1 copy
P.M.C.

An informal report on SMP

An informal report on SMP

Stephen Wolfram
The Institute for Advanced Study, Princeton NJ 08540.

(January 1983)

ABSTRACT

An informal report on the present status of SMP is given. Some aspects of the usage and internal construction of SMP are discussed.

Paper for presentation at the European Computer Algebra Conference EUROCAL '83, in London, England, on March 28-31, 1983.

SMP is a general purpose symbolic manipulation language developed by me and several coworkers* starting in 1980, mostly at Caltech. A full description of SMP is given in the SMP Handbook [1]. The architecture of SMP was briefly outlined in [2]. This paper gives an informal discussion of the present status of SMP, its use and internal construction.

A test site version of SMP has existed for nearly a year, but its widespread distribution was delayed by administrative problems at Caltech [3]. About ten test versions have been running at various sites for about six months. Distribution to well over a hundred academic organizations is now imminent. In addition, sales of SMP to commercial organizations have recently been started. Only a small amount of new code has been added in the past six months, but availability of commercial funds promises to allow more extensive development to be started in the near future.

SMP was originally developed on a VAX running the UNIX operating system. It nows runs on a variety of VAX 11/780 and 11/750 systems under various Berkeley and Bell versions of UNIX. It was at first ported to run under the VMS operating system on the VAX using the EUNICE UNIX emulator; however, distribution and technical problems forced an alternative approach. An essentially complete version of SMP has now been created using the Whitesmith's C compiler under VMS. One may anticipate that a version of SMP will soon exist for the SUN Microsystems MC68000-based workstation running UNIX (and possibly for other MC68000-based systems). The current limitation on this system to 2 megabytes of memory per process (making very large SMP calculations impossible) will apparently be lifted by mid-1983.

The text of SMP is about 1.2 megabytes in size (corresponding to 100000 lines of C source code). A typical SMP job requires about 0.5 to 1 megabyte of working data memory. It has been found that a VAX with more than about 1.5 megabytes of physical memory, and a demand paging operating system, can accomodate several simultaneous SMP jobs.

With a few exceptions (discussed below) the current version of SMP contains a full implementation of all the features described in the SMP manual. The internal code is modular and mostly well-commented, so that maintainence has been comparatively easy. Bugs are by now rather rare; nearly all user problems result from incorrect usage of the SMP language, rather than internal problems. About 250 external files, written in the top-level SMP language, and implementing additional features, are now available, and span a broad area in mathematics, applied mathematics, and physics. A recent innovation is a scheme for labelled comments in the SMP-readable files, which allows the text of the files to be typeset in an easily-assimilable form, and permits features and keywords in the files to be accessed by the SMP database system. So far most external files have been written by system developers; users often create rather general programs but are reluctant to perfect them and format them for general distribution. Commercial incentives should however change this attitude.

Considerable effort has been expended to provide good documentation for SMP. The SMP Handbook is divided into three sections: a summary, giving a short description of all facilities; a reference manual, containing the same text as the summary, but with extensive examples; and a primer, giving a pedagogical introduction to SMP. This format has been very well-received. Most users start by reading the primer; once proficient in SMP, they use the summary and

* SMP was mostly designed by me. C.Cole, J.Greif, T.Shaw and A.Terrano made important contributions in its implementation.

reference manual for reference. (Some prefer the shorter form of the summary; others look immediately at the examples in the reference manual.) A rather complete, hand-generated, keyword index was included in the manual, though in the first edition it was given only with the reference manual, and not with the summary, and was largely unused.

The SMP language is sufficiently simple that new users are able to perform meaningful calculations after only a few minutes of reading the primer and experimentation with SMP. Perhaps because the SMP language is different in structure from most computer languages, new users with no previous computing experience seem to be at little disadvantage. Users almost always start by using only built-in functions, and sometimes also functions in external files (which they usually learn about from experienced users). Only later do they begin to define their own functions.

The SMP language is strongly based on pattern matching, and allows many definitions to be given in close to their mathematical form. Users with no computing experience seem to understand this approach intuitively, and very quickly define complicated systems based on pattern matching. Users with experience in more conventional languages such as FORTRAN or MACSYMA often try to define conventional functions, using conditionals and other standard programming constructs, and find it more difficult to adapt to the pattern matching approach. The simple specification of the operation of the SMP pattern matcher is crucial in allowing users to apply it effectively. Complications or heuristics would confuse users. Once users have understood pattern matching, they often use it almost to the complete exclusion of many built-in functions. For example, users often make pattern-matching definitions to perform transformations on expression which could more easily and more efficiently have been made using existing built-in functions.

When SMP is given an expression which it considers undefined, it simply returns the expression unevaluated. It almost never prints an "error message". Users are very rarely confused by this, and often appreciate the succint output they receive. When SMP encounters a syntax error on input, it immediately reports this and places the user in a simple line editor, indicating the problematic token. Users find it very easy to recognize the appropriate corrections, and respond well to the absence of derogatory verbeage. The most common non-trivial confusion in SMP usage occurs when circular pattern definitions have been given, and the complete processing of an input line would take an infinite time. Such problems are usually recognized by frequent status interrupts, which print the top of the stack of SMP projections being evaluated, and reveal the circular behaviour. It should be noted that the infinite evaluation scheme which makes infinite loops possible is crucial to the basic operation of SMP, and usually behaves exactly as users require.

