y^2 & y^3 \\ 1 & z & z^2 & z^3 \end{bmatrix}.$$

Maple's determinant command computes $\det(V_n)$ in expanded form. For V_4 it obtains

$$w^3x^2y - w^3x^2z - w^3xy^2 + w^3xz^2 + \ \cdots\ - xy^3z^2 + xy^2z^3$$

The determinant in expanded form has $n!$ terms. In the sum-of-products representation these terms look like | PROD 7 | w | 3 | x | 2 | y | 1 |. Notice that in $\det(V_4)$ there are $(n-1)!$ monomials $w^3x^2y^1$, $w^3x^1y^2$, $w^2x^3y^1$, $w^2x^1y^3$, $w^1x^3y^2$ and $w^1x^2y^3$ which as PRODs are permutations of the exponents $1,2,3$ and variables x,y,z, which, using the original hash function, all hash to the same value.

2.3 Programming with Maple's sum-of-products representation.

The two Maple commands `op` and `nops` give the Maple programmer the ability to pick apart any Maple expression (object). They provide an interface between the actual sum-of-products data representation and the Maple programming environment. The command `nops(f)` returns the number of operands (components parts) of an expression f and the command `op(i,f)` for $i > 0$ returns the the i'th operand of f. The special case `op(0,f)` returns the type of f. The following table shows how this works.

f	`op(0,f)`	`nops(f)`	`op(1,f)`	`op(2,f)`
$2x+3y+z$	`'+'`	3	$2x$	$3y$
x^3yz	`'*'`	3	x^3	y
binomial(n,k)	binomial	2	n	k
$[x,x^2]$	list	2	x	x^2
$2/3$	Fraction	2	2	3

3 Some Advantages

3.1 Algebraic Numbers and RootOfs

In [5] Geddes, Gonnet and Smedley implemented a heuristic algorithm for computing a GCD of two polynomials over an algebraic number field. The Maple group needed to decide how we would represent algebraic numbers in Maple. Gonnet proposed and implemented the following design that uses a Maple function as the representation.

If α is an algebraic number with minimal polynomial $m(z)$ then the Maple user can input α to Maple as RootOf$(m(z), z_0)$ or RootOf$(m(z), a..b)$. Here z_0 is input as a (complex) floating point approximation to α with sufficiently many decimal places to distinguish it from the other roots and $a..b$ is real interval (or complex box) that isolates α from the other roots. For example, the algebraic number $\sqrt{2}$ with minimal polynomial $z^2 - 2$ may be input as

```
> alpha := RootOf(z^2-2,1.4..1.5);
```

$$\alpha := RootOf(\_Z^2 - 2,\ 1.4..1.5);$$

```
> op(0,alpha), op(1,alpha), op(2,alpha);
```

$$\text{RootOf},\quad \_Z^2 - 2,\quad 1.4..1.5$$

```
> simplify(alpha^3);
```

$$2\,RootOf(\_Z^2 - 2,\ 1.4..1.5)$$

```
> evalf(alpha);
```

$$1.414213562$$

Notice that the representation for the algebraic number α is a function with the name RootOf and arguments $\_Z^2 - 2$ and 1.4..1.5 and that Maple replaced z by the canonical $\_Z$. Thus RootOf$(z^2 - 2, 1.4..1.5)$ results in the same object as RootOf$(x^2 - 2, 1.4..1.5)$.

However, as the following shows, RootOf$(z^2 - 2, 1.4..1.5)$ and RootOf$(z^2 - 2, 1.414)$ result in distinct objects that the main simplifier does not recognize as being equal. We can see this when $\alpha - \beta$ is created and not simplified to 0 which is a potential problem. It would be better if Maple imposed a canononical for an algebraic number. The same problem occurs whenever two mathematically equal objects are created but not automatically simplified to the same object. For example, in Maple, the exponentials e^{2x} and $(e^x)^2$ are not the same objects and hence $e^{2x} - (e^x)^2$ is not automatically simplified to 0.

```
> beta := RootOf(z^2-2,1.414);
```

$$\beta := RootOf(\_Z^2 - 2,\ 1.414)$$

```
> alpha-beta;
```

$$RootOf(\_Z^2 - 2,\ 1.4..1.5) - RootOf(\_Z^2 - 2,\ 1.414)$$

Consider now the polynomial $f = \alpha x^2 - 3\alpha + 1$ in Maple.

```
> f := alpha*x^2-3*alpha+1;
```

$$\text{RootOf}\left(\_Z^2 - 2, 1.4..1.5\right) x^2 - 3\,\text{RootOf}\left(\_Z^2 - 2, 1.4..1.5\right) + 1$$

Because of the unique representation of Maple objects, one copy of the RootOf α is stored, and this representation is space efficient. However, the display is poor. In some versions of Maple, including the command line interface, Maple will automatically identify common subexpressions in the output and display them using $\%1, \%2, \ldots$ labels if they appear 3 or more times in the output. Maple will display those common subexpressions like this $\%1 = RootOf(\_Z^2 - 2, 1.4 .. 1.5)$ immediately below the output. This general facility is helpful in reducing the size of the output. However, Maple needs a nice way to display formulas involving RootOfs, preferably allowing us to display them as Greek letters which is standard for algebraic numbers. Gonnet suggested, in the discussion, that Maple store the information about an algebraic number α in a global table and use a Maple variable α to represent the algebraic number. This proposal, which would be a side effect on a Maple variable, was rejected because it would break Maple's model for *option remember* (see Section 3.2).

