A Comparison of Two Symbolic Computation Packages

Matt Briggs

June, 1996

Contents

1 Introduction

The purpose of this paper is to present for analysis the implementations of two symbolic computation packages, *SiMath* and *Pari*. Specifically, the data structures each package uses for implementation of basic types such as integers, polynomials, matrices, and algebraic number fields will be compared and contrasted. The programmatic interface, that is, the interface to a package for a programmer wishing to use the package as a library of new C routines and data types, will also be analyzed for each package.

2 Overview of SiMath

2.1 Core System

SiMath's internal structure is centered around the concept of a dynamic *workspace* from which *lists* and *atoms* may be allocated and used for the representation of data types and then returned to the workspace by a garbage collector. This is quite reminiscent of how `LISP` interpreters work. Indeed, *SiMath* consists of a number of routines for performing operations on lists.

2.1.1 The Workspace

All memory allocation in *SiMath* takes place from a memory pool called the workspace. The workspace itself is implemented as a linked list of available memory cells which may be allocated internally by the *SiMath* library routines or by user routines, and another linked list of memory cells that are allocated. Cells in this later list have an additional status bit that indicates whether or not they are actually in use. In this way, the user may allocate memory cells as needed but he *isn't* responsible for freeing them.

2.1.2 Garbage Collection

For any memory allocation request, the garbage collector is invoked. It may also be called manually by the user. Once invoked, the garbage collector begins by scanning the list of memory cells that are already allocated. As cells are encountered that are marked as not being in use, they are returned by the garbage collector to the list of available cells. When enough cells have been returned to the available list to account for the original allocation request, or the end of the list of allocated cells is encountered, the process stops.

If after this process is complete enough memory cells *still* don't exist to satisfy the allocation request, the workspace is allowed to grow to accommodate the request. More memory cells are dynamically created and added to the available list, which are then used to satisfy the allocation request. There is a configurable maximum size to which the workspace is allowed to grow, so that it isn't possible for the memory requirements to grow arbitrarily large.

It should be noted that *SiMath* in essence performs a garbage collection each time it receives an allocation request. This has the effect of reusing old memory cells before new ones, so that the frequency of garbage collections is increased but the duration of each is cut down.

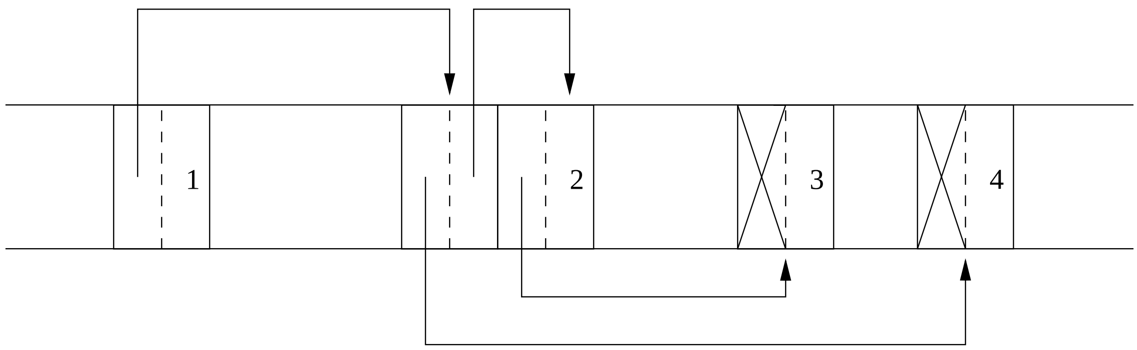
2.1.3 Objects, Atoms, and Lists

The core types in *SiMath* are recursive in definition. The simplest, non-recursive type is the *atom*, which is an integer x with $|x| < \mathrm{BASIS}$, where $\mathrm{BASIS} = 2^{30}$. A *list* is either empty or a finite, ordered sequence of *objects*. An *object* is either a list or an atom.

An object has two components:

1. a pointer to the next object if the object is in a list, or `NIL` if it is the last object in the list or not in a list at all
2. a data block for storing its value if the object is an atom, or a pointer to the first object in the list if the object is a list.

As an example, the list (1(23)4) would be laid out in memory as



2.1.4 Manipulating Lists

SiMath provides a number of useful functions for creating and altering lists of objects. By providing a core set of list manipulation routines, *SiMath* doesn't have to provide specific functions for manipulating each particular data type that it provides. As long as the user is familiar with the structure for a particular data type, they can then use these routines to alter a variable of the type as much as they wish.

Specifically, routines are provided for:

1. parsing input string and creating a list from that string, where the list in the string uses `LISP` notation
2. creating a list from a fixed number of objects
3. the equivalent of the `CAR` and `CDR` functions found in `LISP`
4. concatenating lists, deleting arbitrary elements from lists, and searching a list for a particular element

2.2 Type Representations

All types in *SiMath* are built from lists.