SMP has almost no system-defined global variables (except the lists of input and output expressions, and the preprocessor and postprocessor templates). The lack of global variables and switches has avoided innumerable user problems.

Most of the internal code of SMP is by now very efficient. Much of this efficiency can be traced directly to the low-level nature of some of the code and the presence special purpose internal data structures, made possible by the use of a low-level language such as C, rather than a higher-level language such as LISP. Absolutely crucial to the efficiency of SMP are internal counters and pointers which ensure that an expression, once simplified, is almost never re-simplified unless new definitions may have been given for some of its parts. Also crucial are a number of related mechanisms for sharing common

subexpressions, and representing different occurrences of the same expression by a single structure in memory. The basic SMP internal data structure uses arrays rather than linked lists. Thus insertion of new elements in lists requires a complete copy; however, this can be achieved by a single machine instruction on the VAX, and is therefore very quick. Any additional overhead is by far offset by the proximity of list elements in memory; in a linked list, different list elements may be in different pages of virtual memory, and their retreival may be very time-consuming. The array nature of the SMP internal data structures, together with sharing of common subexpressions, typically causes SMP expressions to be stored rather compactly in memory. This tendency is enhanced by a compacting garbage collector; the garbage collector in some cases in fact constructs a hash table and explicitly shares common subexpressions in memory. SMP calculations have been performed with over 10 megabytes of intermediate data; the compactness of expressions leads to good paging behaviour, with page faults typically generated in sharp bursts, separated by lengthy in-core computations. The use of many special-purpose internal data structures makes asynchronous compacting garbage collection at arbitrary times impossible in SMP. Several crucial routines exist in two versions: one less efficient one in which all data structures are maintained in a form which allows garbage collection, and a more efficient one which does not permit intermediate garbage collection.

The internal processing of an expression proceeds in four basic steps:

1. Input, lexical analysis and parsing
2. Evaluation of system-defined functions
3. Pattern matching to apply user definitions
4. Printing of output.

The lexical analyser and parser are based on the UNIX utilities *lex* and *yacc* respectively. Extreme care was taken in defining the precedence and associativity of operators, and it is believed that, unlike almost all other languages, the SMP grammar corresponds correctly with common mathematical usage. In keeping with common notation, the parser does not require an explicit star between expressions to represent multiplication. A macro preprocessor is available in SMP, and acts before the lexical analyzer. This preprocessor is particularly valuable in overcoming problems with special and escape characters. Syntax extensions are also initiated in the macro preprocessor, thereby allow operators which would usually be considered part of a single lexical token to be identified. The parser is very efficient, and in fact it has turned out that when SMP expressions are to be stored and re-input from disk files, they are better stored in top level form and reparsed, than stored in a representation of internal form, and relocated on input.

The first action of the simplifier on an expression is to test a counter to determine whether new assignments which might affect the expression have been made since it was last simplified; if they have not, the simplifier replaces the expression by the last simplified value to which it points. If they have, or if the expression is newly input, the simplifier processes each part of the expression in turn. All values and properties of all symbols are stored in a symbol table (dynamically allocated and indexed by hashing). When symbols represent system-defined functions, the symbol table contains a pointer to a C text routine which evaluates the function. There are about a hundred basic system-defined functions, together with several hundred mathematical functions. Simplifiers for **Mult** and **Plus** are probably the most commonly called; great care was therefore taken to use optimal sorting algorithms. Both system and user-defined functions can carry properties such as **Flat** or **Trace**. At the user level, these properties

are stored in a property list. Internally, at the time of assignment, bits are set in a special bit field in each symbol table entry to allow rapid testing.

The final stage of simplification consists in applying user-defined rules to expressions by pattern matching. Despite its comparatively simple specification, the code of the pattern matcher is long and complicated, mainly because of numerous optimizations. The lists which represent values for projections contain hash codes for their indices to allow faster comparisons and searches.

The printing form of symbols (and thus projections or functions) are specified by pointers from the symbol table to text routines or user-defined SMP expressions. The primary difficulty of two-dimensional printing is associated with breaking long expressions into several lines. SMP constructs a sequence of rectangular boxes to represent parts of expressions; each box is assigned a breaking factor, and the final boxes are shaped and placed so as to minimize the total breaking factor. User-defined printing forms are usually implemented through the two-dimensional format statement function **Fmt**, which specifies relative positions of expression boxes.

The functions **Graph** and **Plot** yield graphics output. On standard terminals, the output is given using ordinary characters; this is usually quite adequate to obtain a qualitative picture. Special code was included to provide genuine graphics output on several devices; in practice, almost all graphics terminals seem to support Tektronix 401X codes. Rather complete code now exists in SMP for two-dimensional graphics. Contour plots are now available, but complete hidden-surface three-dimensional plots are still not available. The primary reason for this is that, while a line in two dimensions may be represented by a list of points, it is not clear how to make an efficient top-level SMP representation of a three-dimensional surface.