To improve the display, we added a general alias mechanism for this purpose. The command `alias(x=y)` means, when user inputs x, Maple gets y, and when Maple outputs y, the value x is displayed to the user. We illustrate this by redoing the example.

```
> alias( beta=RootOf(z^2-2,1.414) ):
> f := beta*x^2 - 3*beta + 1;
```

$$f := \beta\, x^2 - 3\beta x + 1$$

```
> g := evala( Expand(f*f) );
```

$$g := 2\, x^4 + 2\, \beta x^2 - 12\, x^2 - 6\, \beta + 19$$

```
> evalf(g);
```

$$2.0\, x^4 - 9.171572876\, x^2 + 10.51471863$$

Aliasing is implemented by simple substitution, i.e., by `subs(x=y,expr)` on input and `subs(y=x,expr)` on output. The `subs` command scans the input expression for x, replacing it by y. Substitution is linear time because there is a unique copy of x and comparison of objects in the expression with x is a pointer comparison.

3.2 Remember tables and option remember.

At GNOME 2014, in his talk, Gonnet presented *option remember* as his favourite feature of Maple. This feature, mentioned in the 1983 paper [2], has been refined and extended in the years since. Consider the following Maple function for computing the n'th Fibonacci number F_n using the recurrence $F_n = F_{n-1} + F_{n-2}$ with initial values $F_0 = 0$ and $F_1 = 1$.

```
F := proc(n::nonnegint)
         if n=0 then 0 elif n=1 then 1 else F(n-1)+F(n-2) end if;
     end;
```

Every computer scientist knows that this code is terrible. The two recursive calls mean that the cost of computing F_n is $O(2^n)$ function calls, e.g., computing F_{30} does 2692537 function calls. But, by simply adding *option remember* to this Maple code as shown below, the number of function calls is reduced to n.

```
F := proc(n::nonnegint)
         option remember;
         if n=0 then 0 elif n=1 then 1 else F(n-1)+F(n-2) end if;
     end;
```

The way *option remember* works is as follows. Associated with this Maple procedure is a hash table called a *remember table*. When the function F is called with a given argument n, the remember table is searched for an entry with key n. If not present, the code in the Maple procedure is executed to compute the value of F_n. Before the value is returned, the pair (n, F_n) with key n is inserted into the remember table. If present, the value F_n is retrieved from the remember table and returned. Therefore the code for a Maple procedure with option remember is executed once for a given argument sequence.

For Maple 4.2 the first author added a syntactically nicer version of option remember based on assignment. The following Maple code is equivalent to the previous version of F with option remember. It makes the initial values explicit.

```
F := proc(n::nonnegint) F(n) := F(n-1)+F(n-2) end;
F(0) := 0;
F(1) := 1;
```

The remember table is created when the first assignment `F(0) := 0;` is executed. The only values stored in the remember table are the values that are explicitly assigned. This gives the user the ability to selectively remember values. This way of using a remember table makes F look more like a table than a function. It allows one to define a discrete function as a table where associated with the table is a Maple procedure that is used to compute missing values from the table on demand. Two questions need to be asked about option remember. First, how efficient is it and second, how useful is it?

How efficient is a remember table?

A remember table is a Maple hash table. When $f(x) := y$ is computed, (more generally when $f(x_1, x_2, ..., x_n) := (y_1, y_2, ..., y_m)$ is computed), the pair (x, y) with key x is stored in the remember table. Because Maple stores unique values of objects in memory, the hash function used for the key x is the address of x. Thus the cost of the hash function is $O(1)$. In a system that does not store unique copies of objects, hashing and testing for equality would likely be at least linear in the size of the object x. We have measured actual the cost of using option remember in Maple by doing the following experiment.

```
> n := 10^6:
> st := time(): for i to n do i od: looptime := time()-st;
> g := proc(x) option remember; x end;
> st := time(): for i to n do g(i) od: gcomptime := time()-st-looptime;
> st := time(): for i to n do g(i) od: gremembertime := time()-st-looptime;
> f := proc(x) x end;
> st := time(): for i to n do f(i) od: fcomptime := time()-st-looptime;
```

On an AMD 248 Opteron at 2.2GHz we get the following timings (in CPU seconds) averaged on 3 runs; looptime = 0.186s, gcomptime = 1.899s, gremembertime = 1.163s, and fcomptime = 1.144s. Thus the cost of inserting into the remember table is less than the cost of executing a Maple procedure and looking up a remember table is about the same as executing a Maple procedure call.

The fact that the cost of adding option remember to a function is very low has led to a proliferation of code in the Maple library that uses option remember. For example, the `ifactor` and `factor` commands in

Maple that factor integers and polynomials respectively, have option remember. Adding *option remember* also has a space cost. For each entry (x, y) remembered, Maple must insert it into the remember table. This costs asymptotically two words of memory to store the pair (x, y). To prevent Maple from running out of memory, if option *system* is also added, when garbage collection is triggered, an entry (x, y) is deleted from a remember table if no live objects reference both x and y. Both `ifactor` and `factor` have option system. Another feature added to Maple 10 in 2005 to prevent the remember table from getting too big is option cache(n) that creates a remember table with a limit on the number of entries set at n.

What does *option remember* really buy us?

Many functions in mathematics have recursive definitions. Examples include the Chebyshev polynomials $T_n(x)$ that satisfy $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$ and the binomial coefficients $\binom{n}{k}$ that satisfy $\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1}$. This remembering of values previously computed means that if the Maple user does a computation like

```
> n := 14;
> add( k*binomial(n,k), k=1..n );
```

that computes $\sum_{k=0}^{n} k\binom{n}{k}$, the total cost of computing this sum is $O(n)$ integer operations. The user automatically benefits from the recursive definition without having to worry that Maple is recomputing the binomial coefficients from scratch.

Just adding *option remember* makes coding these functions easy. The many refinements to the basic option remember facility reflect the large number of applications where remembering values is useful in Maple. As a serious application, we present a compact implementation of Shanks' baby-step-giant-step algorithm for computing a discrete logarithm.