2.2.1 Integers

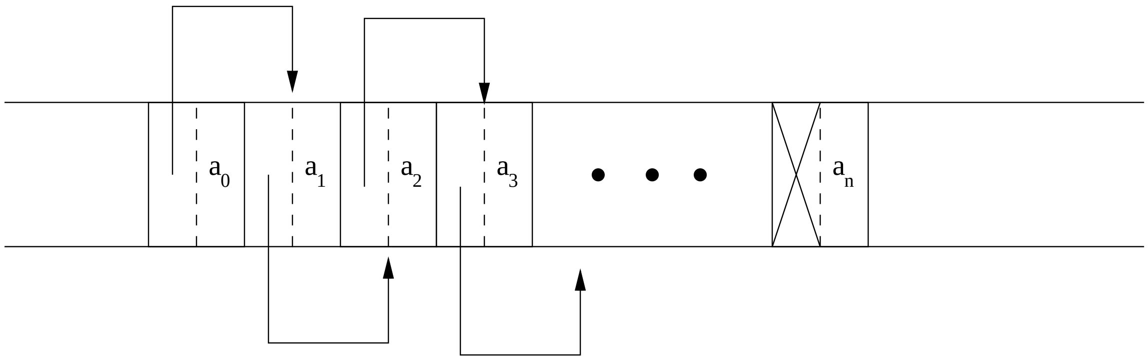
For any integer $n \in \mathcal{Z}$ *SiMath* stores n using base BASIS. Thus, n is represented as a sequence of digits $a_0, a_1, \ldots, a_k$ with

$$n = \Sigma_{i=0}^{k} a_i \cdot \mathrm{BASIS}^i$$

and the following constraints

1. $k \geq 0$, $a_i \in \mathcal{Z}$, $|a_i| <$ BASIS
2. $a_i = 0$ or $\text{sign}(a_i) = \text{sign}(n)$
3. $a_k \neq 0$ if $k > 0$

In *SiMath* each a_i would be stored as an atom and n itself would be a list of these atoms, as follows:x



2.2.2 Modular Integers

Internally, *SiMath* stores modular integers in exactly the same fashion as it stores integers. Functions that perform operations on modular integers, then, must be supplied with a modulus for the numbers that will be operated on. It's up to the user to ensure that the modular integers stay within the bounds of the modulus they are using.

2.2.3 Polynomials

SiMath offers three different representations of polynomials, called the distributive representation, recursive sparse representation, and the recursive dense representation.

In the distributive representation, the list for a polynomial is made up of pairs of objects, with the number of pairs equal to the number of monomials in that polynomial. The first object in a pair is the coefficient for the monomial, and the second object in the pair is a list of the exponents for each variable of the polynomial.

For example, if we have

$$p(x, y) = -xy^3 + y^3 + 9xy^2 + 27x^5y$$

then this would be stored as

$$(-1, (3, 1), 1, (3, 0), 9, (2, 1), 27, (1, 5))$$

In the recursive dense representation of a polynomial, the first object in the list is an atom that contains the degree of the polynomial. The second object in the list is the coefficient of the term of exponent the highest degree. The third object in the list is the coefficient of the term of exponent the highest degree, less one. This pattern continues. Note that if there is no term in the polynomial of a particular degree, then the coefficient for that term will be stored as 0.

For example

$$\begin{aligned} p(x,y) &= -xy^3 + y^3 + 9xy^2 + 27x^5y \\ &= y^3(-1x^1 + 1x^0) + y^2(9x^1 + 0x^0) + y^1(27x^5 + 0x^4 + 0x^3 + 0x^2 + 0x^1 + 0x^0) + 0y^0 \end{aligned}$$

so that the polynomial would be stored as

$$(3, (1, -1, 1), (1, 9, 0), (5, 27, 0, 0, 0, 0, 0), 0)$$

Finally, the recursive sparse representation stores a polynomial as a list with pairs of objects. The first object in a pair represents the exponent of the term of the polynomial, and the second object represents the coefficient for that term.

For example

$$\begin{aligned} p(x,y) &= -xy^3 + y^3 + 9xy^2 + 27x^5y \\ &= y^3(-x^1 + x^0) + y^2(9x^1) + y(27x^5) \end{aligned}$$

so that we have

$$(3, (1, -1, 0, 1), 2, (1, 9), 1, (5, 27))$$

as the representation for the polynomial.

3 Overview of Pari

3.1 Internals

At its core, *Pari* has a fixed-size stack from which it can allocate variables of its provided types. Data types can be thought of as tree like structures, where the user is responsible for allocating the appropriate amount of memory for each structure and subsequently initializing those components. The tree structure of the data types allows for easy implementation of recursive types.