A recent addition to SMP were the functions **L** and **At**, which allow semantic display editing of expressions within SMP. The functions read terminal control codes at initialization time.

It is sometimes thought that the bulk of a symbolic manipulation program consists of programs implementing mathematical algorithms, such as those for symbolic integration. In fact, most of the internal code of SMP implements the very many other functions and operations required to treat expression structurally. Nevertheless, SMP contains an increasingly complete set of mathematical algorithms. Expansion of polynomials (and other distributivity operations) is one of the most important functions. A crucial feature in the implementation of expansion is intermediate simplification of partially expanded results; in this way, whenever cancellations occur, many fewer terms are generated, and an otherwise exponential intermediate expression swell is avoided. Polynomial factorization in SMP is performed using a version of Berlekamp's algorithm. Fast multivariate polynomial factorization, using Zippel's algorithm, is largely implemented, but is not included in the main version of SMP. An implementation (by Terrano) of Risch's algorithm for symbolic integration has recently been completed. Taylor-Laurent, Pade and continued fraction series expansions are implemented by a recursive scheme which ensures that irrelevant terms are never generated.

An important decision made in the implementation of SMP was the internal (but not external) representation of numbers in a double-precision floating point form. The rationale for this was that in practical calculations numbers requiring more than 10 of the 16 available decimal digits are rare, and arithmetic operations on machine floating point numbers are orders of magnitude faster than if implemented in software for a direct representation of, say, rational numbers. The lack of fast floating point hardware for existing MC68000-based computers

slightly decreases the speed advantage in this case. Arbitrary precision numbers in SMP are represented as functions containing a sequence of ordinary numbers. It appears that in most calculations, it is quite clear at the outset whether arbitrary precision numbers will be required or not, so that the calculation may be set up appropriately, and the presence of different number types is not unduly inconvenient.

Operations on objects such as big numbers are performed in SMP through a type extension mechanism. The actual operation of, for example, a **Plus** projection is determined by the types of its arguments. An unsatisfactory (but hitherto often taken) approach consists in including tests and dispatching code in the standard **Plus** function. Since in SMP special data types are always manifest, being represented by specific projections, special expressions may be identified and tagged by bits in the internal representation. The main simplifier then tests these bits (in parallel with several other tests), and if necessary searches the property lists of the special data types (using hashing) to find an alternative function by which the standard **Plus** should be replaced in the particular case. This approach is satisfactory for unary functions, but for multinary functions with several different special types as arguments, it is inadequate. A good general approach yet to be implemented in this case would be to define a network of coercion transformations, each with an associated cost, and then to find a minimum cost routing in the network to transform all the special data types to a common ancestor, on which the unary function approach can be used. The main difficulty in implementing such a scheme is to find a convenient representation of the network in top-level SMP. The network mechanism could also be used to control application of, for example, trigonometric transformations.

SMP is essentially an interpreter. However, the function **Cons** (recently unbundled into the functions **Prog**, **Code** and **Load**) allows definitions and programs written in top-level SMP to be translated into C, on the assumption that all variables have only numerical values. **Cons** translates not only expressions, but also control structures and even some simple pattern-matching processes. The C code it generates can be compiled, then linked into a running SMP job. The code is read into the data space of the job, then accessed through tables inserted in the text space; this mechanism can be used under most operating systems. The compiled code typically runs at least ten and often many more times faster than top-level interpreted code. The ability to generate and test code in the high-level SMP language, then automatically translate it to efficient low-level C, provides the essential elements of automatic programming so much discussed in other contexts. Translation of SMP code into FORTRAN is straightforward given the existing **Cons** function, and will soon be implemented.

The SMP language was essentially designed to be used on a simple line-at-a-time display terminal. SMP commands are therefore input in a direct linear form. Availability of computers such as the SUN workstation, which allow for high-speed high-resolution full-screen input and output, suggest many enhancements. The first of these, now largely implemented, is an interactive front-end for a database management system containing information on system and external file functions, indexed by keywords and key phrases. An important element of the system is an algorithm for taking an English phrase, and stripping its structure and words to canonical roots which may be retreived by hashing from the database. In this way, natural language queries may be responded to in an interactive fashion. A further enhancement along the same lines would be to allow for interactive menu input of all SMP commands. Such a menu can be controlled by the same codes in documentary text as are required for the query system. Use of windows, mice, and other display-oriented mechanisms would provide more complete interactive input and output.

It is now about three years since SMP was started, and I have altogether spent about a year working on SMP and managing the project. All in all, I at least am quite satisfied with the progress of SMP in that time. If I were to start the project again, almost all the things I would do differently are of an administrative, rather than of a technical nature. For the last year or so, I have used SMP extensively to solve problems in theoretical physics (the purpose for which it was originally created). I suspect it will not be too long before SMP has saved me more time than it took to build it, and the project will therefore have been at least a partial success. $