Suppose α is a generator of the integers modulo p, a prime. The discrete logarithm problem is, given an integer y satisfying $0 < y < p$, solve

$$y = \alpha^x \bmod p$$

for x, that is, compute the discrete logarithm $x = \log_\alpha y$. A simple way to find x is to enumerate the powers of α

$$1, \alpha, \alpha^2, \alpha^3, ..., \alpha^x = y$$

stopping when we get to y. Equivalently we can multiply y by α^{-1} and compute

$$y, y\alpha^{-1}, y\alpha^{-2}, ..., y\alpha^{-x} = 1$$

stopping when we get to 1. Both approaches do $(p-1)/2$ multiplications on average. Letting $m = \lfloor\sqrt{p-1}\rfloor$ Shanks' baby-step giant-step algorithm pre-computes and stores the m powers

$$\alpha^0 = 1, \alpha^m, \alpha^{2m}, ..., \alpha^{m(m-1)}$$

using $\Theta(\sqrt{p})$ multiplications. To compute $\log_\alpha y$, when computing the sequence $y, y\alpha^{-1}, y\alpha^{-2}, \ldots$, Shank's algorithm stops as soon as one of the pre-computed powers is found in this sequence. This reduces the cost of computing $\log_\alpha y$ from $\Theta(p)$ to $\Theta(\sqrt{p})$ multiplications. One must be able to search the table of pre-computed powers efficiently. In a typical presentation of the algorithm, e.g., Stinson [12], one sorts the powers and uses binary search. We offer the following elegant Maple solution that defines the function LOG and stores m pre-computed powers in LOG's remember table and hence uses hashing to search for them. This means the expected time for our solution is $O(\sqrt{p})$ instead of $O(\sqrt{p}\log p)$.

```
Shanks := proc(p::prime,alpha::posint)
local invalpha,LOG,m,am,pow,i;

    invalpha := 1/alpha mod p;
    LOG := proc(y) 1+LOG(y*invalpha mod p) end;
    LOG(1) := 0;

    m := isqrt(p-1);
    am := alpha &^ m mod p;
    (pow,i) := (1,1);
    for i while i*m<p-1 do
         pow := am*pow mod p;
         LOG(pow) := i*m;      # record that LOG(alpha^(im)) = im
    end do;
    return LOG;
end;
```

4 Some Disadvantages

4.1 Sorting by address is not nice!

Because terms in large sums and elements of sets are sorted by address, the order of the terms in a sum and elements of a set will appear in a random looking order. The expanded polynomial below is an example. The set $\{\frac{1}{3}, \frac{2}{3}, \frac{1}{4}\}$ could display as $\{\frac{1}{3}, \frac{2}{3}, \frac{1}{4}\}$ in one Maple session but as $\{\frac{1}{4}, \frac{2}{3}, \frac{1}{3}\}$ in another. Any permutation is possible. It simply depends where in memory the fractions are first created. This has been a very annoying feature for Maple users.

The solution for sets, finally introduced in Maple 13 in 2009, is to sort the elements of a set in-place by a natural ordering and not by address, which of course is more expensive than sorting by address. The solution for polynomials introduced for Maple 4 was to provide the user with a `sort` command that sorts the terms of a polynomial (negative and fractional exponents are allowed) in descending order. Several orderings are supported. Lexicographical order is the default. So sort changes the unique representation of the polynomial without changing its hash value. We illustrate the problem and solution from Maple 16 or prior releases. Note that because the sort command operates in place, the value of `op(i,%)` may change. We remark that there is no technical reason why polynomials (monomials and sums) could not be sorted automatically using say a graded monomial ordering. Indeed, many users have wished that sorting was automatic. It's just much more expensive than sorting by address. The addition of POLY in Maple 17 (see Section 5) addresses this issue.

```
> f := expand( (x+1+z+y)^3 );
```

$$f := 1+3x+3z+3y+3x^2+6xz+6xy+3z^2+6zy+3y^2+3x^2z+3x^2y+3xz^2+3xy^2+3z^2y+3zy^2+x^3+z^3+y^3+6xzy$$

```
> sort(f,order=plex(x,y,z));
```

$$x^3+3x^2y+3x^2z+3x^2+3xy^2+6xyz+6xy+3xz^2+6xz+3x+y^3+3y^2z+3y^2+3yz^2+6yz+3y+z^3+3z^2+3z+1$$

```
> op(1,f);
```

$$x^3$$

4.2 Building up a sum, product, set, or list is $O(n^2)$.

Maple users often build sums, products, lists, and sets one item at a time. For example, to construct the polynomial $1 + 2x + 3x^2 + 4x^3 + \ldots + nx^n$ one obvious solution is the following loop.

```
f := 0;
for i to n do f := f + i*x^i end do;
```

As a second example, given a set S of $n \geq 0$ functions one may construct the set of derivatives using

```
T := {};  # empty set
for f in S do T := T union {diff(f,x)} end do;
```

This is is problem because in Maple, sums, products, sets and lists are all represented using vectors and therefore the above Maple codes cost $O(n^2)$ instead of $O(n \log n)$. This is because these loops are creating intermediate sums and sets of size $1, 2, 3, 4, \ldots, n-1$ that are all represented by vectors. Because Maple uses vectors, and because in a Maple session all intermediate objects are simplified and shared, Maple must create new objects rather than alter existing objects.

Is there a way to construct a sum of n terms or a set of n elements in $O(n \log n)$ time in Maple? Yes. The Maple designers have added several commands including `map`, `seq`, `add`, `mul` to enable the user to do this. The easiest way to create the sum $1 + 2x + 3x^2 + \ldots + nx^n$ in $O(n \log n)$ time is to use `add(i*x^i,i=1..n)`. This command builds the `SUM` in the Maple kernel in $\Theta(n)$ time without creating and simplifying intermediate sums. The sorting of the terms during simplification makes it $O(n \log n)$.