3.1.1 The Generic Type GEN

The C type declaration `GEN` is provided to the user for declaring variables of *Pari* data types. `GEN` is basically a pointer to a block of memory, with a portion of that memory being used to store information about the variable and the rest of the memory being used for actual storage for the variable's value. Blocks of memory that are used to record information about the variable are called *code words*, and each code word occupies four bytes of storage.

The `GEN` type provides one code word that is common to all *Pari* data types, since they are all inherited from `GEN`. This code word contains three components:

1. the type of the data, i.e. integer, rational, polynomial, etc..., occupying one byte
2. the reference count, for garbage collecting purposes, occupying one byte
3. the length of the root in 4 byte blocks, occupying two bytes

The byte of data immediately following the codeword in memory marks the beginning of the actual data for a variable. Note that different data types may store additional codewords if they are needed, in addition to the data for the variable's storage.

3.1.2 Integers

Integers are stored in base 2, so that for instance to store a number with at most 100 decimal digits, one would need

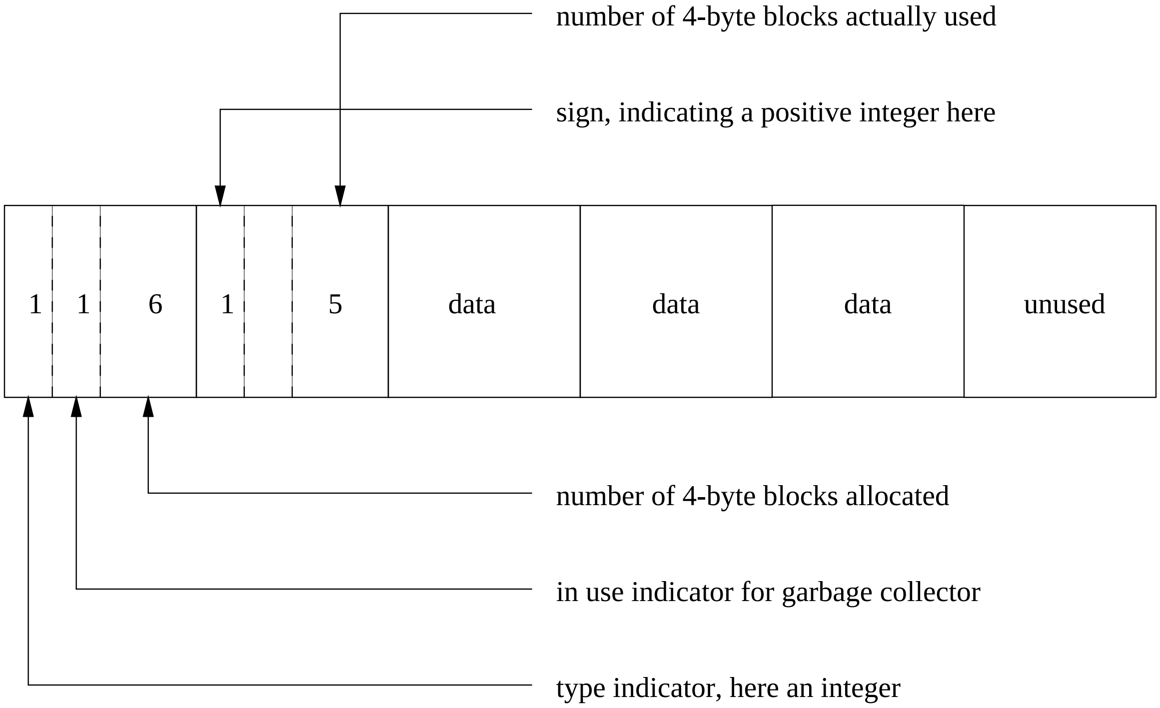
$$\log_2 10^{100} \approx 332.19$$

or 333 bits of storage, which comes out to 42 bytes or eleven 4-byte blocks. The maximum size of the integer, then, is stored in the first code word, while an additional codeword is used to record two pieces of information:

1. the sign of the number, with a value of 1 if the number is positive, -1 if the number is negative, and 0 if the number if zero.
2. the "effective" size of the integer, representing the number of code words actually needed to hold the integer