The easiest way to differentiate the functions in the set S in linear time is to use the `seq` command like this: `T := {seq(diff,t,x),t in S)}` or, to use the `map` command like this: `T := map(diff,S,x)`. In general one can compute and store a sequence of objects in an array or hash table and then convert it to a sum (or product or set or list). We illustrate by generating n monomials x^r where r is chosen at random from $[0, 10^3)$ and putting them into a hash table S using the monomials as keys.

```
R := rand(10^3);  # random number generator on [0,10^3)
S := table();     # create an empty hash table
for i to n do r := R(); S[ x^r ] := 1; end do;
S := {indices(S,nolist)}; # extract sequence of keys of S
```

But these solutions are not entirely satisfactory since the Maple user must be aware of the quadratic cost of coding using the simple for loop approach and know to avoid that.

How serious is this problem? We think it is quite serious since we have seen almost every Maple user make this mistake at some point. Our students make this mistake, usually when constructing a large list of data points, and not knowing any better, just wait for the program to finish. We have decided to turn this into a teaching point.

This problem is not unique to Maple. Other computer algebra systems, including Magma, Mathematica and Singular, also use vectors to represent objects and hence have the same quadratic efficiency problem. To verify this, in Appendix A we have timed Magma, Maple, Mathematica and Singular on adding a term to a polynomial, appending an element to a list, and inserting an element into set. These systems also offer similar solutions to Maple to address this problem, for example, Mathematica provides `Sum[i*x^i,{i,1,n}]` and `Map[D[#&,x],S]` that are linear in n.

That all systems have this quadratic problem seems surprising since there's really no reason why one could not use a balanced binary search tree to represent a set or a polynomial to make insertion of a single element and addition of a single term $O(\log n)$ instead of $O(n)$. We presume one reason this is not done for polynomials is because the designers did not want to allow destructive (inplace) operations on basic mathematical objects.

4.3 DAG representations need caches to be efficient.

Consider the following Maple session where we create the expressions $f = 2x^2 \sin x - 3 \sin x \cos x + 5 \cos x$ and $g = \beta y^2 - 3\beta + 1$ where $\beta = RootOf(z^2 - 2, 1.4)$.

```
> f := 2*x^2*sin(x)-3*sin(x)*cos(x) + 5*cos(x);
```

$$f := 2\,x^2 \sin(x) - 3\,\sin(x)\cos(x) + 5\,\cos(x)$$

```
> beta := RootOf(z^2-2,1.4);
```

$$\beta := RootOf\left(\_Z^2 - 2,\ 1.4\right)$$

```
> g := beta*y^2+3*beta+1;
```

$$RootOf\left(\_Z^2 - 2,\ 1.4\right) y^2 + 3\,RootOf\left(\_Z^2 - 2,\ 1.4\right) + 1$$

In both cases, because functions are represented uniquely, the sum-of-products representation is compact. But, consider what the following Maple commands must do

```
> degree(f,y), degree(g,y);
```

$$0,\quad 2$$

```
> evalf(g,10);
```

$$1.414213562\,y^2 + 5.242640686$$

When the `degree` and `evalf` commands scan the expressions f and g they see $\sin x$ twice, $\cos x$ twice and β twice. The degree command must check that the arguments of the functions $\sin x$ and $\cos x$ and β do not depend on y. If they do then the degree function will return FAIL. In this case this is not very expensive. However, in evaluating the RootOf to a floating point number, we do want to avoid doing this more than once since it involves computing a root of a polynomial to a given precision. In principle, all Maple commands like `degree` and `evalf` could benefit from using a cache or remember table to avoid recomputation when there are common subexpressions present.

For the `evalf(expr,d)` command, Maple has always had a dedicated remember table that depends on the precision d (digits). Thus when we ask to compute β to 10 digits of precision, if it was previously computed to 10 or more digits of precision, the value stored in the remember table is retrieved and rounded to the requested precision. Other remember tables and caches have been added to various Maple kernel commands but not in a systematic way. A simple last-in-first-out cache, even if small and even if only used for functions, can be very effective. Moreover, because of Maple's unique representation of objects, testing for equality with an object in the cache is a pointer comparison, so maintaining a small cache can be done very efficiently. Perhaps such caches should be added.

5 Efficient Polynomial Representations

What is the most important operation or family of operations to make fast in a computer algebra system? We argue that polynomial multiplication and it's sister operation polynomial division are important because (1) unlike operations like addition and differentiation that are linear in the number of terms, multiplication and division are non-linear in the number of terms and (2) many algorithms in computer algebra systems do many polynomial multiplications and divisions. Examples include algorithms for factoring polynomials (see Geddes et. al. [4]) and algorithms for computing determinants of matrices of polynomials (see Gentleman and Johnson [6]).

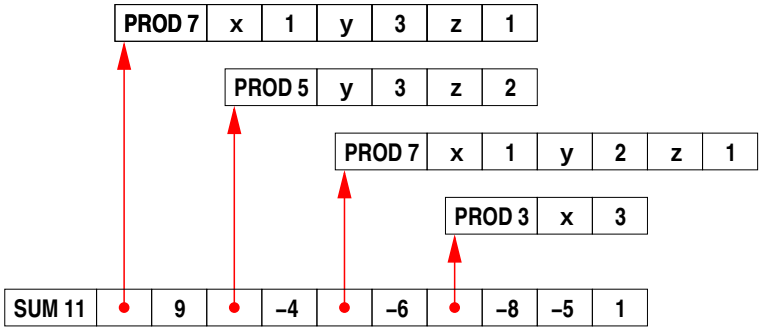

Figure 5: Maple's sum-of-products representation.

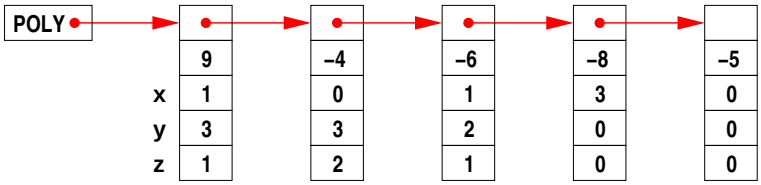

Figure 6: Singular's linked list representation.

Figures 5 and 6 show Maple and Singular's representations for the polynomial $f = 9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$. Looking at Maple's data structure, to multiply two monomials $xy^2z \times xy^3$ the reader can see that Maple must allocate a piece of storage large enough to store the result in a `PROD` then go through both monomials adding exponents of like variables. Since exponents can be fractions, there is a function call to decide how to add the exponents. After the product xy^5z is created, it is simplified and hashed to see if it is in Maple's `simpl` table as described in Section 2.1. We estimate that Maple takes over 200 cycles to multiply two monomials in 3 variables.

Multiplication in Singular is simpler. Again, a piece of storage must be allocated, but its size is known in advance. Then a simple loop adds the exponents that must be machine integers in Singular. We estimate that this can be done in about 50 cycles and as a consequence Singular is likely to be about about four times faster than Maple at multiplication and division of polynomials in 3 variables. The data in Table 1 confirms this.

5.1 The POLY data structure in Maple 17

Figure 7 below shows the new POLY data structure (see [10]) that we put into Maple 17 to improve performance for polynomials.

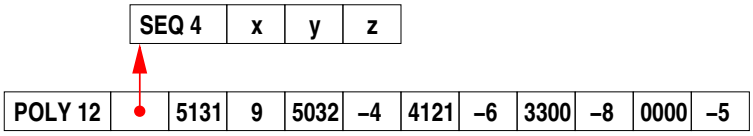

Figure 7: Maple 17's new POLY data structure for
the polynomial $f = 9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$.
It encodes the monomials $x^iy^jz^k$ in single words of memory.

The first word is a header word that encodes the length and type of the object. The second word points to the variables that are sorted in Maple's canonical ordering for sets. This is followed by the monomials and

coefficients, where the monomials encode the exponents and the total degree in a single machine word. For example, for $x^1y^3z^1$, on a 64 bit machine, we encode the four degree values $(5, 1, 3, 1)$ using 16 bits each as the integer $5 \cdot 2^{48} + 2^{32} + 3 \cdot 2^{16} + 1$. Terms are sorted into graded lexicographical order by comparing the monomials as unsigned integers. This gives a canonical representation for the polynomial. The current implementation requires all coefficients to be integers.

Six advantages of the new representation are readily apparent.

1. POLY is much more compact, about four times more compact on this example.
2. Monomial comparisons become machine word comparisons and monomial multiplication becomes machine word addition (provided there is no overflow), and monomial division becomes subtraction with a bitwise test for failure. This dramatically speeds up polynomial arithmetic.
3. Explicitly storing variables and sorting the terms lets us perform many common Maple idioms without looking at all of the terms, e.g. `degree`(f) (total degree), `indets`(f) (extract the set of variables), `has`(f, x), and `type`$(f, polynom)$.
4. Other operations such as `degree`(f, x), `diff`(f, x), and `coeff`(f, x, i) (extract the coefficient of x^i) access memory sequentially to make good use of cache. We can isolate groups of exponents using masks. This eliminates branching and loops at the level of the exponents.
5. For large polynomials, we avoid creating many small Maple objects (the `PROD`s) that must be simplified by Maple's internal simplifier and stored in Maple's `simpl` table. They fill the `simpl` table and slow down Maple's garbage collector.
6. Polynomials will automatically display terms in descending order making it easier for users to locate terms.

An obvious question is, what happens if the monomials cannot be stored in 64 bits? What we decided to do for Maple 17 is use the sum-of-products data structure instead of allocating more words for the monomials. That is, if *all* the monomials of the polynomial fit into one 64 bit word, Maple uses POLY, otherwise it uses the sum-of-products representation. Our thinking is that the monomials of most polynomials that arise in practice will fit in 64 bits. The following table shows, for n variables, how many bits b are available for each variable.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
b	21	16	12	10	9	8	7	6	5	5	4	4	4	4	3	3	3	3	3	2

Because we support two representations, there are conversions to and from the POLY and sum-of-products data structures. Also, for arithmetic between two polynomials in different variables, the monomials must first be repacked. All conversions and repackings are automatic and invisible to the Maple user. To be precise, Maple 17 uses the POLY data structure for a sum of terms if

(i) the coefficients are all integers,

(ii) the variables are Maple *names* with regular evaluation rules, e.g. $\{x, y_1, \pi\}$ but not *infinity* or *undefined* nor functions like $\sin x, f(1)$,

(iii) the number of variables $n < \beta/2$ for β-bit machines, and

(iv) the total degree d satisfies $1 < d < 2^b$ where $b = \lfloor \beta/(n+1) \rfloor$ for $n > 1$ and $b = \beta - 2$ for $n = 1$.

Otherwise the sum-of-products format is retained. Note, for polynomials with total degree $d = 1$, we do not store them in as a POLY dag because Maple's sum-of-products representation is better in this case. For example $f = 2x + 3y + 4z + 5$ is represented as | SUM 9 | x | 2 | y | 3 | z | 4 | 1 | 5 |. This is compact as monomials are not explicitly represented.

Another question is why we chose to store the total degree $i + j + k$ of the monomial $x^i y^j z^k$ as well as the individual degrees i, j, k. What is the advantage of using this graded ordering instead of a pure lexicographical order? For it seems to cost us some bits. Consider multiplying two polynomials $a \times b$. If the total degree $d = \deg a + \deg b$ does not overflow, that is, $d < 2^b$, then the entire product $a \times b$ can be computed without overflow and with no overflow detection needed. This allows us to look at only the leading terms of polynomials and predict overflow in $O(1)$ time.