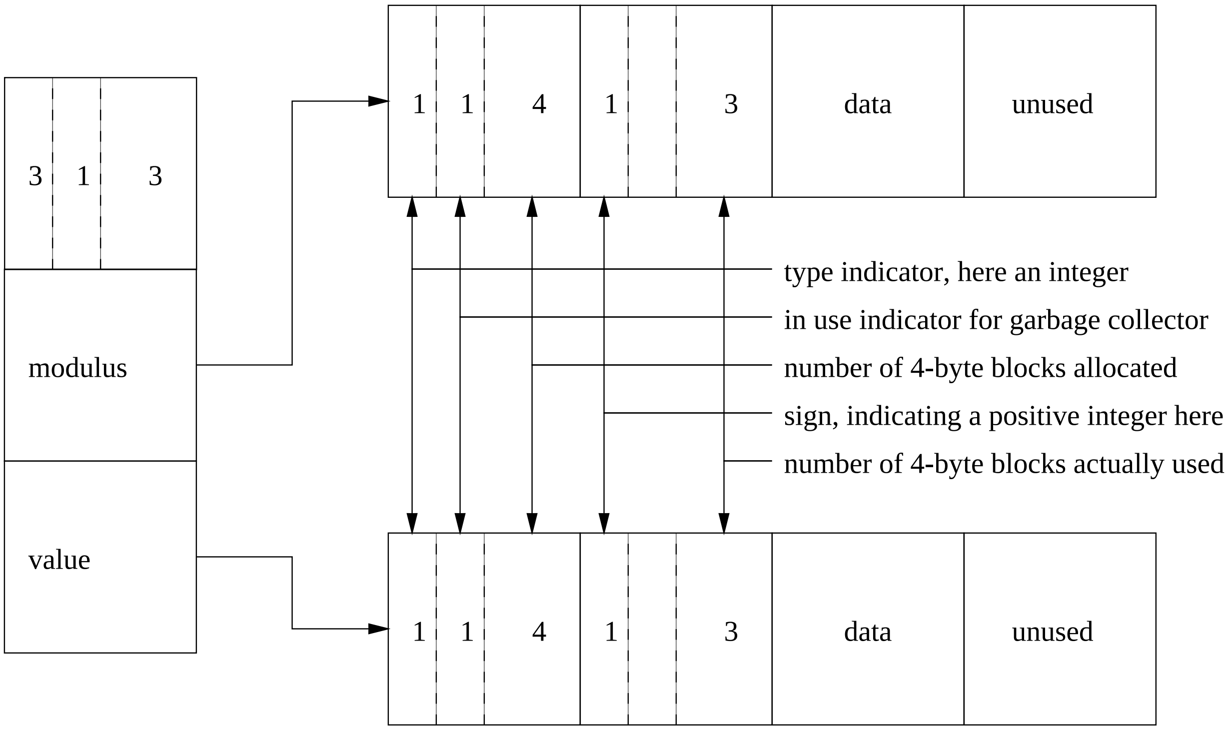
This later field is used so that *Pari* can correctly handle the situation where the actual value stored in an integer requires fewer digits than are maximally available for that particular variable.

For example, an integer variable that uses 5 out of 6 possible code words would be stored as:



3.1.3 Modular Integers

Modular integers have two fields, both of which are pointers, in addition to their code word. The first field points to the modulus for the integer, and the second field points to the class representative of that modulus for the integer. These pointers would normally be pointers to *Pari* integers, which allows for great flexibility in that arbitrarily large modulii may be used.



3.1.4 Polynomials

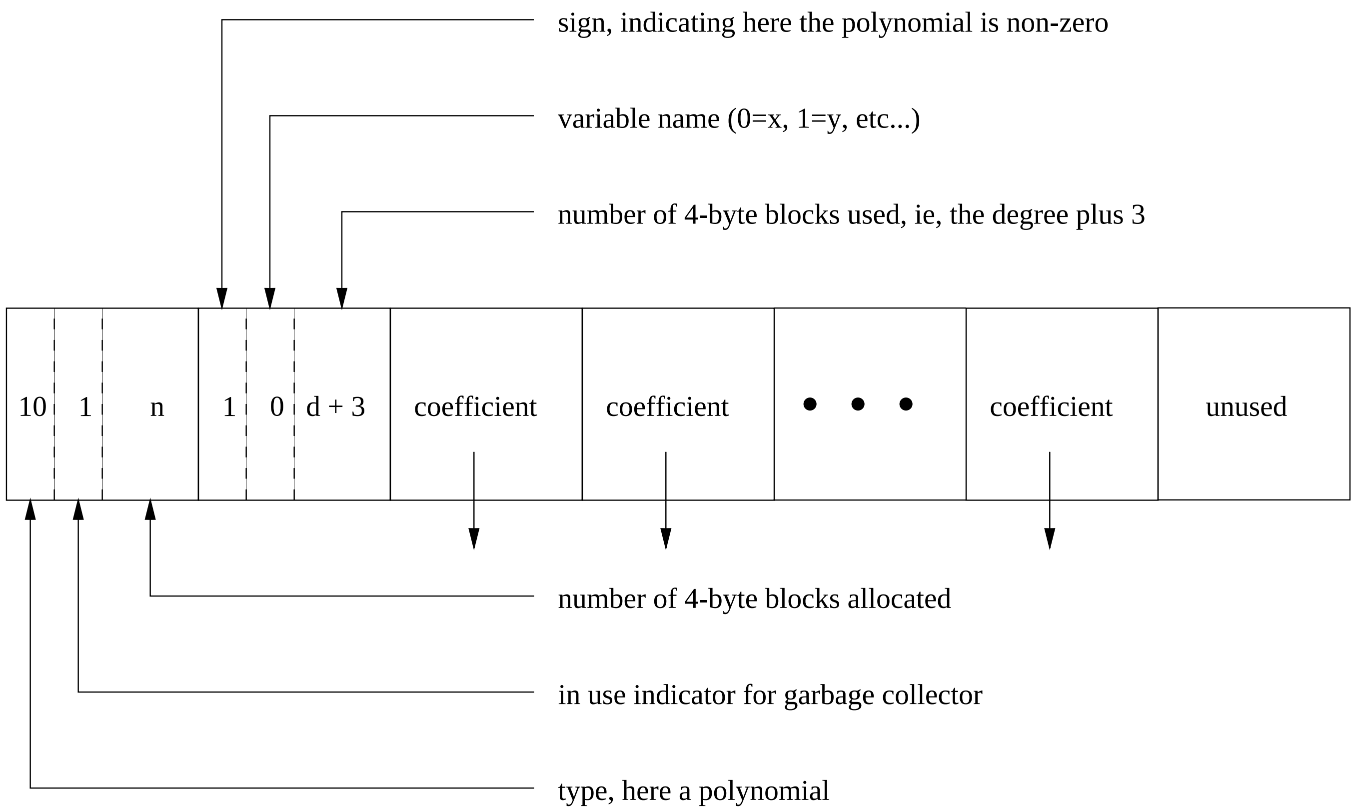
The polynomial implementation in *Pari* is similar to the dense, recursive implementation in *SiMath*. An additional codeword is used for polynomials that specifies:

1. the "sign" of the polynomial, a value of 0 if the polynomial is 0 or 1 if it is not

2. the variable number for the polynomial (0 for x, 1 for y, etc...)

3. the effective length of the polynomial which, as is the case with integers, tells exactly how many data bytes are valid if the size of the polynomial is less than its maximum

The remaining portion of the polynomial is a series of pointers to the coefficients for the terms, starting with the coefficient for the constant term and ending with the coefficient for term of the highest degree. Note the flexibility in that these coefficient pointers could point to integers, modular integers, rationals, or any other type in *Pari*. For instance, if the coefficients were themselves polynomials then in effect we would be representing a polynomial of more than one variable.



4 Implementation Differences

There are a number of issues which are approached in different ways by *Pari* and *SiMath*. Here we will break down some of the more interesting differences into two levels for consideration:

1. *programmatic differences*: Differences in the programming interfaces provided for each package, such as how variables of package-defined types are output, and how aware a programmer using the library must be of memory allocation and deallocation strategies.

2. *data structure differences*: Differences in each package's implementation of some of the fundamental mathematical constructs they provide, such as arbitrarily precise integers, polynomials, and number fields.

By considering the way each package implements a common mathematical mechanism or object we can understand in real-world terms the problems that arise in the implementation of such things, and the solutions that make their implementation feasible.

4.1 Programmatic Differences

Note that all references to the "user" in this section refer to a programmer using either *SiMath* or *Pari* for access to new data types and functions, rather than a high-level user who would be using `GP` or `SimCalc`.

4.1.1 Producing Output

Pari and *SiMath* take fundamentally different approaches to how variables of the package-defined types should be output to the user. In common, both packages provide the ability to output a variable in a canonical form to either the terminal or a file.

Pari provides several functions for printing out variables, with each function producing output in a different form or to a different destination. Since *Pari* only has the single `GEN` type, it is able to use only a small number of functions to print its entire range of types, which is beneficial to the user in that they only have to remember a small number of function names as compared to other packages which provide differently named output functions with similar functionality for each data type. In addition, *Pari* has the ability to output a variable to a string, which although probably seldom needed is a nice convenience for the user.

SiMath's approach aims at making it more natural for C programmers to print variables of new types by adding new formating directives to `printf` and `fprintf`. The preprocessor run by *SiMath* scans through the user's C source code and changes occurrences of the aforementioned function calls with the new directives to output routines for variables of the necessary type. As with *Pari* this is a good system since the user is freed from having to remember names of different output functions based on the types. However, the use of a preprocessor is a cumbersome extra step. In addition, the code the user writes that uses *SiMath* isn't C but rather an extended version of C that has non-standard capabilities added to `printf` and `fprintf`.

4.1.2 Creation of Variables

Initializing variables in *SiMath* is straightforward as long as the user knows beforehand the list structure for the types they are creating. The user simply calls any one of the number of list manipulation routines provided by *SiMath* to construct a list in the form required for whatever type they intend on initializing.

Pari on the other hand is very limited in this respect. The user begins by allocating *Pari* stack space for the root node in the tree, and then continues the process of allocating memory and initializing subtrees of the tree as needed, depending on the complexity of the data structure that is being initialized. For a polynomial or matrix, this can be quite tedious indeed.