Consider dividing two polynomials $a \div b$. In the division algorithm, if one uses pure lexicographical order, degrees in the remainder can increase and could cause overflow. For example, consider the following division

$$(x^2y^5 + y^3) \div (x^2y + xy^5)$$

in lexicographical order with $x > y$. The quotient is $y^4 = x^2y^5 \div x^2y$ and the remainder is $-xy^9 + y^3$. Notice the degree of y in the remainder has increased to 9. If we had 3 bits per variable, the y^9 would overflow. We would need a bit per variable to detect this efficiently. In contrast, when a graded ordering is used the total degree of the monomials that appear in the remainder always decreases. Therefore the degree in any individual variable cannot exceed the total degree. In our example the leading term of the divisor would be xy^5 and the division

$$(x^2y^5 + y^3) \div (xy^5 + x^2y)$$

would result in the quotient x and remainder $-x^3y + y^3$. Thus by using a graded ordering, we don't need extra bits to detect overflow because overflow cannot occur.

5.2 Hashing of POLY

When implementing POLY in the Maple kernel, we took advantage of its unique representation to make hashing of large polynomials sub-linear time. The hash function first hashes the degree, the variables, and the leading term of the polynomial. Then for a polynomial having t terms and length $2t + 2$, we compute an odd stride

$$s = 2\lfloor \sqrt[4]{(2t + 2)/256} \rfloor + 1$$

and hash every s word of the POLY dag, starting from the last, which is the trailing coefficient. For polynomials with fewer than 127 terms, $s = 1$ and we hash every word of the dag. For larger dags, the odd stride ensures that we alternate between hashing monomials and coefficients.

The reasoning behind this is that Maple often creates temporary objects with unique hashes, and we would like to reduce any linear time costs as much as possible. Any sequential overhead, particularly linear time overhead, reduces the potential for parallel speedup according to Amdahl's Law. In addition to hashing, computing the degree in x, testing for sub-expressions, and extracting coefficients in x, all run in sub-linear time by exploiting the degree field of the monomials to bound their search.

One thing that can not be done in sub-linear time is simplification. Maple must check that the monomials in the POLY dag are valid and sorted, and if the polynomial hashes to an existing value, Maple must compare the two objects in linear time. This however, is significantly faster than the comparison for the sum-of-products dag.

5.3 Multiplication, division and factorization benchmarks

We present one set of benchmarks to illustrate the gain made in Maple by the POLY data structure in Table 1. There are six multiplication benchmarks and six factorization benchmarks. The first benchmark

multiplies the polynomials $f_1 \times (f_1 + 1)$ where $f_1 = (1 + x + y + z)^{20} + 1$ then factors the product. This benchmark is taken from Fateman in [3]. Benchmark 2 replaces f_1 by $f_2 = 1 + x^2 + y^2 + z^2$ a polynomial that is sparser than f_1 to see if this makes any difference. Benchmarks f_3 and f_4 are larger versions of benchmark 1. Benchmark 5 is a completely dense benchmark and benchmark 6 is a very sparse benchmark.

The timings are in seconds. They were obtained on a Intel Core i7 920 computer, a quad core desktop computer running at 2.66 GHz.

Maple 13 uses the sum-of-products data structure only. The timings for Maple 16 show the improvement Maple obtains by using our C library for large multiplications and divisions where Maple converts from the sum-of-products to POLY, multiplies (divides) polynomials in the POLY representation using our (parallel) heap based algorithms in [9, 11], then converts the result back to a sum-of-products. This latter conversion can be expensive. For two sufficiently sparse polynomials, it takes more time to construct and simplify their product in the sum-of-products representation than it does to multiply them.

In Maple 17 where POLY is the default data structure in the Maple kernel, there are no conversions and our C library is also used for small polynomials. This is important because the algorithm used in polynomial factorization does many polynomial multiplications, both large and small. Note, the factorization code is the same in Maple 13, 16 and 17. So the improvements in Maple 16 and 17 are due solely to POLY and our C library.

	Maple 13	Maple 16		Maple 17		Magma	Singular	Mathem
multiply		1 core	4 cores	1 core	4 cores	2.19-1	3.1.4	atica 9
$p_1 := f_1(f_1+1)$	1.561	0.063	0.030	0.041	0.012	0.33	0.57	0.120
$p_2 := f_2(f_2+1)$	1.544	0.063	0.032	0.041	0.012	0.33	0.58	0.115
$p_3 := f_3(f_3+1)$	26.501	0.521	0.171	0.403	0.102	3.99	6.77	0.855
$p_4 := f_4(f_4+1)$	98.351	2.180	0.649	1.814	0.416	13.70	30.99	5.732
$p_5 := f_5 g_5$	13.666	1.588	0.384	0.153	0.153	13.24	18.22	1.526
$p_6 := f_5 g_5$	11.486	0.772	0.628	0.204	0.082	0.89	2.75	1.338
factor	uses Hensel lifting which is mostly polynomial multiplication							
p_1 12341 terms	31.33	2.79	2.66	0.79	0.65	6.51	2.01	18.48
p_2 12341 terms	275.51	3.24	3.07	0.99	0.85	7.09	2.10	112.43
p_3 38711 terms	360.86	16.71	14.11	6.93	4.40	119.32	12.48	276.16
p_4 135751 terms	286.39	59.01	46.15	24.35	12.73	320.04	61.85	951.72
p_5 12552 terms	302.45	26.44	16.15	12.13	6.80	105.55	13.83	935.15
p_6 417311 terms	1359.47	51.70	48.81	8.30	6.33	369.12	42.08	432.03

$f_1 = (1 + x + y + z)^{20} + 1$ 1771 terms
$f_2 = (1 + x^2 + y^2 + z^2)^{20} + 1$ 1771 terms
$f_3 = (1 + x + y + z)^{30} + 1$ 5456 terms
$f_4 = (1 + x + y + z + t)^{20} + 1$ 10626 terms
$f_5 = (1 + x)^{20}(1 + y)^{20}(1 + z)^{20} + 1$ 9261 terms
$g_5 = (1 - x)^{20}(1 - y)^{20}(1 - z)^{20} + 1$ 9261 terms
$f_6 = (1 + u^2 + v + w^2 + x - y)^{10} + 1$ 3003 terms
$g_6 = (1 + u + v^2 + w + x^2 + y)^{10} + 1$ 3003 terms

Table 1: Timings (in seconds) for polynomial multiplications and factorizations. Timings were obtained an Intel Core i7 920 quad-core desktop at 2.66GHz.