This is one of the few places where *SiMath*'s choice of data structures beats out that of *Pari*. By providing the user data types based on lists and a number of powerful routines to manipulate such lists, *SiMath* lets the user easily and intuitively construct constants for those variables "by hand" in the program.

Another alternative to doing this initialization by hand is to have a constant string in the program and then call a package function that initializes a variable with that string. In *Pari* this is quite powerful as the function provided that parses the string is actually the same parser used by `GP` to evaluate expressions.

4.1.3 Type Checking

Pari provides the user with several new types, but it does so through one single C structure called `GEN`. This causes some difficulty in the area of type checking. Since all variables of any of the package-provided data types are seen to be of type `GEN` by the compiler, compile time type checking is rendered ineffectual. To distinguish between its types, *Pari* reserves a portion of the `GEN` structure for typing information. This leaves it to the library functions of *Pari* to verify that they are taking the correct parameters at runtime. Depending on how often this is done, a significant amount of slow down can result in the overall computation. The alternative would be not to do any runtime type checking whatsoever. This later option could be dangerous, as it assumes that the user is providing the data in the format the function expects. On the other hand, since *Pari* gives the user the ability to easily manipulate the data structures for variables anyway, the user could fake out *Pari* by creating a bogus data structure and setting the type portion of the `GEN` structure to a value *Pari* expects. Thus, the typing information for *Pari* objects is used more for generic functions that call more specific functions based on the types of the objects, rather than for explicit type checking.

SiMath on the other hand does provide separate C type definitions for each of its new data types, and as such compile-time checking is available for the user. This has the dual benefits of ensuring that the user is actually using the types as expected by the library functions, and not requiring any type of runtime checking as would be required in a package like *Pari* were it to do that kind of checking.

4.1.4 Overloaded Functions

One of the main attractions of *Pari*'s system of using a single `GEN` type to represent multiple types is that there only needs to be a single function for each major function that the different types would have in common. Thus, whether the user wants to add polynomials, matrices, vectors, or integers, he would still use the same function. The function called by the user acts as a "dispatcher" function, in that it checks the type of its `GEN` parameters and then calls an appropriate routine that will add polynomials, integers, or whatever other types can be added.

There are three immediate benefits to this system. First and foremost, it cuts down dramatically on the number of function names the user has to be familiar with. Other packages get around this by having solid naming conventions that intermix the data type with the function being performed, but nothing is simpler than having to remember a single name that works regardless of the type. Secondly, if the user changes his code to use rationals instead of integers, for instance, he need not change the call to the operations he's performing on those objects. This cuts down a great deal on maintenance of the code. And finally, when new types are added to the *Pari* system, the amount of change in the library routines provided to the user is minimal since few new functions would have to be provided *to the user* to access those functions (although new ones *would* have to be provided that are callable by the higher level functions that *are* available to the user).

Comparatively, *SiMath*'s typing system is much weaker. Since data types are essentially just lists, library functions have no idea what the lists they are operating on are supposed to represent unless those functions *assume* the lists are representing a particular type. Thus, functions applicable to each type have to be created and made available to the user, so that for instance there is no longer a single function to add objects. Instead, there is a function that adds and assumes that the objects it is working on are integers, and another function that assumes its objects are rationals when it adds them, etc...For the same reasons that *Pari*'s system was appealing, the system presented by *SiMath* and other packages is much less attractive. If the user changes objects from one type to another, all the function calls that operate on those types must be changed. In addition, the user is often required to thumb through documentation or online help to remember the exact name for the operation he wants to perform.

4.1.5 Memory Handling

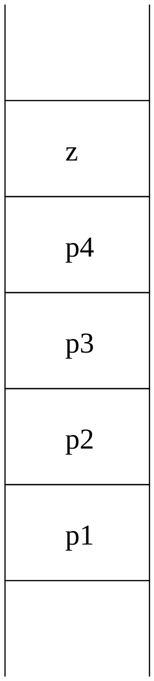
SiMath has a clear advantage in its workspace-list implementation compared to *Pari*'s stack-tree concept. In *SiMath* all garbage collection is done automatically by the system, although the user can intervene at any stage and force garbage collection. In addition, there are no constraints on when memory blocks can be freed or where in the allocation heap they have to be. *SiMath* is also flexible in that it allows its workspace to grow, up to a maximum value, if the garbage collection process is unable to find enough free cells for the next allocation. Overall, this makes the process of memory allocation as transparent to the user as possible, letting them focus instead on their programming task.