The improvement in Maple 16 over Maple 13 shows that by improving the performance of multiplication and division only, Maple moves from being the slowest system at factorization to the fastest! But, if you look at the parallel timings for Maple 16 closely, you will see that the parallel speedup for the multiplication benchmarks is not 4 on this quad-core desktop. For p_4 it is $2.180/0.649 = 3.36\times$. And the parallel speedup of the factorization of p_4 is poor. This is because of the cost of converting from POLY to the sum-of-products representation. In Maple 17 this conversion is eliminated. The parallel speedup for the multiplication is

now $1.814/0.416 = 4.3\times$ and for the factorization of p_4 is now $24.35/12.73 = 1.9\times$ which we think is a very good result. The best improvement in Maple 17 over Maple 16 is for the last benchmark, the sparse benchmark. Here the full advantage of POLY over the sum-of-products data structure is seen.

Note, the timings for Magma and Singular are for 1 core. They have no parallel algorithms. Mathematica is using all 4 cores for multiplication which is new in version 9 but the factorization code uses 1 core. Mathematica 8 and 7 do not have a parallel multiplication and their timings for these multiplication benchmarks are slower than Maple 13. Note, factorization of $p_2(x, y, z)$ in Maple 16, Maple 17, Magma 2.19 and Singular 3-1-4 proceeds by first factorizing $p_2(x^{1/2}, y^{1/2}, z^{1/2})$ which is effective. Maple 13 and Mathematica 9 do not do this. Note, for the dense multiplication benchmark p_4 where the Maple 17 time is 0.153s, there is no parallel speedup because a serial FFT based algorithm was chosen.

5.4 Maple 18 Work

For Maple 17, the number of bits we allocated for the total degree is the same as the number of bits for each variable's exponent. So for example, for $n = 5$ variables, we allocate $\lfloor 64/6 \rfloor = 10$ bits per variable and 10 bits for the total degree. So the largest degree is 1023 and 4 bits are unused. At that time we did not think it was worth the extra coding effort needed to use those extra bits for total degree. The following table shows for n variables how many bits are for each variable (row b) and how many extra bits are available for the total degree (row x).

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
b	21	16	12	10	9	8	7	6	5	5	4	4	4	4	3	3	3	3	3	2
x	1	0	4	4	1	0	1	4	9	4	12	8	4	0	13	10	7	4	1	20

Notice that in the range of $n = 8$ to 14 variables, Maple could benefit significantly if we used the extra bits for total degree. For Maple 18 we have recoded our algorithms to make use of these extra bits for the total degree. This was painful but it appears to have been worthwhile. As one example of a practical problem where this increases substantially the range of problems that POLY can handle consider again the problem of computing $\det(V_n)$ where V_n is the $n \times n$ Vandermonde matrix (see Section 2.2). The determinant $\det(V_n)$ when expanded as a polynomial has degree $n - 1$ in each variable and total degree $n(n-1)/2$. Table 3 below shows that if we use the extra bits for the total degree, the range for which $\det(V_n)$ can be represented with the POLY data structure increases from $n = 9$ variables to $n = 14$ variables, a large increase. Column $\deg(\det(V_n))$ shows the total degree $n(n-1)/2$ of $\det(V_n)$. Shown also in the table is the time it took Maple 16, 17 and Maple 18 to compute $\det(V_n)$. The reader can see that Maple 18 computes $\det(V_n)$ for $n = 11$ and $n = 12$ quite quickly. Note, $\det(V_{13})$ has 13! terms. It is too big to compute on our machine as it requires $13! \times 2 \times 8 = 99.6 \times 10^9$ bytes to store in the POLY representation. The number of terms 13! also exceeds Maple's maximum object size of $2^{31} - 1$ terms because the header word uses 32 bits for the length (in words) of an object.

	per variable		total degree		Timings for computing $\det(V_n)$			
n	#bits	maxdeg	**#bits**	maxdeg	$\deg(\det(V_n))$	Maple 16	17	18
7	8	255	8	255	21	0.012s	0.005	0.004
8	7	127	8	255	28	0.093s	0.027	0.026
9	6	63	10	1023	36	1.35 s	0.218	0.150
10	5	31	14	16383	45	15.95s	25.44	1.57
11	5	31	9	511	55	–	–	18.87
12	4	15	16	65535	66			236.4
13	4	15	12	4095	78			–
14	4	15	8	255	91			
15	4	15	4	15	105			
16	3	7	16	65535	120			

Table 2: Real times in seconds to compute $\det(V_n)$ in Maple 16, 17 and 18.
Timings were obtained on a Intel Core i7 2600 running at 3.40 GHz.

6 Conclusion

Over the course of a large project like the development of Maple, thousands of design decisions are made. Some decisions are borrowed from other systems, some are evolutionary, some are revolutionary, and some have undesirable consequences.

The Maple sum-of-products data structure for mathematical expressions uses vectors. Each object and sub-object in a Maple session is represented uniquely. These two design decisions have efficiency advantages but also disadvantages that Maple users and Maple programmers have lived with for many years. This article has presented some of the solutions that we and others have implemented to address these disadvantages.

Currently we are extending the POLY data structure so that it allows the coefficients to be fractions and floating point numbers as well as integers and so that the variables may also be functions like $\sin x, f(1), RootOf(z^2-2, 1.4), etc.$ This will solve the problem with using a DAG representation identified in Section 4.3.