Pari on the other hand puts garbage collection directly in the hands of the user. The primitive function it provides for returning an allocated object to the stack only works if the object being returned is on the top of the stack. Thus, if a calculation is performed that puts intermediate results on the stack first and then stores a final result on the stack the intermediate results use memory that can't be recovered. The idea is to keep as few memory blocks as possible and get rid of intermediate results as soon as they are no longer needed. To do this, *Pari* requires the user to record the location of the top of the stack before any intermediate results are placed on the stack, then record the top of the stack again after all intermediate results are used. When the calculation's final result is on the top of the stack, the user forcibly calls a garbage collection routine that flushes memory blocks from the previously recorded stack positions and then moves the rest of the stack contents up to meet the final result that is at the top. Needless to say, this is quite cumbersome and tedious and is a serious detractor from *Pari*. Not only is it necessary to do this manual garbage collection, but it is also often necessary for the user to reorder their calculations so that all the intermediate results appear contiguously on the stack so that the garbage collection will actually work.

As an example, consider a computation to compute the two element vector $(x*x+y, y*y+x)$:

```
p1 = multiply(x,x)
p2 = add(p1, y)
p3 = multiply(y,y)
p4 = add(p3, x)
z = vector(p2, p4)
```

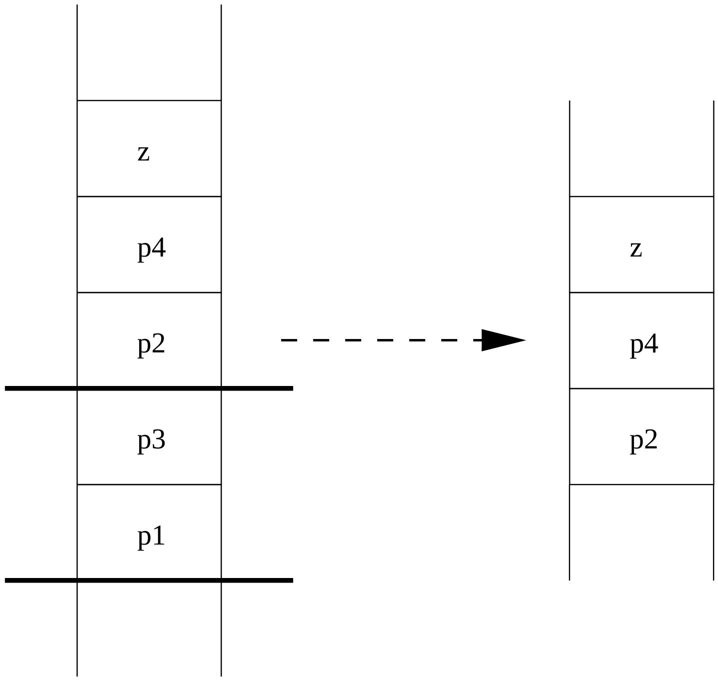
which would leave the stack in the following configuration



If however the calculation is rearranged as

```
s1 = stacktop()
p1 = multiply(x,x)
p3 = multiply(y,y)
s2 = stacktop()
p2 = add(p1, y)
p4 = add(p3, x)
z = vector(p2, p4)
garbagecollect(s1, s2)
```

then the stack appears in the configuration below and to the left, so that after the garbage collection routine is called the stack is reduced to the configuration on the right



4.2 Data Structure Differences

4.2.1 Integers

Pari's implementation is slightly better than that of *SiMath* simply because of its underlying tree structure. Recalling that *SiMath* uses a list of digits in base BASIS, note that each digit has associated with it not only a value but also a pointer to the next digit in the overall integer. This adds up to a lot of overhead, basically twice the size on a 32-bit computer of the same integer if no pointers had to be used. In addition, the algorithms that work with such integers would naturally be slower since they are continually traversing linked lists. On the other hand, the linked list representation of integers in *SiMath* allows for true arbitrary precision, whereas with *Pari* the size of a large integer is fixed at allocation time (although it may be resized at a later time, this is still not taken care of automatically as it is with *SiMath*). This ability to grow arbitrarily large isn't usually needed though, in that it is often sufficient just to work with an integer that can be arbitrarily large, up to a maximum, so that *Pari* is quite adequate.

4.2.2 Modular Integers

The only difference structurally between modular integers in *SiMath* and *Pari* is that *Pari* actually stores the modulus with the integer and further ensures that the value of the integer stays within the range of 0 to one less than the modulus. Storing the modulus facilitates sanity checking in *Pari*'s library routines, while forcing the value of the integer to be within legal boundaries ensures that functions that assume this condition indeed are presented with it.

SiMath takes the other approach and treats modular integers as regular integers, without storing any modulus with the integer and not ensuring that it it stays within boundaries. Not storing the modulus is somewhat cumbersome in that for any functions that act on modular integers, the user must always supply the modulus. In addition, if routines in *SiMath* assume that the integers stay within the legal bounds for a particular modulus and the modular integer supplied as a parameter isn't within that bound, such routines could crash the *SiMath* system. This is less than ideal.

On the other hand, if the user is conscientious about making sure his modular integers stay within bounds, then *SiMath* is somewhat more flexible in that it allows the same variable to be considered as an element of any number of residue class rings.

4.2.3 Vectors

SiMath makes no distinction between column vectors and row vectors, while *Pari* does. In terms of storage, *Pari* treats the two types of vectors exactly the same and functions that operate on vectors will operate on either type, although such functions will verify that their parameters are both of the same type. Although not essential, this distinction does allow for slightly finer grained type checking in *Pari*, allowing it, for example, to make sure if a matrix and a vector are being multiplied, with the matrix on the left hand side of the operation, that the vector is a row vector.

4.2.4 Polynomials

SiMath is by far more intelligent in its implementation of polynomials, offering up three different representations for any single polynomial, while *Pari* limits itself to just one. The motivation is that certain algorithms will perform better when they use one representation of a polynomial compared to another, equivalent representation. Thus, *SiMath* provides functions for converting between its different representations for polynomials, along with operations on those polynomials that take as a representation the one that is the most efficient for that particular operation.

A somewhat better strategy, although more costly memory-wise, is to store a polynomial as *all* of its representations. Then, whenever a particular operation is called to act upon that polynomial, the operation uses whichever representation of the polynomial is the most efficient for that algorithm. This saves the time-consuming task of actually *converting* between the operations as is currently required in *SiMath*, which in effect causes at least a partial loss in the speedup obtained by using several representations.

4.2.5 Number Fields

Pari's implementation of number fields is slightly more intelligent than that of *SiMath* in that it precomputes several structures associated with an algebraic number field such as the discriminate, integral basis, and the root of the minimal polynomial. *SiMath* on the other hand simply stores the minimal polynomial for the number field and then computes other information as necessary. *Pari*'s decision has the disadvantage of initially using much more memory than *SiMath* would for a number field, but it also saves time computationally when information it has precomputed is needed. In addition, *Pari* uses more sophisticated algorithms than *SiMath* for doing computations in number fields that takes advantage of the precomputed information to greatly speedup the calculations.

5 Conclusions

From the previous discussion regarding the different implementation aspects of each system, we note the following points:

1. Wherever possible it is best to store additional information within the structure for a particular data type. This is almost accomplished in *SiMath* with polynomials, in that three

different representations of polynomials are provided so that algorithms which are more effective on particular representations may be used. In *Pari* this concept was illustrated in the implementation of algebraic number fields, where several components of a number field are created when it is initialized and then stored along with the minimal polynomial. Algorithms for performing computations in number fields are then able to use these precomputed components without having to recompute them themselves each time the algorithm is called.

The benefit for such a scheme is speedup, while as expected the price for such speedup is higher memory utilization.

2. *Pari*'s typing scheme is ideal, where several new data types are dynamically associated with a single C structure. Such a system cuts down on the number of library functions the user must be familiar with, allowing the same function names to be used regardless of the actual type. An additional benefit is very low code maintenance if the data objects in a program change, since the function names are the same regardless of the object's type. The one drawback to the scheme is that compile-time checking of data types isn't possible, so that either no type checking or run-time checking is needed. The later option can slow down computations, and the former option is rather dangerous.

3. Library functions should be provided that make the initialization of data objects easier. In particular, *Pari* provides little in this arena, basically forcing the user to create the tree for an object by hand. Not only is this tedious, but also quite error prone. If the user isn't intimately familiar with how the data types are implemented, he can very easily forget a portion of the initialization, leading to stack corruption and eventually disastrous effects. Ideally, library routines or macros would be provided for each major type that allow the creation of such a type and any kind of manipulation of that type's components. For instance, a routine to create a polynomial of a particular degree over a given algebraic ring would be useful, along with routines to access the coefficients in the polynomial. As was seen in the *Pari* example in section 4.1.2, this can be quite cumbersome in a package that doesn't provide such library routines.

4. Memory allocation and deallocation should either be transparent to the user or intuitively obvious. *SiMath*'s system of garbage collection is the ideal. In such a system, objects are allocated by the user as needed and then forgotten about, with *SiMath* maintaining reference counts for the objects so that when they are no longer being referenced the objects may be disposed of, while the user remains completely obvious. If the user is rather sloppy, this can lead to large amounts of memory being allocated that isn't really in use by the user's program, and further make garbage collections prohibitively long. An alternative is to make the user responsible for the allocation and deallocation of memory objects, in a way analogous to how C handles dynamic memory.

   No matter what scheme is chosen, though, it should *not* be a burden to the user and it should *not* be difficult to follow. Having either or both of these properties is a recipe for the user making a mistake and destroying the stack. In this respect, *Pari* is an excellent example of what not to do. The constraint that the user reorder steps of his computation so that intermediate values can be kept in contiguous order on the stack is burdensome, and the additional requirement that the user actually keep track of the top of stack at various points and then *manually* call a garbage collector to flush this garbage is absurd.