Acknowledgement

The authors wish to thank the referees for their detailed suggestions for improving this article.

References

[1] Bosma, W., Cannon, J., Playoust, C., 1997. The Magma Algebra System I: The User Language. *J. Symb. Cmpt.* **24**(3-4), 235–265. See also `http://magma.maths.usyd.edu.au/magma`

[2] Bruce W. Char, Keith O. Geddes, W. Morven Gentleman, Gaston H. Gonnet.
The design of maple: A compact, portable and powerful computer algebra system. *Proceedings of EUROCAL '83*, 101–115, Springer-Verlag, 1983. Accessible as Research Report CS-83-06, https://cs.uwaterloo.ca/research/tr/1982/CS-82-40.pdf.

[3] Fateman, R., 2003. Comparing the speed of programs for sparse polynomial multiplication. *ACM SIGSAM Bulletin* 37 (1), pp. 4–15.

[4] Keith O. Geddes, Stephen R. Czapor, George Labahn. *Algorithms for Computer Algebra*, Kluwer, 1992.

[5] Keith O. Geddes, Gaston H. Gonnet, Trevor J. Smedley. Heuristic Methods for Operations with Algebraic Numbers. *Proc. of ISSAC '88*, ACM Press, 475–480, 1988.

[6] Gentleman, W.M., Johnson, S.C. Analysis of Algorithms, A Case Study: Determinants of Matrices with Polynomial Entries. *ACM Trans. on Math. Soft.*, **2**(3), pp. 232–241, 1976.

[7] Greuel, G.-M., Pfister, G., Schönemann, H., 2005. Singular 3.0: A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra, University of Kaiserslautern. http://www.singular.uni-kl.de

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Addison Wesley, 2nd edition, 1981.

[9] Michael Monagan and Roman Pearce. Parallel Sparse Polynomial Multiplication using Heaps. *Proceedings of ISSAC '09*, pp. 263–269, ACM Press, 2009.

[10] Michael Monagan and Roman Pearce. POLY: A new polynomial data structure for Maple 17. Extended Abstract, *Communications in Computer Algebra* **46**(4), 164−167, 2012.

[11] Roman Pearce and Michael Monagan. Parallel Sparse Polynomial Division using Heaps. *Proceedings of PASCO '2010*, ACM Press, pp. 105–111, 2010.

[12] Douglas R. Stinson. *Cryptography: Theory and Practice* Chapman & Hall, 3rd edition, 2006.

Appendix A

Maple, Mathematica, Magma and Singular code for creating the sum $x+2x^2+\ldots+dx^d$, the list $[1,2,\ldots,d]$ and the set $\{1,2,\ldots,d\}$ using loops. Note, Singular has no set data structure.

Maple

```
S := 0:
d := 4000:
st := time():
for i from 1 to d do S := S+i*x^i od:
time()-st;

S := []:
d := 4000:
st := time():
for i to d do S := [op(S),i] od:
time()-st;

S := {}:
d := 4000;
st := time():
for i to d do S := S union {i} od:
time()-st;
```

Mathematica

```
f = 0;
d = 4000;
Timing[ Do[ {f = f+i*x^i}, {i,1,d} ] ]
```

```
S = {};
d = 4000;
Timing[ Do[ {S = Append[S,i]}, {i,1,d} ] ]

S = {};
d = 4000;
Timing[ Do[ {S = Union[S,{i}]}, {i,1,d} ] ]
```

Magma

```
Z := IntegerRing();
P<x> := PolynomialRing(Z);
f := 0;
d := 4000;
time for i in [1..d] do f := f+i*x^i; end for;

Z := IntegerRing();
d := 4000;
L := [];
time for i in [1..d] do L := Append(L,i); end for;

d := 4000;
S := {};
time for i in [1..d] do S := Include(S,i); end for;
```

Singular

```
ring R=0,(x),lp;
poly S;
int i,d,st;
S = 0;
d = 4000;
st = timer;
for (i=1; i <= d; i++) { S = S+i*x^i; }
timer-st;

list S;
int i,d,st;
S = list();
d = 4000;
st = timer;
for (i=1; i <= d; i++) { S = insert(S,i); }
timer-st;
```

	Maple 18			Magma 2.19-6			Mathematica 9			Singular 3.1-4		
d	sum	list	set	sum	list	set	sum	list	set	sum	list	set
4000	0.278	0.179	0.371	0.13	0.10	0.26	1.19	0.052	0.438	0.15	0.62	NA
8000	1.008	0.651	1.474	0.52	0.33	1.03	5.07	0.216	1.932	0.57	2.46	NA
16000	3.949	2.597	5.841	2.02	1.32	4.17	18.58	0.946	7.375	2.22	9.70	NA

Timings in CPU seconds on an Intel Core i7 920 at 2.67 GHz
showing that the time taken by all systems is quadratic in d.

Appendix B

Maple, Magma, Singular, and Mathematica code used for the first benchmark.

Maple

```
kernelopts(numcpus=4);
f1 := expand( (1+x+y+z)^20 )+1:
p1 := CodeTools[Usage]( expand( f1*(f1+1) ) ):
CodeTools[Usage]( factor(p1) ):
```

Magma

```
Z := IntegerRing();
P<x,y,z> := PolynomialRing(Z,3);
f := (1+x+y+z)^20+1; g := f+1;
time h := f*g;
time ff := Factorization(h);
```

Singular

```
system("--ticks-per-sec", 100);
ring R=0,(x,y,z),lp;
poly f = (1+x+y+z)^20+1; poly g = f+1;
int TIMER = timer; poly p = f*g; timer-TIMER;
    TIMER = timer; list L = factorize(p); timer-TIMER;
```

Mathematica

```
f = Expand[(1+x+y+z)^20]+1;
AbsoluteTiming[p1 = Expand[f*(f+1)];]
AbsoluteTiming[h1 = Factor[p1];]
```